Proceedings

30th Annual Computer Security Applications Conference

ACSAC 2014

Proceedings

30th Annual Computer Security Applications Conference

New Orleans, Louisiana, USA
8–12 December 2014

Sponsored by
Applied Computer Security Associates

The Association for Computing Machinery
2 Penn Plaza, Suite 701
New York, New York 10121-0701

ACM COPYRIGHT NOTICE. Copyright ©2014 by the Association for Computing Machinery, Inc. Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Publications Dept., ACM, Inc., fax +1 (212) 869-0481, or permissions@acm.org.

For other copying of articles that carry a code at the bottom of the first or last page, copying is permitted provided that the per-copy fee indicated in the code is paid through the Copyright Clearance Center, 222 Rosewood Drive, Danvers, MA 01923, +1-978-750-8400, +1-978-750-4470 (fax).

Notice to Past Authors of ACM-Published Articles
ACM intends to create a complete electronic archive of all articles and/or other material previously published by ACM. If you have written a work that was previously published by ACM in any journal or conference proceedings prior to 1978, or any SIG Newsletter at any time, and you do NOT want this work to appear in the ACM Digital Library, please inform permissions@acm.org, stating the title of the work, the author(s), and where and when published.

ACM ISBN: 978-1-4503-3005-3

Editorial production by Adam Hahn
Published by ACM, Inc. within the ACM International Conference Proceedings Series

Table of Contents

Technical Program v
Message from the Conference Chair x
Message from the Program Chairs xi
Organizing Committee xii
Steering Committee xii
Program Committee xiii
External Reviewers xiii
Professional Development Course Proposal Reviewers xiv
ACSA Members xv
Message from the Sponsor xvi
Contributing Sponsor xvii

Technical Program

Invited Essayist

Taking Two-Factor to the Next Level: Protecting Online Poker, Banking, Healthcare and Other Applications 1
Aviel D. Rubin

Secure Builds and Configuration

Hot-Hardening: Getting More Out of Your Security Settings 6
Sebastian Biedermann, Stefan Katzenbeisser, Jakub Szefer

Challenges and Implications of Verifiable Builds for Security-Critical Open-Source Software 16
Xavier de Carné de Carnavalet, Mohammad Mannan

Cloud Radar: Near Real-Time Detection of Security Failures in Dynamic Virtualized Infrastructures .. 26
Sören Bleikertz, Thomas Groß, Carsten Vogel

Mobile Systems Security I

Scippa: System-Centric IPC Provenance on Android 36
Michael Backes, Sven Bugiel, Sebastian Gerling

Android Security Framework: Extensible Multi-Layered Access Control on Android 46
Michael Backes, Sven Bugiel, Sebastian Gerling, Philipp von Styp-Rekowsky

Towards a Salable Resource-driven Approach for Detecting Repackaged Android Applications 56
Yuru Shao, Xiapu Luo, Chenxiong Qian

Social Computing and Networks

TROGUARD: Context-Aware Protection Against Web-Based Socially Engineered Trojans 66
Rui Han, Alejandro Mesa, Mihai Christodorescu, Saman Zonouz

Spam Ain't As Diverse As It Seems: Throttling OSN Spam With Templates Underneath 76
Hongyu Gao, Yi Yang, Kai Bu, Yan Chen, Doug Downey, Kathy Lee, Alok Choudhary

A Taste of Tweets: Reverse Engineering Twitter Spammers 86
Chao Yang, Jialong Zhang, Guofei Gu

Systems Security

New Models of Cache Architectures Characterizing Information Leakage from Cache Side Channels .. 96
Tianwei Zhang, Ruby B. Lee

ICE: A Passive, High-Speed, State-Continuity Scheme 106
Raoul Strackx, Bart Jacobs, Frank Piessens

Interrupt-oriented Bugdoor Programming: A Minimalist Approach to Bugdooring Embedded Systems Firmware 116
Samuel Junjie Tan, Sergey Bratus, Travis Goodspeed

Cyber Physical Systems I

Through the Eye of the PLC: Semantic Security Monitoring for Industrial Processes 126
Dina Hadžiosmanović, Robin Sommer, Emmanuele Zambon, Pieter Hartel

CPS: Market Analysis of Attacks Against Demand Response in the Smart Grid 136
Carlos Barreto, Alvaro Cárdenas, Nicanor Quijano, Eduardo Mojica-Nava

CPS: Driving Cyber-Physical Systems to Unsafe Operating Conditions by Timing DoS Attacks on Sensor Signals 146
Marina Krotofil, Alvaro Cárdenas, Bradley Manning, Jason Larsen

Secure Distributed Systems

Uncovering Network Tarpits with Degreaser 156
Lance Alt, Robert Beverly, Alberto Dainotti

Network Dialog Minimization and Network Dialog Diffing: Two Novel Primitives for Network Security Applications 166
M. Zubair Rafique, Christophe Huygens, Juan Caballero, Wouter Joosen

Lightweight Authentication of Freshness in Outsourced Key-Value Stores 176
Yuzhe Tang, Ting Wang, Ling Liu, Xin Hu, Jiyong Jang

Securing Memory and Storage

SEER: Practical Memory Virus Scanning as a Service 186
Jason Gionta, Ahmed Azab, William Enck, Peng Ning, Xiaolan Zhang

MACE: High-Coverage and Robust Memory Analysis For Commodity Operating Systems 196
Qian Feng, Aravind Prakash, Heng Yin, Zhiqiang Lin

Assisted Deletion of Related Content 206
Hubert Ritzdorf, Nikolaos Karapanos, Srdjan Capkun

Mobile Systems Security II

Morpheus: Automatically Generating Heuristics to Detect Android Emulators 216
Yiming Jing, Ziming Zhao, Gail-Joon Ahn, Hongxin Hu

Design and Implementation of an Android Host-based Intrusion Prevention System 226
Mingshen Sun, Min Zheng, John C.S. Lui, Xuxian Jiang

MoRePriv: Mobile OS Support for Application Personalization and Privacy 236
Drew Davidson, Matt Fredrikson, Benjamin Livshits

Secure Communications

IMSI-Catch Me If You Can: IMSI-Catcher-Catchers 246
Adrian Dabrowski, Nicola Pianta, Thomas Klepp, Martin Mulazzani, Edgar Weippl

Advanced Wi-Fi Attacks Using Commodity Hardware 256
Mathy Vanhoef, Frank Piessens

Whitewash: Outsourcing Garbled Circuit Generation for Mobile Devices 266
Henry Carter, Charles Lever, Patrick Traynor

Usable Security

Using Automatic Speech Recognition for Attacking Acoustic CAPTCHAs: The Trade-off between Usability and Security 276
Hendrik Meutzner, Viet Hung Nguyen, Thorsten Holz, Dorothea Kolossa

Understanding Visual Perceptions of Usability and Security of Android's Graphical Password Pattern 286
Adam J. Aviv, Dane Fichter

It's the Psychology Stupid: How Heuristics Explain Software Vulnerabilities and How Priming Can Illuminate Developer's Blind Spots 296
Daniela Oliveira, Marissa Rosenthal, Nicole Morin, Kuo-Chuan Yeh, Justin Cappos, Yanyan Zhuang

Privacy

Exploring and Mitigating Privacy Threats of HTML5 Geolocation API 306

Hyungsub Kim, Sangho Lee, Jong Kim

Differentially Private Data Aggregation with Optimal Utility 316
Fabienne Eigner, Aniket Kate, Matteo Maffei, Francesca Pampaloni, Ivan Pryvalov

On the Privacy Provisions of Bloom Filters in Lightweight Bitcoin Clients 326
Arthur Gervais, Ghassan Karame, Damian Gruber, Srdjan Capkun

Network Infrastructure Security

OSPF Vulnerability to Persistent Poisoning Attacks: A Systematic Analysis 336
Gabi Nakibly, Adi Sosnovich, Eitan Menahem, Ariel Waizel, Yuval Elovici

Less is More: Cipher-Suite Negotiation for DNSSEC 346
Amir Herzberg, Haya Shulman, Bruno Crispo

DNS Authentication as a Service: Preventing Amplification Attacks 356
Amir Herzberg, Haya Shulman

Access Control and Malware

Relation Extraction for Inferring Access Control Rules from Natural Language Artifacts 366
John Slankas, Xusheng Xiao, Laurie Williams, Tao Xie

Centrality Metrics of Importance in Access Behaviors and Malware Detections 376
Weixuan Mao, Zhongmin Cai, Xiaohong Guan, Don Towsley

Scalability, Fidelity and Stealth in the DRAKVUF Dynamic Malware Analysis System 386
Tamas Lengyel, Steve Maresca, Bryan Payne, George Webster, Sebastian Vogl, Aggelos Kiayias

Software Security

Towards Automated Integrity Protection of C++ Virtual Function Tables in Binary Programs 396
Robert Gawlik, Thorsten Holz

Leveraging Semantic Signatures for Bug Search in Binary Programs 406
Jannik Pewny, Felix Schuster, Christian Rossow, Lukas Bernhard, Thorsten Holz

IntFlow: Improving the Accuracy of Arithmetic Error Detection Using Information Flow Tracking . . . 416
Marios Pomonis, Theofilos Petsios, Kangkook Jee, Michalis Polychronakis, Angelos D. Keromytis

Cyber Physical Systems II

CPS: Beyond Usability: Applying Value Sensitive Design Based Methods to Investigate Domain Characteristics for Security for Implantable Cardiac Devices 426
Tamara Denning, Batya Friedman, Brian Gill, Daniel B. Kramer, Matthew R. Reynolds, Tadayoshi Kohno

A Security Evaluation of AIS Automated Identification System 436
Marco Balduzzi, Alessandro Pasta, Kyle Wilhoit

Web Security

NodeSentry: Least-Privilege Library Integration for Server-Side JavaScript 446
Willem De Groef, Fabio Massacci, Frank Piessens

TrueClick: Automatically Distinguishing Trick Banners from Genuine Download Links 456
Sevtap Duman, Kaan Onarlioglu, Ali Osman Ulusoy, William Robertson, Engin Kirda

JShield: Towards Real-time and Vulnerability-based Detection of Polluted Drive-by Download Attacks 466
Yinzhi Cao, Xiang Pan, Yan Chen, Jianwei Zhuge

Message from the Conference Chair

In March 1985, a group of computer professionals gathered in McLean, Virginia, to discuss issues regarding the application of computer security technologies to aerospace programs. That first conference, a one-day affair, featured nine papers in two tracks, attracted about 115 people, and was acclaimed a success. The Applied Computer Security Associates (ACSA) formed to continue the meeting on a regular basis, and three decades and roughly 1700 technical presentations later, the Annual Computer Security Applications Conference (ACSAC) continues to fulfill its core mission to discuss issues related to the practical application of computer security technology.

Welcome to the thirtieth meeting of ACSAC! This year's meeting continues the strategic focus on cyber-physical system security began last year with even more peer-reviewed papers and expert panels. We also continue our discussion about the practical integration of security engineering with system engineering and building trustworthy, resilient systems.

ACSAC 30 presents the latest in cybersecurity research, hosts workshops in layered assurance, malware memory forensics, and program protection and reverse engineering, and provides training in the latest technologies and security standards. As a special feature, ACSAC will celebrate the golden anniversary of the Multics project with a distinguished panel of Multicians who will discuss Multics' influence on information security and examine how the world has — and has not — embraced Multics' core security principles.

ACSAC is also pleased to feature Dr. Aviel D. Rubin of Johns Hopkins University as the 2014 Invited Essayist. Dr. Rubin will speak on new directions in two-factor authentication to protect cyber applications. His paper, along with all of the peer-reviewed technical papers, is included in this volume.

A program this comprehensive results from the dedicated efforts of talented individuals. Special thanks go to the Organizing Committee, the Program Committee and the Steering Committee for their extraordinary contributions. Thanks also to our Sponsors for their faith in the mission of ACSAC.

We hope that the beauty and rich history of our host city of New Orleans has inspired you. We have certainly enjoyed spending time here. Next year, ACSAC will travel to a new city, and with your participation, it will continue to fulfill its mission for years to come. Thank you!

Sincerely,
Charles Payne
ACSAC 30 Conference Chair

Message from the Program Chairs

It is our pleasure to be the technical program chairs for the 30th Annual Computer Security Applications Conference. This is a special milestone, and the conference is celebrating this anniversary with an excellent program that represents thousands of hours of hard work. We are proud to continue to attract top quality work from academia, industry, and government, and the conference continues to become increasingly international, with papers submitted from North America, Europe, and Asia. It is due to the hard work of these authors, the members of the Program Committee tasked with reviewing and selecting the papers for inclusion to the program, and the members of the Organizing Committee who have worked diligently to ensure that the conference has an excellent venue for fostering discussion, that this conference is a success.

The authors of all of the submitted papers, accepted or not, deserve our sincere thanks for selecting ACSAC as the venue for sharing their cutting edge research. This year represents a record number of submissions, with 237 papers submitted and 47 accepted, for a 19.9% acceptance rate. ACSAC is one of the most competitive venues for systems security research, and the high quality of papers, including those that we were unfortunately unable to accept, demonstrates this. The papers cover topics from applied cryptography to deployable systems. This is the second year that the conference has had a "Hard Topics" focus on Cyber-Physical Systems, and we received some excellent submissions covering this important topic.

We are also extremely grateful to our Program Committee and sub-reviewers, who put in many hours of work to review the paper submissions. After the initial reviews, a substantial online discussion phase resulted in hundreds of comments and subsequent revisions to reviews. The conference and the community as a whole owes the committee a debt of thanks for the work that they put in to ensure that authors of all submitted papers received helpful and insightful feedback.

Finally, we thank the attendees of ACSAC. If this is your first time attending the conference then welcome, and if you are returning for another conference, welcome back! It is an honor to be part of ACSAC's 30th anniversary. Your continued support and submission of your strong work to ACSAC will ensure that the conference remains a vital forum for presenting and discussing the leading edge of applied computer security research for years to come.

Regards,
Kevin Butler and **Micah Sherr**
ACSAC 30 Program Chair and Program Co-Chair

Organizing Committee

Charles N. Payne, Jr., Adventium Labs (Conference Chair)
Kevin Butler, University of Florida (Program Chair)
Micah Sherr, Georgetown University (Program Co-Chair)
Adam Hahn, Washington State University (Proceedings Chair)
Larry Wagoner, NSA (Case Studies Chair)
Marshall Abrams, The MITRE Corporation (National Interest Track Chair)
Hassan Takabi, University of Pittsburgh (Panels Chair)
Raheem Beyah, Georgia Institute of Technology (Posters & Works-in-Progress Chair)
Daniel P. Faigin, The Aerospace Corporation (Professional Development Chair)
Harvey H. Rubinovitz, The MITRE Corporation (Workshops Chair)
Rance DeLong, LynuxWorks (Layered Assurance Workshop Chair)
Gabriela Ciocarlie, SRI International (Layered Assurance Workshop Deputy Chair)
Dan Thomsen, SIFT (Knowledge Coordinator)
Marshall Abrams, The MITRE Corporation (Local Arrangements Coordinator)
Art Friedman, NSA (Registration Coordinator)
Luanne Goldrich, Johns Hopkins Applied Physics Laboratory (Sponsorships Coordinator)
David Balenson, SRI International (Student Conferenceship Chair)
Robert H'obbes' Zakon, Zakon Group LLC (Web Advisor)
Jay Kahn, The MITRE Corporation (ACSA Communications Chair)
Carrie Gates, CA Labs (ACSA Government Liaison)

Steering Committee

Marshall Abrams, The MITRE Corporation
Jeremy Epstein, National Science Foundation and SRI International
Daniel P. Faigin, The Aerospace Corporation
Carrie Gates, CA Labs
Ann Marmor-Squires, The Sq Group
Ron Ross, National Institute of Standards and Technology
Christoph Schuba, Ericsson
Cristina Serban, AT&T Security Research Center
Dan Thomsen, SIFT
Robert H'obbes' Zakon, Zakon Group LLC

Program Committee

Kevin Butler, University of Florida (Program Chair)
Micah Sherr, Georgetown University (Program Co-Chair)

Adam Aviv, US Naval Academy
Lucas Ballard, Google
David Barrera, Carleton University
Rida Bazzi, Arizona State University
Matt Bishop, UC Davis
Srdjan Capkun, ETH Zurich
Justin Cappos, NYU
Henry Carter, Georgia Institute of Technology
Hao Chen, UC Davis
Gabriela Ciocarlie, SRI International
Jeremy Clark, Concordia University
Petros Efstathopoulos, Symantec Research Labs
William Enck, NC State
Ariel Feldman, University of Chicago
Matthew Fredrikson, University of Wisconsin
Ryan Gerdes, Utah State University
Steven J. Greenwald, Independent Consultant
Guofei Gu, Texas A&M
Ragib Hasan, University of Alabama Birmingham
Amir Houmansadr, University of Massachusetts, Amherst
Trent Jaeger, Penn State University
Rob Jansen, U.S. Naval Research Laboratory
Adam J. Lee, University of Pittsburgh
Brian Levine, University of Massachusetts, Amherst
Zhiqiang Lin, UT Dallas
Ulf Lindqvist, SRI International
Long Lu, Stony Brook University
Mohammad Mannan, Concordia University
Wojciech Mazurczyk, Warsaw University of Technology
Stephen McCamant, University of Minnesota
Damon McCoy, George Mason University
John McDermott, Naval Research Laboratory
Stephen McLaughlin, Narus Inc.
Thomas Moyer, MIT Lincoln Laboratory
Steve Myers, Indiana University
Daniela Oliveira, University of Florida
Xinming Ou, Kansas State University
Zachary Peterson, Cal Poly State University
Michalis Polychronakis, Columbia University
Georgios Portokalidis, Stevens Institute of Technology
Raj Rajagopalan, Honeywell
William Robertson, Northeastern University
Franziska Roesner, University of Washington
Ahmad-Reza Sadeghi, Technische Universität Darmstadt
Joshua Schiffman, Advanced Micro Devices, Inc.
Gaurav Shah, Google
Clay Shields, Georgetown University
Kapil Singh, IBM T.J. Watson Research Center
Robin Sommer, ICSI/LBNL
Abhinav Srivastava, AT&T Research
Cynthia Sturton, UNC Chapel Hill
Patrick Tague, CMU
Eugene Vasserman, Kansas State University
Hayawardh Vijaykumar, Samsung R&D, USA
Andrew West, Verisign Labs
Charles Wright, Portland State University
Dongyan Xu, Purdue University
Danfeng (Daphne) Yao, Virginia Tech
Heng Yin, Syracuse University
Wenchao Zhou, Georgetown University

External Reviewers

AbdelRahman M. Abdou, Carleton University
Abner Mendoza, Texas A&M University
Alexandru Bardas, Kansas State University
Andrew Henderson, Syracuse University
Andy Sayler, University of Colorado, Boulder
Apu Kapadia, University of Indiana, Bloomington
Arash Shahkar, Concordia University
Aravind Prakash, Syracuse University
Brad Reaves, Georgia Institute of Technology
Chao Yang, Texas A&M University
Cory Thoma, University of Pittsburgh
Dimitrios Damopoulos, Stevens Institute of Technology
Elias Athanasopoulos, FORTH-ICS

Erick Bauman, University of Texas at Dallas
Fengguo Wei, Kansas State University
Frank Capobianco, Penn State University
Giuseppe Petracca, Penn State University
Guangliang Yang, Texas A&M University
Haopei Wang, Texas A&M University
Jialong Zhang, Texas A&M University
John Sonchack, University of Pennsylvania
Kevin Hong, Texas A&M University
Lei Xu, Texas A&M University
Lianying Zhao, Concordia University
Mahesh V. Tripunitara, University of Waterloo
Martin Georgiev, UT Austin
Matthew Monaco, University of Colorado, Boulder
Ming Li, Utah State University
Mu Zhang, Syracuse University
Nicholas L. Farnan, University of Pittsburgh
Nirupama Talele, Penn State University
Qian Feng, Syracuse University
Raphael Reischuk, ETH Zürich
Rundong Zhou, Syracuse University
Sankardas Roy, Kansas State University
Sathya Chandran Sundaramurthy, Kansas State University
Sooel Son, UT Austin
Swarup Chandra, University of Texas at Dallas
Tam Chantem, Utah State University
William C. Garrison III, University of Pittsburgh
Xavier de Carné de Carnavalet, Concordia University
Xiaolong Wang, Kansas State University
Xinyang Ge, Penn State University
Xunchao Hu, Syracuse University
Yangchun Fu, University of Texas at Dallas
Yechen Qiao, University of Pittsburgh
Yue Duan, Syracuse University
Yufei Gu, University of Texas at Dallas
Yuping Li, Kansas State University
Yuqiong Sun, Penn State University
Zhaoyan Xu, Palo Alto Networks

Professional Development Course Proposal Reviewers

Daniel P. Faigin, The Aerospace Corporation (Professional Development Chair)
Colin Cole, The Aerospace Corporation
Ryan Eads, Sallie Mae
Stefanie Falvey, Lockheed Martin Corporation
John Mildner, SPAWAR Systems Center Atlantic
W. Warren Pearce, Retired
Harvey Rubinovitz, The MITRE Corporation
Michelle Ruppel, Saffire Systems
Cristina Serban, AT&T Security Research Center

ACSA Members

Marshall Abrams, The MITRE Corporation (Founder and Curmudgeon)
Matt Bishop, University of California, Davis
Jeremy Epstein, National Science Foundation and SRI International
Daniel P. Faigin, The Aerospace Corporation (Secretary)
Carrie Gates, CA Technologies
Ann Marmor-Squires, The Sq Group (Vice President)
Harvey Rubinovitz, The MITRE Corporation (Treasurer)
Cristina Serban, AT&T Security Research Center (President)
Robert H'obbes' Zakon, Zakon Group LLC
Mary Ellen Zurko, Cisco Systems

Message from the Sponsor

Applied Computer Security Associates

ACSA had its genesis in the first Aerospace Computer Security Applications Conference in 1985. That conference was a success and evolved into the Annual Computer Security Applications Conference (ACSAC). ACSA was incorporated in 1987 as a non-profit association of computer security professionals who have a common goal of improving the understanding, theory, and practice of computer security. ACSA continues to be the primary sponsor of the annual conference.

In 1989, ACSA began the **Distinguished Practitioner Series** at the annual conference. Each year, an outstanding computer security professional - or a panel, like this year - is invited to present a lecture of current topical interest to the security community. In 1991, ACSAC began the **Best Paper by a Student Award**, presented at the Annual conference. This award is intended to encourage active student participation in the conference. The award winning student author receives an honorarium and conference expenses. Additionally, our **Student Conferenceship** program assists selected students in attending the Conference by paying for the conference fee.

An annual prize for the Outstanding Paper has been established for the Annual Computer Security Applications Conference. The winning author receives a plaque and an honorarium. The award is based on both the written and oral presentations. ACSA initiated the **Marshall D. Abrams Invited Essay** in 2000 to stimulate development of provocative reading material for students of Information Security, thereby forming a set of Invited Essays. Each year's Invited Essayist addresses an important topic in Information Security not adequately covered by the existing literature.

The Classic Papers (or Books) feature begun in 2001. The classic papers are updates of some of the seminal works in the field of Information Security that reflect developments in the research community and industry since their original publication. Presentations consider how these classical security results will impact us in the years to come.

ACSA continues to be committed to serving the security community by finding additional approaches for encouraging and facilitating dialogue and technical expertise sharing. In previous years, ACSA has sponsored small **workshops** to explore various topics in Computer Security (in 2000, the Workshop on Innovations in Strong Access Control; in 2001, the Workshop on Information Security System Rating and Ranking; in 2002, the Workshop on Application of Engineering Principles to System Security Design). In 2003, ACSA became the sponsor of the already established New Security Paradigms Workshop (NSPW). In 2010 we welcomed the Layered Assurance Workshop (LAW) as an affiliated ACSA activity. In 2012, ACSA co-founded the Learning from Authoritative Security Experiment Results (LASER) workshop, which became an annual event.

After 2010's ACSAC tribute to Paul Karger and Bob Abbott, ACSA has initiated work on an **In Memoriam** section of our website to capture the many contributions made by our fellow practitioners; in 2013 we bid farewell to Willis Ware, and in 2014 to Yves Deswarte.

Building for the future, in 2011 ACSA started the Scholarships for **Women Studying Information Security (SWSIS)** program to encourage young women to join the field. We awarded $25,000 in scholarships in 2012 and 2013, and substantially increased this amount in 2014 by partnering with HP. Some of our winners are at ACSAC this year; please congratulate them and welcome them to our field!

ACSA is always interested in suggestions from interested professionals and computer security professional organizations on additional ways to achieve its objectives of encouraging and facilitating dialogue and technical interchange. For more information on ACSA and its activities, please visit http://www.acsac.org/acsa.

Contributing Sponsor

The National Science Foundation

Taking Two-Factor to the Next Level: Protecting Online Poker, Banking, Healthcare and Other Applications

Aviel D. Rubin
Johns Hopkins University
3400 N. Charles St.
Baltimore, MD 21208
+1-410-516-8177
rubin@jhu.edu

ABSTRACT
As an avid poker player, I enjoyed playing low stakes cash games and low buy-in tournaments on Full Tilt Poker before Black Friday.[i] However, as a Computer Scientist specializing in network and software security, I would never play poker online for any significant stakes, due to security concerns around malware and malicious remote access tools. Similarly, malware and remote access tools threaten online banking, online access to healthcare records and many other applications. In this article, I describe a new solution to the problem of remote access via malware that is easy to adopt, requires no new hardware or user training, and which I believe greatly reduces the threats to online applications.

Categories and Subject Descriptors
K.4.4 [**Security**]: Computers and Society; Electronic Commerce; Security

General Terms
Security

Keywords
Two-Factor; Security; Online Applications Security

1. THE PROBLEM OF MALWARE
Remote Access Tools allow an attacker to take complete control of a remote computer, much the way legitimate tools such as remote desktop connect, VNC, and Parallels Access do. There is no way to guarantee that a standard Windows, Mac, or Linux computer is uninfected. The latest attacks can bypass even the most advanced host-based mechanisms such as anti-virus tools. What's even scarier is that poker-specific malware has been identified in the wild, and there are documented cases of attackers using RATs to cheat at poker by remotely observing players' hole cards in high-stakes games.[ii] One of the best-known, poker-specific malware products is a Russian-based Trojan called i2Ninja, which collects information from a player's machine, such as their secret hole cards, and sends it to the attacker.[iii]

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.
ACSAC '14, December 08 - 12 2014, New Orleans, LA, USA Copyright 2014 ACM 978-1-4503-3005-3/14/12...$15.00
http://dx.doi.org/10.1145/2664243.2684461

When the stakes are high, even low-tech threats abound. Case in point: an online poker pro named Douglas Polk lost $35,000 when an attacker manually inserted malware on his computer that allowed for remote viewing of his hole cards.[iv] The attacker gained physical access to the computer when he stayed at Mr. Polk's house.

Would you play online poker if your opponent were sitting directly behind you and could see your computer screen, including your hole cards? If you play high stakes online poker, your opponent may indeed be able to see your poker hands without being detected.

2. TWO-FACTOR AUTHENTICATION
The security community, as well as the broader technology industry have widely recognized the shortcomings of password-based authentication. Two-factor authentication is now used in many real-world applications, and the banking industry has adopted this improved authentication technology broadly. Google and Apple now strongly encourage and in some cases require users to use two-factor authentication to access email and other applications that require security.

Two-factor relies on what a user knows as well as what the user has. The most common application of this is to send a text message with a code to a user's smartphone when the user is attempting to authenticate at a computer, and to require the user to enter the code at the computer. This ensures that the user is in possession of the smartphone that was registered when the user set up his account with two-factor authentication.

While two-factor represents a giant leap forward in security, the technology is still limited in its benefit. In particular, if a computer is infested with malware, the two factor authentication protocol will ensure that the user is able to authenticate, but the session that follows can still be compromised by the attacker, who after successful authentication can now piggyback on the resulting connection.

The goal of this paper is to build on the idea of two-factor authentication by utilizing two devices throughout an application session, such that if either one of the devices is compromised, the attacker will not succeed in attacking the application. We focus on poker as the motivating example and also describe applications to banking and healthcare.

3. THE RISKS OF TRADITIONAL ONLINE POKER
The risks to online poker fall into several categories. Some have to do with the integrity of the game. Is the house cheating? Are the cards randomly shuffled? Are any employees at the poker game site trying to cheat? There are documented cases of serious

problems with a corrupt house, the best known of which is the Absolute Poker scandal.[v] While these concerns are real, I believe that there is so much economic incentive for the large, legitimate corporations running poker sites to behave correctly, and very little for them to gain by cheating, that I think with proper audit and oversight, it is reasonable to trust the major industry players to run a poker site fairly.

Another real concern with online poker is that of collusion among poker players. In cash games, it is not too difficult for people to join the same poker table and to share information about their hole cards with each other over an out of band channel, such as over the phone. In online tournaments, it is more difficult for colluding players to sit at the same table, but when they do, they can cheat by dumping chips from one player to the other, potentially giving the receiving player a tremendous advantage in the tournament.

While collusion definitely happens, there are techniques, using statistical analysis, to detect and combat this type of cheating. Offending users can be banned from poker sites, and given that users are required to provide real-world identification and financial information to set up accounts, there is a limit to how many times such cheaters can join a site.

There are other security concerns in online poker including automated playing bots, denial of service attacks, and vulnerable user authentication. All of these have reasonable mitigations, and I believe that an online poker site that is run by skilled administrators can handle these.

The biggest open problem with online poker is the prevalence of malware such as Remote Access Tools (RAT). There is no security solution that protects against an attacker running RAT tools on a victim's machine.

4. AVOIDING MALWARE

Nobody has solved the problem of malware on users' computers, and no solution to that intractable problem is offered here. Instead, this paper suggests a new solution for playing online poker in a way that is secure, even if the user's computer is completely infected with malware.

The idea is to avoid the malware.

Here's the key. A player's secret hole cards are sensitive information. Everything else on a poker table should be publicly known. So the proposal is simple: Do not display the player's hole cards on the online poker table on the computer. In fact, do not even communicate the information about which cards the player is dealt to the user's computer. If the computer does not know the hole cards, then that information is safe from any attacker, even one running RAT tools on the machine.

Instead, communicate the user's hole cards to another device held by the user, such as a smartphone or tablet. That is the central idea. While it is common for computers to be infected with malware, and while it is increasingly the case that smartphones are vulnerable as well, it is significantly more difficult for an attacker to compromise both a user's computer and the same user's smartphone simultaneously in such a way that allows cheating at poker.

The user in this solution plays poker with the same online poker application and the same interface to which he is accustomed. The only difference is that instead of displaying the hole cards on the screen, the cards are not shown at all, and the user learns which cards were dealt to him by looking at a companion app that is running on a smartphone. Here is an illustration. Below is a poker table from PokerStars, and the user's hole cards are hidden. The user's smartphone displays the hand # and the user's hole cards. Whereas the user would see his hole cards in traditional online poker, in the proposed solution, the user's hole cards are not even available on the computer.

The players can perform all of the betting and other actions using their computer, as they've always done in the past. But they look at their smartphone to see what hand they are holding.

This solution can be added to existing online poker sites incrementally. Poker players can configure their account to use a smartphone for their hole cards, or to play the old fashioned way, and a poker table can consist of some players who are using smartphones for their hole cards and others who are just using their computers. Figure 1 illustrates the concept

4.1 Multi-tabling

The smartphone app described above supports multi-tabling, where a user plays on multiple poker tables at once. One way to do this is to have the poker client on the computer inform the smartphone about which table has the current *focus*. For example, say that a user has 6 simultaneous poker tables running, and that the displays for those tables are stacked. So, the smartphone app has 6 hands to display, and those hands are linked to the corresponding tables on the user's computer. When it is a user's turn to act at one of the tables, that table's image pops up to the front of the screen on top of all the other tables. At the same time, the computer can communicate, either through the game server, or even over a direct WiFi or Bluetooth connection, a table or hand identifier to the smartphone, which can then automatically display the corresponding hole cards. The user can also scroll manually through the active hands by swiping on the smartphone screen and rotating through the active hands.

One way to help the user link the hands on the smartphone to the tables on the computer is to use visual cues, such as matching screen skins. For example, each table can have a corresponding background color and outline design that matches an outline color and design on the corresponding hand on the smartphone.

Another way to display the hole cards on the smartphone is to show small images of all of the hole cards for all the active hands. Depending on the screen size and resolution and the number of active tables, this option may be viable. It works even better if the player uses a tablet instead of a smartphone. Here is an example of what a smartphone display can look like with multiple tables:

5. ONLINE BANKING APPLICATION

The main idea described in this paper is to protect a user's online activities from malware on his or her computer by splitting an application into two components. One of the components runs on a user's computer, and the other runs on a smartphone, over an independent network channel. This can be applied to online banking and many other online applications that involve sensitive information that could be compromised by malware on the user's machine.

An online banking application can be designed such that the user utilizes his computer and his smartphone such that if an attacker compromises either one of them, but not both, there is no risk to the user. For example, the application can be split so that the user logs into the account form the computer, and selects a bill payment. The split application then requires the user to use his smartphone to indicate the target of the bill payment, as well as the amount. Next, in this example, the user then completes the transaction on the computer.

The banking application described here differs from traditional two-factor authentication in that the application is split in an application-specific way. That is, the splitting of the application into two parts includes knowledge and understanding of what occurs inside of an online banking transaction. By contrast, traditional two-factor authentication typically involves sending a text message to a cell phone to authenticate a user, regardless of the specific application that requires the authentication.
An illustration of this process for a transfer transaction is shown in Figure 2.

Other applications that can be split in this manner include stock purchasing, access to sensitive medical records, online shopping, and any other application involving financial or otherwise sensitive information.

6. RETROFITTING A SOLUTION

This paper describes a way to split applications such as online poker and online banking such that portions of the application run on one device, such as a computer and other security-sensitive portions of the application run on another device, such as a smartphone. Let's take the case of online poker to examine how such a solution might be adopted.

One way to adopt this solution is for the online poker sites to re-implement their software such that users can register a second device, and the hole cards can be sent directly to that device. However, the online sites may decide that implementing such a solution is not worth it to them. In that case, a third party could implement a trusted proxy. The proxy would behave as though it were a poker client when interacting with the gaming server, and it would impersonate the gaming server to poker clients. In fact, the gaming site could implement the solution themselves as such a proxy to avoid rewriting their code.

The proxy could then handle the distribution of the hole cards to the smartphone, while preventing the sending of the hole cards to the user's computer. One challenge of this approach is that the proxy would most likely have to implement a man in the middle on an SSL connection. This is not a problem if the game server themselves wants to implement the proxy, but it would present a challenge for a third party proxy.

7. RESEARCH PROBLEMS

The concept of using two devices to perform application-specific splitting of an online service is relatively straightforward. However, actually implementing an application-specific split introduces several research problems.

For example, say that an online banking application is written in JAVA. Is there a way to describe, in some computer-readable way, the functionality that needs to be split, and to identify in a computer-understandable way, which items are security sensitive, and then to have a program automatically generate the code for the split application from the original program? Perhaps this could be achieved if the original program were annotated or designed in some way to allow for later splitting. So, the first question is whether there is a way to build applications such that it is easier to split them later. Furthermore, is there a way to automate the splitting of the code?

If the program can be split using source code, then a harder problem is to take the binaries for a program such as the server-side application and the client-side app and to use them to generate three new binaries with portions that run on the server, the client, and a new client, the user's smartphone.

Solving these research problems may be too difficult to achieve in the general sense, but it would be interesting to explore how far one could get with automated techniques for splitting client-server applications into client-client-server applications with the security properties described in this paper.

8. CONCLUSIONS

This paper addresses the problem of malware in online applications, and proposes a solution, which is an evolutionary step from the currently known two-factor authentication. We describe a simple solution to protect online poker hands, whereby a smartphone with an independent communication channel to the poker server is used to display the secret hole cards. This avoids the problem of malware on computers compromising the secrecy of user's cards and significantly raises the bar for an attacker. The solution is easy to implement and can be added incrementally to existing online poker servers. The technique can also be applied to online banking and other security-sensitive applications.

Figure 1: A user sees his hole cards on his smartphone, and the cards are hidden from the computer screen in the game of online Poker.

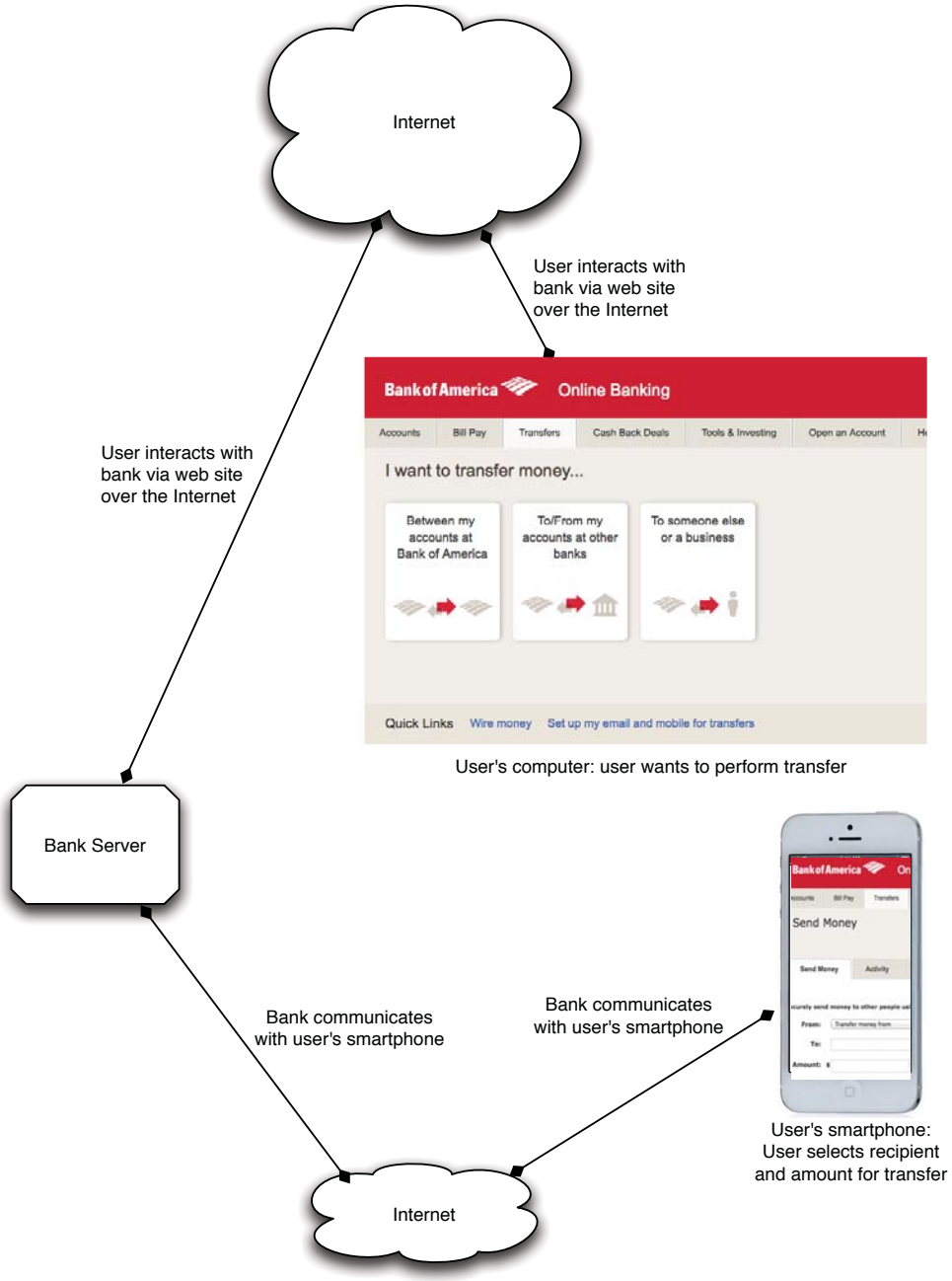


Figure 2: A user selects some options on the computer and then selects other options on his smartphone. In this example, the user selects the option to transfer funds on the computer and then picks a recipient and the amount on his smartphone.

---

[i] Friday, April 15, 2011, is commonly referred to as Black Friday in the poker community. On that date, the US Department of Justice issued indictments against the largest online poker operators, effectively shutting them down in the United States.

[ii] See http://arstechnica.com/security/2013/12/card-sharks-infect-professional-poker-players-laptop-with-a-dirty-rat/ for an eye-opening account.

[iii] See http://pokerfuse.com/news/media-and-software/new-financial-malware-targets-poker-players-with-pokergrabber-module-02-12/ and http://blog.trendmicro.com/online-poker-community-targeted-i2ninja-malware-module/.

[iv] http://www.onlinepokerreport.com/8040/wcgrider-refunded-for-hacking-implicates-suspect/

[v] See http://freakonomics.com/2007/10/17/the-absolute-poker-cheating-scandal-blown-wide-open/

Hot-Hardening: Getting More Out of Your Security Settings

Sebastian Biedermann Stefan Katzenbeisser
Security Engineering Group, Technische Universität Darmstadt
{biedermann, katzenbeisser}@seceng.informatik.tu-darmstadt.de

Jakub Szefer
Computer Architecture and Security Laboratory, Yale University
jakub.szefer@yale.edu

ABSTRACT

Applying optimized security settings to applications is a difficult and laborious task. Especially in cloud computing, where virtual servers with various pre-installed software packages are leased, selecting optimized security settings is very difficult. In particular, optimized security settings are not identical in every setup. They depend on characteristics of the setup, on the ways an application is used or on other applications running on the same system. Configuring optimized settings given these interdependencies is a complex and time-consuming task. In this work, we present an autonomous agent which improves security settings of applications which run in virtual servers. The agent retrieves custom-made security settings for a target application by investigating its specific setup, it tests and transparently changes settings via introspection techniques unbeknownst from the perspective of the virtual server. During setting selection, the application's operation is not disturbed nor any user interaction is needed. Since optimal settings can change over time or they can change depending on different tasks the application handles, the agent can continuously adapt settings as well as improve them periodically. We call this approach hot-hardening and present results of an implementation that can hot-harden popular networking applications such as Apache2 and OpenSSH.

1. INTRODUCTION

Configuration settings which are not well-chosen, not correctly applied or too weak can lead to security breaches and loss of sensitive information. For example, allowing users to set short passwords while also not limiting the amount of login attempts can make systems vulnerable to brute force password guessing attacks. Often, user selected security settings are lax and do not fit well into the appropriate scenario. On the other hand, strong and static security settings can be perceived as being cumbersome since they can drastically decrease usability. In the worst case, they can even cause users to become annoyed and motivated to disable or circumvent the settings. For example, recall the famous anecdote about users writing down complex passwords on a memo mounted on the computer display.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.
ACSAC '14, December 08 - 12 2014, New Orleans, LA, USA
Copyright 2014 ACM 978-1-4503-3005-3/14/12 $15.00
http://dx.doi.org/10.1145/2664243.2664246.

In this work, we analyzed configuration settings of 28 popular Linux networking applications. The average application has 43 different settings while 17 of these settings are related to security characteristics (40%). This means if a server runs three different networking applications, on average more than 50 security settings can be adapted to the setup and its specific properties.

In this context, we introduce the term "custom-made" settings. For example, the optimal limit for the maximal allowed length of requests to a Web server varies depending on the type of content a Web server provides and which kind of tasks it usually processes. Selecting the same limit for different Web servers will not be optimal even if these Web servers are the same application – the properties of the target specific setup in which a Web server runs need to be included in the selection process. By analyzing properties and characteristics of a Web server's setup an optimal setting can be derived which we call custom-made setting.

Moreover, depending on how a setup is changing over time, if software is installed or removed, if the number of users or the ways in which applications are used varies, custom-made security settings can change over time as well. Maintaining an overview in this medley of settings is a complex and time-consuming task in which failures can occur. For example, SSL/TLS misconfiguration attacks have been presented where if default setting values are not correctly set they will result in potential man-in-the-middle attacks due to peer hosts not being verified [1]. Obviously, an automated approach for continuous hardening of applications seems to be a promising solution.

In this work, we present an agent that can autonomously and transparently improve the security settings of applications that run in virtual servers. The agent generates custom-made settings for each application from its setup by analyzing the application's characteristics and properties, for example by investigating an application's network traffic or certain log files. The agent autonomously deploys these settings to the running application unbeknownst from the perspective of the virtual server and without the need to restart the application. The agent can periodically adapt the settings and dynamically change them depending on occurred events or depending on changes in the characteristics of the setup over time. Furthermore, the agent can also change the settings promptly depending on different tasks an application handles. This way, the agent can continu-

ously provide optimized settings for an application in its unique scenario. We call this approach "hot-hardening". In particular, we define hot-hardening as the *continuous and transparent adaptation of security-related configuration settings of an application depending on properties and characteristics of its setup*. Hot-hardening can be provided as a scalable cloud service in a cloud computing scenario.

The remainder of the paper is organized as follows. Section 2 presents related work. Section 3 shows the architecture of the proposed agent and the different techniques it uses. Section 4 explains the dynamic features of the agent in detail. Section 5 evaluates the agent by investigating different hot-hardening procedures, Section 6 discusses limitations and Section 7 concludes.

2. RELATED WORK

In this section we list related work in the fields of hot-patching and virtual machine introspection.

2.1 Hot-Patching of Applications

Hot-patching is an existing technique for transparently updating a target application's binary code, while our new technique hot-hardening focuses on security settings of applications and not the executable binary code. Hot-patching was developed in order to minimize application and server down-times, as it applies patches to an application during its run-time. In particular, this technique is of interest when administrators want to deploy regularly released security patches. Huang et al. [12] developed an autonomous hot patching framework which can automatically identify the cause of a failure and patch the binary code of Web-based applications. However, hot-patching can also be risky since the system can run into an inconsistent state. To solve the consistency issues, Ramaswamy et al. [16] proposed a method for hot-patching ELF binaries which supports synchronized global data. In their work, they use Patch Objects which are patches encoded as ELF relocatable object files and which can be automatically created and applied to a running process. Payer et al. [13] developed an update system that provides hot-patching by integrating dynamic patches with a sandbox based on dynamic binary translation. In a prototype implementation, they could patch 45 of 49 bugs on the Apache Web server at run-time.

2.2 Virtual Machine Introspection

Virtual machine introspection (VMI) allows to read and write the memory of a target running virtual machine (VM) from another privileged VM which is located on the same physical server. This way, transparent security tools can be developed [10, 15]. These tools are isolated from the operating system of the target VM which they monitor. Accordingly, these tools are tamper-resistant since they can not be manipulated or detected from the perspective of the target VM. VMI is the technique of timely and transparently applying forensic methods on a target operating system's memory with the difference that the system is still running and not terminated. Based on this technique, Baiardi et al. [2] proposed an intrusion detection tool which merges traditional host-based and network-based intrusion detection techniques with VMI. Fraser et al. [8] developed a security tool which automatically identifies and repairs infected kernels using various methods which include VMI. Payne et al. [14] proposed security tools which actively monitor hooks of VMs while these tools are isolated and tamper-resistant in another trusted VM. Benniger et al. [4] used light-weight introspection techniques based on para-virtualization in order to efficiently and transparently detect malware in commercial cloud computing environments. VMI can be used to build security tools that passively monitor target VMs, however, VMI can also be used to inject content into the operating system's memory of a VM and this way, for example, enforce the execution of code [11].

In order to successfully use VMI, information about the operation systems internals of the target VM is required. This is the so called "semantic gap" – certain static memory addresses depending on the used operating system need to be known to be able to extract useful information from a running VM. This introspection information is usually retrieved manually or assumed as being known. However, there are several approaches for automatically bridging this semantic gap. Dolan-Gavitt et al. [7] presented an approach to retrieve the introspection information by automatically analyzing dynamic traces of small programs running in the target VMs. Yangchun Fu et al. [9] identified the required introspection information using system wide instruction monitoring.

In our work, we use fine-grained VMI to transparently locate and modify security settings of applications which run in VMs. In particular, we focus on networking applications.

3. HOT-HARDENING ARCHITECTURE

In this section we describe our techniques used to transparently change settings of running applications, we introduce application-dependent hot-hardening templates used by our agent, and the ways in which settings are improved.

3.1 Hot-Hardening Procedure Overview

The agent runs in a separate agent-VM, which is a small VM equipped with our software. The agent is privileged and can access the hypervisor layer as well as all the memory addresses on the hardware component on which it runs. The agent runs co-located with multiple user-VMs and can periodically hot-harden the user-VMs and their applications with our proposed techniques. An simplified overview of a hot-hardening procedure can be seen in Figure 1.

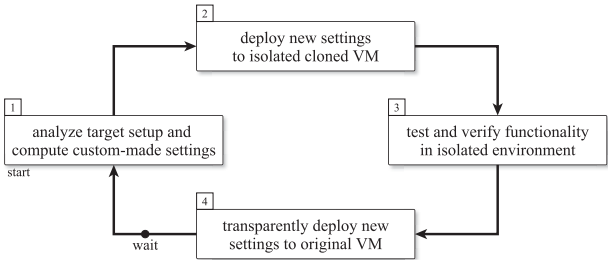


Figure 1: Simplified overview of the steps in a hot-hardening procedure performed by the agent.

In the first step, the agent discovers the operating system of the target VM and its running applications. For each operating system, there is a set of applications that the hot-hardening agent can recognize and hot-harden. For supported applications, the agent analyzes the properties and the status of the setup and computes custom-made settings

for that application. In the second step, the agent locates and deploys the new settings to a clone of the target VM in an isolated environment. This way, the settings can be tested and the execution of the original target VM is not disrupted. In the third step, the agent verifies the correct execution of the application as it runs in the cloned VM; and finally, in step four, it transparently deploys the new settings to the original target VM. Because custom-made settings can change over time, the agent repeats the hot-hardening procedure in certain intervals.

Since retrieving custom-made settings and testing the settings is a complex task requiring different strategies as well as semantic knowledge about the settings and their characteristics, we equip our agent with application-dependent hot-hardening templates – one template for every supported application. The agent can be equipped with multiple templates to support various applications. A template needs to be manually created and defines properties of an application and characteristics of its settings which are supported for hot-hardening. For each supported setting, the template can define two modules that are used by the agent: 1) to retrieve a custom-made setting from a setup (get-modules) and 2) to test a setting on a setup following a well-defined strategy (test-modules). The modules retrieve and analyze specific setting-dependent information from a setup.

3.2 Discovery and Hot-Hardening Templates

In first steps, the agent investigates which applications run on a target user-VM by transparently analyzing the VM's file system on the storage and open ports on the network layer. For discovery, the agent can use third party tools like Nmap[1] which offers a database of fingerprints of more than 2000 well-known services. The agent can also transparently scan the volatile memory of the VM and use a database of hashes of know applications to discover running applications. Once an application was discovered and the agent owns a hot-hardening template for this application, the agent initiates a hot-hardening procedure. Figure 2 shows an example template for an fictitious application called *mysrv*. It contains the name and the version of *mysrv* used to match a template to a discovered application. It defines the path to *mysrv's* configuration file located on the user-VM's storage which is required to retrieve *mysrv's* currently deployed settings. Not all settings of an application are suitable for hot-hardening. Supported settings are manually selected and defined in the template. Some applications may not be suitable for hot-hardening because they do not have any settings or they only use settings in their start-up phase and do not read them again during run-time or they do not store settings in the VM's volatile memory. For the example application *mysrv*, three different settings are supported:

- Setting s01 can be set in a range of $0 - 100$, stepwise changed by 10. The direction "low" is defined because the security characteristics of *mysrv* increase the lower s01 is set. There is a module *get01.py* available that derives a custom-made setting for s01 from a target setup. Furthermore, there is a module *test01.py* that can test the effect of s01 in a setup. In a real application, this setting could be a limit on the number of allowed requests.

[1] http://nmap.org

```
<template>
<name>mysrv</name>
<version>1.1b</version>
<config>/etc/example.cfg</config>
<setting name="s01">
        <range>0-100</range>
        <steps>10</steps>
        <direction>low</direction>
        <getModule>get01.py</getModule>
        <testModule>test01.py</testModule>
</setting>
<setting name="s02">
        <range>0-2</range>
        <steps>1</steps>
        <direction>low</direction>
        <testModule>test02.py</testModule>
</setting>
<setting name="s03">
        <range>1024-4096</range>
        <steps>128</steps>
        <direction>high</direction>
        <getModule>get03.py</getModule>
        <testModule>test03.py</testModule>
</setting>
<instruction>s01|0|s03|*|s02</instruction>
<hothard-steps>1,2,3</hothard-steps>
<location>heap</location>
</template>
```

Figure 2: Hot-hardening template of an application.

- Setting s02 can only be set in a range of $0 - 2$, stepwise changed by 1 where lower settings are beneficial for security of *mysrv*. There is no module available to derive a custom-made setting for s02 which means the setting is independent from a setup. However its effect can be tested with the module *test02.py*. In a real application, this setting could be a string, for example defining a log-level (high, medium, low), but stored in volatile memory as an integer.
- Setting s03 has a possible range of $1024 - 4096$ while it can be stepwise changed by 128 and higher settings are beneficial for security of *mysrv*. There is a module *get03.py* that computes a custom-made setting for s03 and a module *test03.py* that can test its effect. In a real application, this setting could be the length of a cryptographic key.

In a hot-hardening procedure, the agent does not just set the best possible (highest or lowest) setting. Instead, the agent uses the get-modules to find the best setting for the specific target setup in the allowed ranges defined by the template. Additionally, the template defines an instruction indicating how to generate a pattern which is used to locate the data structure that stores the currently deployed settings in *mysrv's* volatile memory. In this context, the order of the settings in memory as well as the allowed content in-between them needs to be defined. In order to define this instruction, manual investigations of the application's source code is required. Furthermore, the template defines a memory address region in *mysrv's* memory in which the data structure of the settings is stored (heap) to accelerate the localization procedure and the different steps required in *mysrv's* hot-hardening procedure. These steps are explained in Section 3.4.

3.3 Isolated Setting Testing

Changing settings via VMI during an application's run-time can lead to interruptions in the application's operation, unexpected results and to misleading entries in log files. In order to avoid these problems on the user-VM, the agent clones the user-VM, deploys and tests settings on the clone. Once better working settings could be found, the agent deploys them on the original user-VM. Live cloning [17, 5] is a modified live migration [6] process which creates a synchronous replica of a target VM during its run-time. A new VM-container is created and the VM's memory is copied while memory pages which were modified during the copy process are again copied in multiple iterations until a certain synchronization is reached. The VM's storage is not copied, instead a copy-on-write snapshot is used (Figure 3).

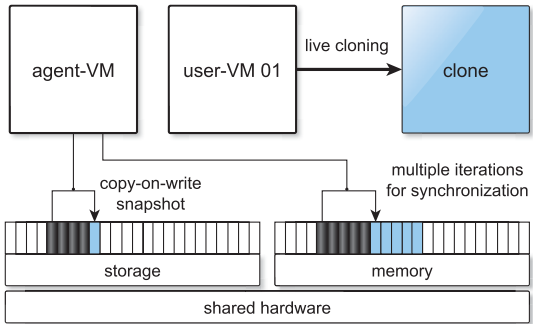


Figure 3: The live cloning process triggered by the agent to test new settings on the clone.

If the agent changes multiple different settings or settings of different applications on the same user-VM, dependencies of settings can play a role. For example, application A can operate together with application B and changing a setting of application A can increase their mutual workload while changing a setting of application B decrease the amount of workload application B can handle. In order to test new settings and to detect potential unknown dependencies or any other problems, the agent evaluates the effect of each new setting by executing a setting dependent test-module. Each test-module which tests the characteristics of a setting i returns a numerical result R_i that can deviate from the known theoretically expected result R_i^e depending on the currently deployed setting. If multiple settings are changed, the agent first deploys the best settings as defined in the template and tests the effects on the setup. If the results of the test of a new setting deviates from its expected effect ($R_i^e - R_i > 0$), the agent stepwise reduces the setting according to the defined steps in the template. In multiple rounds, the agent evaluates the tests of all changed settings and tries to find the best group of settings. This way, the agent can reveal if changing a setting influences the effect of another setting.

It may happen that an application terminates during the hot-hardening procedure. In this case, the agent creates a new clone. Once the best working group of settings could be found, the agent terminates the clone and finally deploys the new tested settings on the original user-VM.

3.4 Transparent Setting Deployment Steps

Transparently changing settings on a running application is facilitated by the underlying virtualization technologies that allow the agent to use different transparent techniques on a user-VM. First, the agent can use VMI to read and modify a running application's volatile memory. Second, the agent can use forensic methods on the running VM's raw storage and this way retrieve information about stored non-volatile data like files. Settings are automatically replaced by the agent unbeknownst from the perspective of the user-VM and without the need to interrupt an application's operation. In particular, the agent deploys new settings by executing up to three different steps which are illustrated in Figure 4:

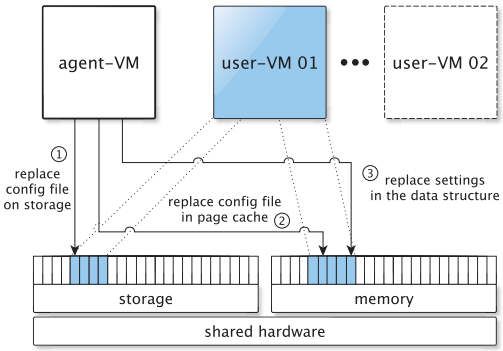


Figure 4: The agent can use three different steps to transparently deploy new settings to a running application on a user-VM.

1. The agent locates the configuration file of the application by investigating the file system's meta data on the user-VM's raw storage. Once the corresponding sectors are located, the sectors are overwritten and the content is replaced with new content in the same structure containing the new settings. Replacing the configuration file on the storage prevents reuse of old settings once the application is restarted or if it reloads settings from its configuration file during run-time.

2. The agent tries to locate and replace a cached copy of the application's configuration file in the operating system's page cache with the new content. The page cache helps applications to retrieve stored data faster by keeping recently requested data in memory. Replacing potentially cached content of the configuration file prevents reuse of old settings.

3. The agent locates and replaces the currently used configuration settings in the application's volatile memory. To reach this goal, data structure alignment and endianness of the settings need to be considered. There are different ways in which an application can process a configuration file and store its settings in memory, depending on the programmed data structure and on the used data types. Often, settings are summarized and stored in a struct data type, sometimes settings are stored independently. However, sometimes settings are not stored in memory and read from the file each time they are required. In order to locate the settings in memory, the agent uses the $< instruction >$ block from the template to generate a pattern and the target memory region defined in the template.

Sometimes, depending on the target application, only steps 1 and 2 are required, because the application does not store settings in memory. Sometimes only step 1 is required, because there is no page cache as well as there are no settings in memory. Sometimes only step 3 is required, because there is no configuration file but there are settings in memory fetched from other sources such as system variables or command line arguments. The template defines which steps are required for the application.

4. DYNAMIC CHILD ADAPTATION (DCA)

Up to this stage, the agent can transparently change settings, test the new settings in an isolated environment (cloned user-VM) and finally deploy them on a user-VM. In order to define the semantics of settings and allow the agent to work autonomously, we introduced application-dependent templates. Furthermore, the proposed agent uses setting-depending modules to retrieve a custom-made setting by analyzing a target setup and to test the effects of a new setting. According to Figure 1 the agent can periodically repeat this hot-hardening procedure on the same user-VM and this way adapt settings which may become suboptimal over time since the setup or the scenario can change.

However, sometimes it is beneficial to allow the agent to dynamically change settings depending on events. In this context, an event can be a new connection or a new task the application handles. Often, an application creates temporary child processes to handle different tasks separately, for example it creates a new child process per new network connection. Sometimes, an application also assigns the data structure of its settings to the child processes or the workflow of the child processes can be adapted by temporary changing the stored settings of the main process for a very short-time during child process creation. This allows the agent to modify the settings of new child processes separately in the volatile memory. This way, the agent can deploy different settings for each new child process depending on certain events or properties.

We call this fine-grained and dynamic hot-hardening procedure Dynamic Child Adaptation (DCA). In particular, a property of a child process, for example of the connection it processes, can lead to an adaptation of the child's settings. If an application has been hot-hardened once, the agent can continuously monitor the user-VM and adapt certain settings of the corresponding child processes. As an illustrative example, we extend the hot-hardening template of *mysrv* which now additionally defines for which settings DCA is supported and which properties lead to which adaptations (c.f. `<dca>` tags in Figure 5). The template allows the agent to use DCA for the two settings s01 and s03. Once a child process handles a connection from the own system, setting s01 is set to 100 for the child. Once a child process handles a connection from the same sub net, setting s03 is set to 1024 for the child. This way, security settings can be relaxed under certain circumstances in order to increase usability or security characteristics can be dynamically amplified. Furthermore, in this case, the settings are directly deployed on the user-VM without creating and testing them on a clone first. However, DCA is only supported after the application has already been hot-hardened, which means changing settings was already tested. The DCA procedure which the agent performs to adapt child processes is illustrated in Figure 6 in more detail. First, a new incoming connection is detected and analyzed by the agent before it reaches the user-VM. The agent retrieves the source IP address and other information about the new connection. This is possible since communication is tunneled through the virtualized bridge of the hypervisor. If the properties of the new connection match a DCA event definition in the extended template, the connection is delayed until the settings of the application are adapted. After adaptation, the connection is allowed to proceed and the application now creates a specially adapted child process. Once the child was created, the agent immediately reverts the settings in the parent application back to the custom-made settings which are effective for other clients.

```
<template>
<name>mysrv</name>
<version>1.1b</version>
<config>/etc/example.cfg</config>
<setting name="s01">
        <range>0-100</range>
        <steps>10</steps>
        <direction>low</direction>
        <getModule>get01.py</getModule>
        <testModule>test01.py</testModule>
        <dca>
                <event><src>127.0.0.1</src></event>
                <set>100</set>
        </dca>
</setting>
...
<setting name="s03">
        <range>1024-4096</range>
        <steps>128</steps>
        <direction>high</direction>
        <getModule>get03.py</getModule>
        <testModule>test03.py</testModule>
        <dca>
                <event><src>192.168.0.0/24</src></event>
                <set>1024</set>
        </dca>
</setting>
...
```

Figure 5: Extensions to *mysrv's* hot-hardening template in order to support DCA for s01 and s03.

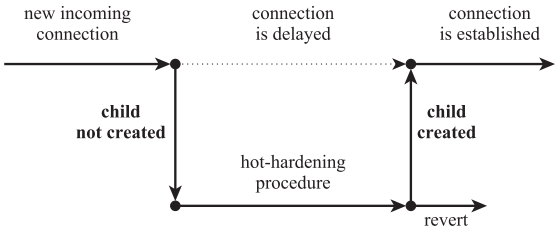


Figure 6: Steps in the Dynamic Child Adaptation (DCA) hot-hardening procedure in detail.

5. EVALUATION

In the evaluation, we investigate hot-hardening applied to popular applications and analyze the results in detail. We use a Xen [3] server with a Quad CPU (2.67 GHz) and 8 GB memory and a user-VM with 4 GB memory. Both run Ubuntu Linux 12.04 64-Bit.

5.1 Example: Apache2

First, we investigate a hot-hardening procedure for the Apache2 Web server 2.2. Our Apache2 Web server is a production system hosting our Web page and giving access to a regularly used SVN repository. Without considering any Apache2 extensions, we identified 7 settings of the application which are related to security and which are suitable for hot-hardening. Apache2's configuration file is located in */etc/apache2/apache2.conf*. The settings are stored in Apache2's volatile memory as a *struct* data type called *server_rec* defined in the source code in *include/httpd.h*:

```
struct server_rec {
    ...
    /** Timeout, before we give up */
    apr_interval_time_t timeout;
    /** Interval we will wait for another request */
    apr_interval_time_t keep_alive_timeout;
    /** Maximum requests per connection */
    int keep_alive_max;
    ...
};
```

By default, Apache2 handles new connections in child processes which maintain an own struct of the settings. This means that DCA is applicable.

5.1.1 Apache2's Hot-Hardening Template

A fragment of the hot-hardening template for Apache2 can be seen in Figure 7. Seven different settings are supported. Their effects mitigate Brute-Force attacks, prevent malicious tools scanning for vulnerabilities and misconfiguration and avoid Denial-of-Service (DoS) attacks and the execution of exploits (for example based on large or unusual input). The supported settings are briefly explained in the following:

- s01 defines the log level $(0-7)$ and is originally a string (emerg, alert, crit, ..., info, debug). The template limits the setting to seven levels $0-6$ (no debug mode) which can be changed stepwise by 1 while 6 logs the most (info). Logging more details increases Apache2's security characteristics.
- s02 defines the time Apache2 waits to receive a request from a new connection before terminating it. The template defines a range of $10-300$ seconds for s02 which can be stepwise changed by 10. A low setting increases the security characteristics of Apache2.
- s03 defines the maximum number of requests allowed during a persistent connection before Apache2 terminates it. The template defines a range of $10-100$ for s03 that can be stepwise changed by 2, the lower it is set the better the security characteristics are.
- s04 defines the time for which a connection is maintained for a next request. The template defines a range of $2-30$ seconds, a lower value is beneficial for security.
- s05 defines a limit for the size of HTTP request lines in bytes. The template defines a range of $1024-8192$ while low is assumed as being better.
- s06 limits the size of request header fields in bytes and ranges between $1024-8192$.
- s07 limits the number of header fields in requests. It can be set to a range of $10-100$.

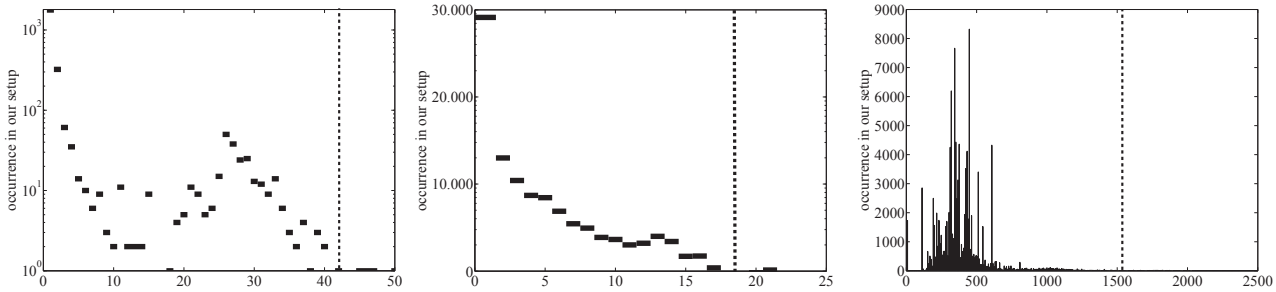
```
<template>
...
<setting name="s01">
        <range>0-6</range>
        <steps>1</steps>
        <direction>high</direction>
        <testModule>test01.py</testModule>
        <dca>
                <event><src>192.168.0.0/24</src></event>
                <set>1</set>
        </dca>
</setting>
...
<setting name="s06">
        <range>1024-8192</range>
        <steps>512</steps>
        <direction>low</direction>
        <getModule>get06.py</getModule>
        <testModule>test06.py</testModule>
        <dca>
                <event><src>192.168.0.0/24</src></event>
                <set>8192</set>
        </dca>
</setting>
...
<instruction>
s01|*|s02|0|s04|0|s03|.|s05|0|s06|0|s07</instruction>
<hothard-steps>1,2,3</hothard-steps>
<location>anonymous</location>
</template>
```

Figure 7: The hot-hardening template of Apache2.

For s01 no get-module is available since this setting is assumed as being independent from a setup. For the settings s02, ..., s07, get-modules are available which can retrieve a custom-made setting from a setup (see Section 5.1.2). The effect of each setting can be tested with a setting-dependent test. The template also defines an instruction how to generate a pattern in order to locate the currently deployed settings (derived from the configuration file) in Apache2's volatile memory. In this context, the order of the stored settings and the content in-between them (for example other settings) is given. This information was derived from the source code. The template defines which hot-hardening steps are required (Section 3.4) and the memory region in which the data structure of the settings is stored. In this case, it is stored in anonymous mappings created by mmap() with the *MAP_ANONYMOUS* flag. Analysis of the source code showed that a child process maintains the data structure. Accordingly, the template allows dynamic child adaptation (DCA), but restricted to the settings s01 and s06. Setting s01 is dynamically set to 1 which means less events need to be logged if a client connects from our sub network. Setting s06 is set to the maximum of 8192 for clients connecting from our private sub network which allows them to perform longer requests with more headers.

5.1.2 Apache2's Hot-Hardening Modules

The template specifies six get-modules to retrieve custom-made settings for s2, ..., s7 from our setup. The computed settings depend on various properties, for example the workload of our system, other installed applications or the kind of data our Web server provides. Our Apache2 server hosts



(a) Requests per connection. (b) Time between requests [s]. (c) Length of requests [bytes].

Figure 8: Distributions of different events extracted from our setup by the get-modules which use this data to derive a custom-made setting for s03, s04 and s05 (marked as dotted line).

a Web page and a repository (SVN) that can be remotely used by employees who mostly connect from within our private network. This is a special property of our setup which actually leads to a frequent occurrence of multiple GET or PUT requests in a row caused by users rapidly synchronizing SVN projects consisting of multiple files. The get-modules compute an average value of events occurring on the setup which are affected by a setting or analyze the distribution of events based on analysis of log files or network traffic of the application. The get-modules select the next best possible value as custom-made setting according to the definitions in the template.

- get02.py analyzes network traffic and the timings of new connections and their first request. In our setup, the mean is 2.89±4.01 seconds which leads to a custom-made setting of 10 seconds (default is 300).
- get03.py analyzes network traffic and investigates the amount of requests performed per connection (Figure 8a). In our setup, there is a second peak in the figure caused by users synchronizing multiple files with our SVN repository. This is a special property of our setup and leads to a custom-made setting for the limit of requests per connection of 42 (default is 100).
- get04.py analyzes timings between multiple request on a persistent connection (Figure 8b). It calculates a custom-made setting of 18 seconds for our setup (default is 30).
- get05.py analyzes the length of requests retrieved from the Apache2 access log file (Figure 8c). It calculates a custom-made setting of 1536 bytes for our setup (default is 8190).
- get06.py analyzes the length of the headers in requests from the Apache2 log file. It calculates a custom-made setting of 1024 bytes for our setup (default is 8190).
- get07.py analyzes the numbers of fields used in headers in requests. It calculates a custom-made setting of 30 (default is 100).

The get-modules consider the special properties of our setup and derive custom-made settings for Apache2 which increase security while minimally affecting the work-flow. The test-modules actively perform tests on the setup. Each test-module analyzes the effect of a currently deployed setting. For example, they establish new connections to Apache2 to verify expected timeouts, limits or other characteristics caused by the settings. In this context, the test-module for s01 analyzes the information that is logged while it actively performs requests, other test-modules analyze the timeouts for s02 and s04 while maintaining a connection and the module for s03 tests how many requests are possible on a persistent connection. The modules which test s05, s06 and s07 perform different kinds of requests (length and header fields) to evaluate the corresponding setting.

5.1.3 Apache2's Hot-Hardening Procedure

In this section, we investigate the hot-hardening procedure in detail after the custom-made settings have been computed. First, the agent clones the user-VM before applying and testing any new settings. In our setup, live cloning of a running user-VM with 4 GB of memory into a synchronous replica can be performed in 17 ± 2 seconds on average. This timing strongly depends on the current workload of the VM. In order to reduce the time required for cloning, prepared clones can be maintained or they can be created using copy-on-write (CoW) methods for both memory and storage which enables the creation of a clone in the range of milliseconds [17]. In the first step, the agent identifies the currently deployed settings by locating and reading the corresponding sectors of Apache2's configuration file on the clone's raw storage. In general, struct members are stored in the order they are declared (C99 standard) and if necessary, padding is added before each struct member to ensure correct alignment. Setting s02 and s04 are stored as long data types representing microseconds. The other settings are stored as integer data types. Furthermore, the order of the byte representation of s02, s04, s05 and s06 has to be adapted according to Little-Endianness. With the help of the current settings and the instruction defined in the template, the agent generates a pattern to locate the deployed settings in Apache2's memory. The generated characteristic pattern P in hexadecimal representation is

```
P = 04(.)*?00a3e111(0)*?80c3c901 ... fe1f(0)*?64
```

The pattern allows padding with zeros between successive settings and arbitrary content between settings which have other settings in-between.

Retrieving custom-made settings from our setup using the seven different get-modules takes 121 ± 4 seconds based on

an anonymized network capture of 62 MB and a log file of 24 MB. Finding the corresponding sectors of the configuration file on the storage and replacing the content with new content takes only 0.3 ± 0.1 seconds. Locating and replacing the content of the cached configuration file in the Linux page cache takes 5.2 ± 0.9 seconds. Generating the pattern according to the instructions, locating the data structure of the settings in Apache2's memory and replacing the current settings with new settings takes 2.4 ± 0.1 seconds. Once new settings are deployed, the agent executes the setting dependent tests and analyzed the results in order to reveal unexpected deviations. Evaluating the seven tests takes 46 ± 8 seconds. Finally, a full hot-hardening procedure for Apache2 takes around 3 minutes in our setup without any user interaction. This is a good result considering that the agent performs deep analysis on network traffic and log files and that the agent needs to wait during executing test-modules in order to realistically verify timeouts.

For DCA, no configuration files need to be replaced. The settings s01 and s03 are dynamically hot-hardened. Hot-hardening of a child once the triggering properties of a new client could be detected takes only 2.1 ± 0.1 seconds.

5.1.4 Extension: Apache2 and PHP5

Apache2 usually runs in combination with several extensions. In this section, we introduce a template for PHP5 hot-hardening and let the agent hot-harden Apache2 and PHP5 on the same setup. Interesting security-related settings of PHP5 are stored in a struct data type called *php_core_globals* defined in the source code in the file *php_globals.h*:

```
struct _php_core_globals {
    ...
    long memory_limit;
    long max_input_time;
    ...
};
```

The template for PHP5 (no Figure shown) provides five different settings which are related to security and supported by the agent:

- s01 defines the maximum amount of memory (MB) a PHP script can consume (default 128MB).
- s02 defines the maximum time a PHP script can spend parsing a request (default 60 seconds).
- s03 defines the maximum size (MB) for uploads (default 2MB).
- s04 defines the maximum level of nested requests, for example of an array[][][] (default 64).
- s05 defines the maximum number of GET and POST input variables (default 1000).

The get-module of s02 analyses the timings between requests for PHP scripts and the resulting replies sent from the server. The get-module of s03 analyzes the Apache2 log for uploads and their sizes. The get-modules of s04 and s05 analyze the PHP code found in Apache2's public directory (default */var/www/*) for definitions of nested variables and their depth and the amount of variables accepted by scripts.

The agent retrieves the currently deployed settings from the configuration file */etc/php5/apache2/php.ini* and generates a pattern according to the instruction given in the PHP5 template while taking into account data alignment and endianness. The settings are stored in the anonymously mapped memory region of libphp5.so used by Apache2. The agent uses the get-modules to derive new custom-made settings from our setup. The agent uses the test-modules to evaluate the new settings, in this case also by creating and inserting PHP code on the clone. To reach this goal, the agent creates specific PHP files in the public directory of Apache2 and triggers the execution in order to evaluate the PHP characteristics. Execution of the modules could be performed in 104 ± 3 seconds and insertion of new custom-made settings to PHP5 running in the clone while also replacing the stored configuration file and the file in the page cache could be performed in 8 ± 1 seconds.

In the first step, Apache2 is hot-hardened and the setting dependent tests are evaluated for both Apache2 and PHP5 according to the agent's testing strategy defined in Section 3.3. In the second step, the agent hot-hardens PHP5 and again evaluates the tests for both applications. Since the agent evaluates two templates, dependencies can come into play. For example, during Apache2's hot-hardening, the evaluation of PHP5 test03.py detected a high deviation between the expected characteristics and the actually measured characteristics on our setup. PHP5 test03.py evaluates the maximum allowed size for uploads according to the PHP5 setting s03 which defines 2 MB in our setup before hot-hardening. After the agent changed Apache2 setting s02 which defines the timeout for requests to the new custom-made setting of 10 seconds, the PHP5 test03.py could not upload a file of size 2 MB since the new timeout was too low for the upload. According to the proposed strategy, the agent automatically revoked Apache2 setting *s*02 and deployed the next possible step defined in Apache2's template, which is 20 seconds. However, identifying a dependency between Apache2 setting *s*02 and PHP5 setting s03 caused two additional testing rounds and which led to a hot-hardening time for Apache2 of 8 minutes instead of 3 minutes. Dependencies like this can occur when applications share resources or somehow work together like in this scenario. However, with the help of the agent's strategy and well-defined templates, dependencies can be revealed and settings can be successfully adapted. Finally, hot-hardening of Apache2 and PHP5 takes around 11 minutes in our setup.

5.2 Example: OpenSSH2

Finally, we investigate hot-hardening of OpenSSH2 in detail. Our OpenSSH2 server is a production system having several users who log in regularly, also using SCP and sFTP.

5.2.1 OpenSSH2's Hot-Hardening Template

The settings of OpenSSH2 are stored in a struct data type called *ServerOptions* defined in *servconf.h*. The template for OpenSSH2 can be partially seen in Figure 9 and supports 8 different security settings. The supported settings are briefly explained in the following:

- s01: Size of the key (protocol 1, default 768 bits).
- s02: Time after which the key is regenerated (protocol 1) in seconds (default 3600 seconds).
- s03: Different ciphers which are allowed (protocol 2) (e.g. aes128-cbc, blowfish-cbc, ...). The template defines three groups classified by strength.

```
<template>
...
<setting name="s04">
        <range>5-120</range>
        <steps>5</steps>
        <direction>low</direction>
        <getModule>get04.py</getModule>
        <testModule>test04.py</testModule>
</setting>
<setting name="s05">
        <range>1-6</range>
        <steps>1</steps>
        <direction>low</direction>
        <getModule>get05.py</getModule>
        <testModule>test05.py</testModule>
</setting>
<setting name="s06">
        <range>0-1</range>
        <steps>1</steps>
        <direction>low</direction>
        <getModule>get06.py</getModule>
        <dca>
                <event><src>192.168.1.7</src></event>
                <set>1</set>
        </dca>
</setting>
...
</template>
```

Figure 9: The template of OpenSSH2.

- s04: The server disconnects after this defined time if the client has not logged in (default 120 seconds).
- s05: Maximum number of authentication attempts allowed before the connection is terminated (default 6).
- s06: Specifies if host-based authentication (together with public key) is allowed (default *no*).
- s07: Specifies if forwarding of X11 is allowed (default *no*).
- s08: Specifies if root login is allowed (default *no*).

The template defines DCA for the setting s06, s07 and s08. Host-based authentication is only enabled for the back-up server (192.168.1.7) that automatically transfers and stores new data daily. This is done by DCA. Accordingly, no other client is allowed to use host-based authentication. Forwarding of X11 is allowed only for clients connecting from our private network and remote root login only allowed for one host which is used by the administrator. The OpenSSH2 server is dynamically hot-hardened which allows different settings under different circumstances which normally could only be statically deployed for all clients.

5.2.2 OpenSSH2's Hot-Hardening Modules

The get-modules and their results in our specific setup are briefly explain in this section:

- get01.py investigates if protocol 1 is supported and which key size is used by default. It then suggests to use highest setting (2048).
- get02.py and get03.py investigate the status of the setup and propose 300 for s02 and 3 for s03, which means only the strongest cipher algorithms are supported.
- get04.py investigates a network capture and the authentication log file and analyzes the timings between the establishment of a new connection and a successful login. In our setup 7.31 ± 2.76 which leads to a suggested custom-made setting of 10.
- get05.py analyzes the login tries users need to successfully log-in. In our setup 1.11 ± 0.39 which leads to a custom-made value of 2.

The get-modules get06.py, get07.py and get08.py investigate the status and suggest the best setting if feasible.

5.2.3 OpenSSH2's Hot-Hardening Procedure

In our test-runs, the agent clones the user-VM in 16 ± 4 seconds and evaluates the get-modules to retrieves custom-made settings in 47 ± 5 seconds Finding the sectors of the OpenSSH2 configuration file and replacing the content with new content takes 0.2 ± 0.3 seconds. Locating and replacing the file's content in the Linux page cache takes 5.4 ± 0.8 seconds. Generating the pattern and locating the deployed settings in OpenSSH2's volatile memory and replacing the current settings with new settings takes 1.9 ± 0.2 seconds. The testing phase takes only 31.2 ± 2.2 seconds. Finally, the hot-hardening procedure of OpenSSH2 could be automatically performed by the agent in around 1 minute and 42 seconds.

After the main hot-hardening procedure is finished, DCA is enabled, which means the agent continuously monitors the user-VM and dynamically adapts the settings s06, s07 and s08 for new child processes depending on properties of the IP address of a new client. DCA of OpenSSH2 can be performed by the agent in 2.1 ± 0.2 seconds which means a new TCP connection to the SSH port for which DCA is allowed needs only to be delayed for that period of time until it can proceed. Other connections for which DCA is not enabled are only delayed in the range of milliseconds. In our scenario, DCA enables specific settings for our backup server which now can log-in via host-based authentication while this option is disabled for all other clients. In the same manner, X11 forwarding and remote root login is only allowed for specific clients while maintaining stronger security settings for the others. DCA allows to define the utilization of standard settings on a more fine-grained way and can greatly increase security as well as usability.

6. LIMITATIONS

The agent can not hot-harden every application. Some applications do not store configuration settings in memory or they do not provide any configuration settings. Only applications which store settings in memory and which also regularly read and use these settings during their operation are suitable. Some applications only adapt their operation during the start and do not read the settings again, they use external resources or there is a risk of creating race conditions. Additionally, not all security settings of an application are suitable for hot-hardening – they need to be well-chosen and defined in the template. Furthermore, there is no guarantee that unknown dependencies of different applications

can be detected in every scenario. The templates need to be well-defined and the agent should only hot-harden settings which effects can be reliably estimated. For DCA, it is only applicable if the application assigns separate settings to the child processes or adapts their operation based on settings.

7. CONCLUSION

Applying optimized security settings for different applications is a difficult and time-consuming task. In this work, we presented an autonomous agent that can periodically improve the security settings of different applications running on virtual servers which we called hot-hardening. The proposed agent only uses transparent techniques based on virtualization technologies in order to update configuration files and data structures that store settings in an application's memory. During setting selection and improvement, the operation of an application is not disturbed as well as no user interaction is required. The agent can support applications for which a template is defined and for which setting-dependent modules are available. Modifying the source code of an application is not required. The agent uses setting-dependent modules to retrieve custom-made settings from a specific setup and setting-dependent tests in a testing strategy to verify correct functionality. Since optimal settings can change over time the agent can periodically analyze the characteristics of a setup and periodically adapt the security settings. In order to support different setting for different tasks, we introduced Dynamic Child Adaptation (DCA) which enables the agent to rapidly deploy specific settings for new child processes depending on properties of the task the child handles. In an evaluation, we showed that the agent can hot-harden three different popular applications (Apache2, PHP5 and OpenSSH2) in feasible time and automatically improve their security characteristics. Code related to this publication will be made available online at `http://caslab.eng.yale.edu/code`.

Acknowledgments

The work presented in this paper was performed in the context of the Software-Cluster project SINNODIUM (`www.software-cluster.org`). It was funded by the German Federal Ministry of Education and Research (BMBF) under grant no. "01C12S01V".

8. REFERENCES

[1] #PeerJacking - SSL Ecosystem Attacks Against Online Commerce. `http://www.unrest.ca/peerjacking`.

[2] F. Baiardi and D. Sgandurra. Building Trustworthy Intrusion Detection through VM Introspection. In *Proceedings of the Third International Symposium on Information Assurance and Security (IAS)*, pages 209–214, 2007.

[3] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the Art of Virtualization. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP)*, pages 164–177, 2003.

[4] C. Benninger, S. Neville, Y. Yazir, C. Matthews, and Y. Coady. Maitland: Lighter-weight vm introspection to support cyber-security in the cloud. In *Proceedings of the 5th IEEE International Conference on Cloud Computing (CLOUD)*, pages 471–478, 2012.

[5] S. Biedermann and E. Tews. How to enable Live Cloning of Virtual Machines using the Xen Hypervisor. Technical report, 2013.

[6] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live migration of virtual machines. In *Proceedings of the Symposium on Networked Systems Design & Implementation*, pages 273–286, 2005.

[7] B. Dolan-Gavitt, T. Leek, M. Zhivich, J. Giffin, and W. Lee. Virtuoso: Narrowing the semantic gap in virtual machine introspection. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, pages 297–312, 2011.

[8] T. Fraser, M. Evenson, and W. Arbaugh. Vici: Virtual machine introspection for cognitive immunity. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, pages 87–96, 2008.

[9] Y. Fu and Z. Lin. Space Traveling across VM: Automatically Bridging the Semantic Gap in Virtual Machine Introspection via Online Kernel Data Redirection. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, pages 586–600, 2012.

[10] T. Garfinkel and M. Rosenblum. A virtual machine introspection based architecture for intrusion detection. In *Proceedings of the Network and Distributed Systems Security Symposium (NDSS)*, pages 191–206, 2003.

[11] Z. Gu, Z. Deng, D. Xu, and X. Jiang. Process implanting: A new active introspection framework for virtualization. In *Proceedings of the 30th IEEE Symposium on Reliable Distributed Systems (SRDS)*, pages 147–156, 2011.

[12] H. Huang, W.-T. Tsai, and Y. Chen. Autonomous hot patching for web-based applications. In *Proceedings of the 29th Annual International Computer Software and Applications Conference (COMPSAC)*, volume 2, pages 51–56, 2005.

[13] M. Payer and T. Gross. Hot-patching a web server: A case study of asap code repair. In *Proceedings of the 11th Annual International Conference on Privacy, Security and Trust (PST)*, pages 143–150, 2013.

[14] B. Payne, M. Carbone, M. Sharif, and W. Lee. Lares: An architecture for secure active monitoring using virtualization. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, pages 233–247, 2008.

[15] B. Payne, M. de Carbone, and W. Lee. Secure and flexible monitoring of virtual machines. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, pages 385–397, 2007.

[16] A. Ramaswamy, S. Bratus, S. Smith, and M. Locasto. Katana: A Hot Patching Framework for ELF Executables. In *Proceedings of the International Conference on Availability, Reliability, and Security (ARES)*, pages 507–512, 2010.

[17] Y. Sun, Y. Luo, X. Wang, Z. Wang, B. Zhang, H. Chen, and X. Li. Fast Live Cloning of Virtual Machine Based on Xen. In *Proceedings of the 11th IEEE International Conference on High Performance Computing and Communications (HPCC)*, pages 392–399, 2009.

Challenges and Implications of Verifiable Builds for Security-Critical Open-Source Software

Xavier de Carné de Carnavalet and Mohammad Mannan
Concordia Institute for Information Systems Engineering
Concordia University, Montreal, Canada
{x_decarn, mmannan}@ciise.concordia.ca

ABSTRACT

The majority of computer users download compiled software and run it directly on their machine. Apparently, this is also true for open-sourced software – most users would not compile the available source, and implicitly trust that the available binaries have been compiled from the published source code (i.e., no backdoor has been inserted in the binary). To verify that the official binaries indeed correspond to the released source, one can compile the source of a given application, and then compare the locally generated binaries with the developer-provided official ones. However, such simple verification is non-trivial to achieve in practice, as modern compilers, and more generally, toolchains used in software packaging, have not been designed with verifiability in mind. Rather, the output of compilers is often dependent on parameters that can be strongly tied to the building environment. In this paper, we analyze a widely-used encryption tool, TrueCrypt, to verify its official binary with the corresponding source. We first manually replicate a close match to the official binaries of sixteen most recent versions of TrueCrypt for Windows up to v7.1a, and then explain the remaining differences that can solely be attributed to non-determinism in the build process. Our analysis provides the missing guarantee on official binaries that they are indeed backdoor-free, and makes audits on TrueCrypt's source code more meaningful. Also, we uncover several sources of non-determinism in TrueCrypt's compilation process; these findings may help create future verifiable build processes.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification; K.6.5 [**Management of Computing and Information Systems**]: Security and Protection

Keywords

Deterministic build, Reproducible build, TrueCrypt, Tor, Bitcoin, Debian

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.
ACSAC '14 December 08 - 12 2014, New Orleans, LA, USA
Copyright 2014 ACM 978-1-4503-3005-3/14/12…$15.00
http://dx.doi.org/10.1145/2664243.2664288

1. INTRODUCTION

When building a software package, the compilation process generally results in different binary outputs from one compilation instance to another, especially in different environments. This leads to various problems from deduplication to trust issues. In this paper, we are mostly concerned about the security implications of such a build process, more specifically for security-critical software systems, e.g., TrueCrypt, Tor, Bitcoin Core [3], and Debian. The idea of reproducing strictly identical builds, i.e., being able to independently reconstruct the same binaries as advertised by the developers using the same source files, has been suggested only a few years ago; the oldest reference we can find is a discussion from 2007 on the Debian mailing list [46]. Several blog articles and tickets in bug tracking systems refer to this idea as machine-independent builds [7], deterministic builds [6, 35], bit-by-bit identical builds [28], idempotent builds [26], verifiable builds [43, 11], bit-perfect [12], or byte-by-byte reproducible builds [4, 5]. Some ad-hoc solutions have been discussed in these references. However, we could not locate any reference on the verifiability of open-sourced application binaries in academic literature, which we believe is a critical, but mostly overlooked aspect of current open-sourced software distribution infrastructure.

Making a build process deterministic is evidently non-trivial, as compilers and other tools in the toolchain have not been designed with this goal in mind. One of the most common sources of difference comes from embedded timestamps that reflect the time of compilation. Two compilation instances run at one second apart on the same environment may generate slightly different outputs. More difficult examples that we have found include variations linked to the type of the processor used for compilation. File checksums and cryptographic hashes differ as a result of such changes. As such hashes are used when comparing files or verifying their integrity, it then becomes irrelevant to compare hashes of different builds of the same source.

On the other hand, several existing security mechanisms rely on randomized build processes to achieve automated software diversity to restrict large-scale exploitations of software vulnerabilities. Possibly the best example of security through diversity is address space layout randomization (e.g., PaX ASLR [41]), which has been implemented in most widely-used operating systems. Software diversity has a rich history; see e.g., Forrest et al. [14], and Franz [15] (also the recent survey on diversity [27]). By definition, software diversity mechanisms are mostly adverse towards deterministic builds. Other sources of diversity in binary code may in-

clude compiler optimization techniques, especially the ones independent of source code, e.g., profile-guided optimization (PGO [31], recently adopted by Mozilla Firefox [33]).

The importance of deterministic builds has gained general awareness after classified documents from the U.S. National Security Agency (NSA) have been leaked by Edward Snowden starting in June 2013. The magnitude and depth of the intelligence programs explained in these documents suggest, among other things, the ability of the NSA to perform surreptitious man-in-the-middle attacks to infect targeted systems, or more generally make those systems download malicious software (cf. NSA TURBINE [16] and the Dual_EC_DRBG case [29]). It then becomes easy to insert malicious behavior such as backdoors in binary files that are downloaded by a majority of users, particularly on the Windows platform. Man-in-the-middle attacks can be prevented by the use of TLS-encrypted communications. However, such protection does not prevent a malicious or coerced software author from distributing malicious binaries that do not match the available source code. Moreover, an author can digitally sign her software package, e.g., via an embedded X.509 certificate on Windows executables; however, this may constitute a single point of failure (cf. Flame malware [20]). The ability to verify both the source code and the corresponding executable against a consensus could prevent such attacks on downloaded software.

Security-sensitive applications such as Bitcoin Core and Tor have started using a deterministic build approach, to enable decentralized trust through multiple independent signatures from users who compiled the application and submitted signed hashes of the output. Both Bitcoin Core and Tor are based on Gitian [17], a Ruby and bash wrapper for Ubuntu's python-vm-builder that was created in 2009 to ensure the build security and integrity of Bitcoin. In 2013, the Tor Browser Bundle [35] adapted Gitian for its purpose, which involves the cross-compilation of the bundle for Mac OS X and Windows from a Linux environment. However, several aspects for a deterministic build remain unaddressed.

As a case study, we focus on TrueCrypt [44], an open-source tool capable of on-the-fly encryption.[1] It creates encrypted containers at different levels (file, partition and entire disk), provides a full disk encryption (FDE) feature under Windows with pre-boot authentication, and even offers plausible deniability of hidden partitions. TrueCrypt is available as compiled binaries for Windows, Linux and Mac OS X, along with its source code. However, as we mentioned, simply compiling the source code is not enough to replicate a match with the given binaries. Hence, reviews and audits of its source code usually discard this difficult verification step, and may qualify TrueCrypt as secure, regardless of whether the audited source corresponds to the targeted compiled application that is distributed and used by the majority of users. Our analysis brings this missing connection and makes the conclusions of those audits more meaningful. Similar to other security software, such as Tor, TrueCrypt also appears to be a perfect target for government surveillance (see [16]). A legitimate question then arises: Are the binaries provided on the website different than the available source code and do they include hidden features such as a backdoor? In this paper, we provide the answer to this question and make the following contributions:

1. We recompiled sixteen versions of TrueCrypt from v5.0 to v7.1a and analyzed the differences between our builds and the official ones. These versions date back from February 2008 up to the latest fully-functional version released in February 2012.
2. We detail the major challenges we faced to replicate a close match, and successfully explain the remaining differences, if any. We then conclude that all of TrueCrypt's signed binaries directly come from their respective sources and no backdoor has been inserted in the binaries.
3. We identify several key sources of non-determinism to be taken into account to realize deterministic builds. Our methodology can also help verify other applications that do not provide a deterministic build.
4. Finally, we summarize the lessons learned from other ongoing projects that aim at achieving deterministic or reproducible builds, and their limitations.

2. DEFINITION, THREATS AND CHALLENGES

2.1 Definition

The general idea behind a deterministic build is to record the environment when building the official release of a project, then replay the behavior of this environment in later builds to achieve the same results. This process removes sources of non-determinism that are out of control in a regular building process. A broader term has been coined by Debian as reproducible build [10], which emphasizes more on replicating the official build, regardless of the process involved. To reconcile existing terminologies, we suggest the following definition:

A build is verifiable if any two instances of the build process produce identical results. This can be achieved through a deterministic process, in which case both builds are byte-by-byte identical (and hence the process is machine-independent); or by matching the builds at a higher semantic level (e.g., by ignoring unimportant differences).

2.2 Assumptions

We assume that the compiler is trusted; cf. Thompson [42] (see also [49] for a proposal addressing the untrusted compiler problem). We also trust the operating system (OS), as it would make no sense to trust a regular program (e.g., a compiler in this case) running on an untrusted OS. The hardware platform including the CPU of the build system is also trusted (but see e.g., [25]).

We also make the following assumption regarding the multiple independently signed hashes of recompiled binaries. If the verification process succeeds by consensus (i.e., the majority of the signed hashes match the hash computed by the user), we assume that malicious signatures cannot represent a majority in the list of signatures. This assumption does not hold in situations where a powerful attacker can compromise a significant number of signatures (i.e., beyond the majority threshold) with ones that correspond to malicious binaries. Other verification schemes could be considered (e.g., see Perspectives [48]). Finally, note that deterministic builds are only interested in matching sources with

[1] Even though recently the TrueCrypt developers announced the discontinuation of their work, other groups are joining effort to keep such an essential tool alive; see, e.g., http://truecrypt.ch. The TrueCrypt audit project (http://istruecryptauditedyet.com) also remains active.

binaries; hence, we do not make any assumption regarding the trustworthiness of the source, which remains under the scope of source-code audits.

2.3 Threats considered

Our primary consideration is that users do not recompile software packages from the source. Based on it, we consider the following threats and explain how a deterministic build prevents against them.

Targeted attacks on binaries. We assume that an attacker can alter the binaries received by a targeted user (e.g., a coercive government against a Tor user). Such modifications will remain undetected if the integrity of the delivery channel or an author's signing key is compromised. This attack is applicable to TrueCrypt, as it could only be downloaded through an insecure channel without a TLS-encrypted connection. A deterministic build provides a match between the source of an application and its compiled version distributed to users. The match operates at the file level on the output side, enabling hash comparison of output files and verification of independent signatures; such verification enables decentralized trust. Thus, a user can simply compare the hashes of her copy of the application against the independently signed hashes, and identify whether she has been subject to a targeted attack.

Untrusted authors. We assume that an untrusted open-source developer (coerced or malicious) in charge of compiling the official build, provides backdoored binaries for distribution through the official channel (e.g., website or update server), but leaves the available source untouched. If targeted attacks on binaries are addressed, developers can no longer provide binaries that do not match the source, since the official signatures would differ from any independent ones generated by recompiling the application from its source. Addressing targeted attacks on binaries hence provides the side effect of protecting against untrusted authors.

Targeted attacks on the source. We assume that an attacker can alter the source obtained by a targeted user. Such type of attackers can include the developers of an application, who wish to mislead a source reviewer (as in the case of a security audit). If targeted attacks on binaries are addressed, a user wishing to recompile the application from the source benefits from an additional feature: she can also verify that the source code she obtained corresponds to the official binaries that have been independently signed. Indeed, if the hashes of the recompiled binaries do not match the ones that are independently signed, the user will detect that she has obtained a different source. The granularity of the detection of mismatching inputs may vary (i.e., a single file or a group of files), since mismatching binaries can be compiled from several input files.

Targeted attacks on both the source and binary files. In a particular case of targeted attack, we assume an attacker can alter both the source and binaries for a specific user through a man-in-the-middle attack, or through the official delivery channel. Such a user could then be tricked into thinking that she has verified the official build by recompiling the source herself and matching her build with the official downloaded binaries. Even though a TLS-encrypted channel provides better security against a man-in-the-middle attack, it does not fully prevent against this threat (e.g., if altered files are made available through the official channel for a particular user). Multiple independent signatures, enabled by deterministic builds, reduce the probability of such attacks, since the difficulty to compromise several keys would also increase.

A deterministic build thus bridges the gap between sources and binaries in both directions, i.e., it allows any user to match binaries to reference sources, and to match sources to reference binaries. Also, it does so without involving average users into any technical details of the compilation process.

2.4 Verifying non-deterministic build

Context. Supposing that all future open-source software follow a deterministic build process, one may still be concerned about past software packages that provide compiled binaries but were not compiled with verifiability in mind. In this case, the problem is to match the sources with the available binaries that are (ideally) signed by the developer, or provided through an integrity-protected channel to consider them as official. TrueCrypt falls in this category, which relates to the untrusted authors threat.

Feasibility. A naive approach to achieve the verification of past software is by manually inspecting each version of a selected application and replicating a close match, then explaining any remaining differences. A custom build process could be created for a specific software instance to make it deterministically buildable. It would probably be version-specific, as supporting several versions and handling many potential environments would require designing a universal deterministic builder. However, manual review of past software is painstaking, as illustrated by our case study in Section 3, and adapting a deterministic build that exactly matches the official binaries may be sometimes impossible.

Automated verification of past software can also be very challenging. In case the required resources (e.g., the compiler information, project configuration or building steps) are unavailable or not properly identified, one would need to leverage compiler fingerprinting techniques (cf. FLIRT [21]) to identify the optimization level and other options passed to the compiler, along with the correct dependencies. In addition, software-specific differences (e.g., TrueCrypt installer's custom checksum, see Section 3.3.3) may prevent generalization of an automated verification technique for such non-deterministic builds.

Note that verifying future deterministic builds and past software packages do not share the same requirements. For packages whose output is not deterministic, the equivalence with the official build needs to be proven for every build, while waiting for newer versions that will hopefully leverage a deterministic build process. This situation also highlights the fact that a build does not need to lead to an output that is byte-by-byte identical with the official build to be verifiable. Superficially variable areas (e.g., timestamps) can be ignored by a high-level comparison, as opposed to a simple file hash comparison. As we observed in our analysis of TrueCrypt in Section 3, the remaining differences we faced after setting up a proper environment were found to be legitimate and explainable, as opposed to malicious differences.

3. CASE STUDY: TRUECRYPT

TrueCrypt does not provide any explicit way to verify its compiled binaries as made available for Windows, Linux and Mac OS X. Anyone wishing to compile the sources will get

binaries different than the official ones, as identified by others in the past (see e.g., [37]). This has led to some speculations regarding the possibility of having backdoors in the official binaries that cannot be found easily as they would not be apparent from the source. This concern has also been raised in a security analysis of TrueCrypt by the Ubuntu Privacy Remix Team [45], in which the authors conclude that they cannot link the result of their code analysis to the official binaries because it would require "a very expensive reverse engineering".

In this section, we present the challenges we faced to compare all the official releases of TrueCrypt for Windows since February 5, 2008, including versions 5.0, 5.0a, 5.1, 5.1a (and its second release), 6.0, 6.0a, 6.1, 6.1a, 6.2, 6.2a, 6.3, 6.3a, 7.0, 7.0a, 7.1 and 7.1a (February 7, 2012) with the alleged corresponding sources. An analysis of version 7.1a was also documented by the first author in an online report [8];[2] we reuse parts of that report here. We present the challenges in the order as we faced, to highlight the difficulties posed by the sources of non-determinism, and the reasoning we followed to get down to the root cause of them.

3.1 Our test environment

To replicate a clean environment with the control over which compiler and tools are installed, we leverage the snapshot feature of VMware Workstation. It enables the creation of virtual machines snapshots that we can later fork to a different path. This feature is useful when installing multiple applications in a series, as we can take a snapshot after the installation of each of them, and later backtrack to a particular snapshot to continue installing different versions of the remaining tools in the toolchain. We configured three virtual machines: one running Windows 7 Professional 64-bit edition and the two others running Windows XP Professional 32-bit edition; in total, we created 45 snapshots, which took about 175GB of disk space. For our experiments, we mostly use a PC with an Intel Core i5-2400 processor; we also performed some tests on another machine with an AMD FX-8350 processor to confirm the origin of yet another source of non-determinism involving the CPU manufacturer.

3.2 Preparing the environment

3.2.1 Prerequisites

In the latest version of TrueCrypt's source package, the `readme` file specifies the following list of software, tools, SDK and additional files as requirements (as quoted from the file): (a) Microsoft Visual C++ 2008 SP1 (Professional Edition or compatible); (b) Microsoft Visual C++ 1.52 (available from MSDN Subscriber Downloads); (c) Microsoft Windows SDK for Windows 7 (configured for Visual C++); (d) Microsoft Windows Driver Kit 7.1.0 (build 7600.16385.1); (e) RSA Security Inc. PKCS #11 Cryptographic Token Interface (Cryptoki) 2.20 header files; (f) NASM assembler 2.08 or compatible; and (g) gzip compressor.

For the older versions of TrueCrypt we studied, older versions of similar tools are required. Going back to version 5.0, we need to gather Microsoft Windows Driver Kit (WDK) for Windows Vista (build 6000) and version 7.0.0 (build 7600.16385.0), Microsoft WDK for Windows Server 2008 SP0/SP1 (build 6001.18001/2), Microsoft Visual C++ 2005 SP1 Professional Edition, NASM assembler 0.99 and 2.06, along with Yasm assembler (no version specified).

The first problem in creating the build environment is that it requires compilers and resources that can be difficult to find or are non-free; notably, Visual C++ 1.52 that was released 20 years ago (in 1994) and is only available to MSDN subscribers (which membership costs at least US$1,200 at the time of the writing), or for Microsoft Certified Trainers (MCT). In our case, we first had to search for copies online before being provided with the original file by an anonymous contributor with an MSDN subscription, in reaction to our online report. Indeed, the academic access to MSDN (MSDNAA) of our university was not privileged enough to access such old software. For Microsoft Visual C++ 2008 (Visual Studio), one also needs to have an MSDN subscription since it is now an old version that is no longer publicly available as a trial version on Microsoft's website (although a direct link to the ISO file on Microsoft's servers can be found by crawling the web). This software was accessible via our MSDNAA access.

Second, Microsoft WDK 7.0.0 and below are no longer available at Microsoft at the time of the writing (May 2014) because of newer versions that superseded the previous ones, but was available at the time of the first analysis (October 2013). This also highlights a general problem for verifying even relatively new software, as current dependencies may become permanently unavailable at any time. Additionally, there exist three versions of the Windows 7 SDK: version 7.0 for .NET Framework 3.5 SP1, v7.0A included in Visual Studio 2010 only, and v7.1 for .NET Framework 4.0. Also, we do not know *a priori* to any test whether a different version than the one in the authors' environment can lead to changes in the compiled output. Fortunately, is it still possible to find direct links to download old SDKs at Microsoft. For old WDKs though, only MSDN subscribers can obtain legal copies, which pushed us to search for other channels due to limitations of our academic access to MSDN. We wish to thank anonymous contributors for providing us with the missing pieces that we could not find. We verified the hashes of the files we received against the official hashes published on the MSDN website.

Finally, the RSA PKCS #11 Cryptographic Token Interface (Cryptoki) 2.20 header files and NASM assembler 2.08 are freely available online. The gzip compressor is also available for Windows thanks to the GnuWin project [18]. The version number used by the developers is not mentioned. However the latest GnuWin's gzip version (1.3.12-1) dates back to 2007, so we assumed that this version or a compatible one was used by the original developers. A different compressor or version of that compressor can lead to a different compression algorithm or file format and result in a different output. The GnuWin's version of gzip (1.3.12-1) fortunately worked for our purpose. We later found in the source code that gzip.org's release was suggested by the developers.

Although not mentioned in the TrueCrypt's project `readme` file but pointed out in [39], the *dd* tool [22] is also required during the build process. Some dd ports for Windows do not behave correctly during the compilation process (e.g., different arguments are expected and no output is generated). We first used an incompatible version (from chrysocome.net) that achieved incomplete builds. We later found a working version in the CoreUtils package for Windows [18].

[2]The article was discussed in a Slashdot post; see: http://it.slashdot.org/story/13/10/24/169257/how-i-compiled-truecrypt-for-windows-and-matched-the-official-binaries.

Table 1: File names and sizes from our first compilation attempt (VS2008 SP1 without updates) vs. the original ones

File name	Sizes in bytes	
	Our build	Official build
TrueCrypt.exe	1,507,840	1,516,496
TrueCrypt Format.exe	1,602,048	1,610,704
truecrypt.sys	224,128	231,760
truecrypt-x64.sys	223,744	231,376
TrueCrypt Setup.exe	1,056,768	N/A
TrueCrypt Setup 7.1a.exe	3,432,471	3,466,248

3.2.2 Initial challenges

Once the environment is correctly installed, we can open the Visual Studio solution file *TrueCrypt.sln* and build it in "release" configuration.

Flat comparison. Our first naive attempt takes place in a virtual machine running Windows 7 Professional 64-bit edition with the prerequisites installed. Table 1 shows the sizes of our compiled binaries and the official ones from v7.1a.

File sizes do not match, leading to different hashes. If the build process was made deterministic, we would have a match at this point. First of all, one obvious reason for the file size difference is the presence of digital signatures (or lack thereof) that we explain below. However, there are also a number of other differences that we explore in this section.

The official binaries are all signed with an embedded X.509 certificate that belongs to the "TrueCrypt Foundation", which increases the file size. This is an issue for the purpose of reproducing an exact match, since it is impossible to generate a signature for the binaries on the developers' behalf. Hence, it is not possible to reproduce an exact match of any of the binaries if we are not in possession of the official build's files. Indeed, the original signatures can be extracted from the official files and reused in our build as is. These signatures would be valid if the rest of the files also matched. Also, note that the installer (TrueCrypt Setup.exe) is incomplete after compilation. We must run it with a special flag (`-p`) to package the other binaries together and output a complete installer named TrueCrypt Setup 7.1a.exe. In Table 1, we packaged our own binaries for comparison. To replicate an exact match, the binaries should be signed prior to being packaged. We therefore ignore the installer until we can reproduce the other binaries.

Visual Studio updates. The digital signature represents a small fraction of the binary files (7,631 bytes in each file in v7.1a), and is placed at the end of the files, which makes it easy to locate. However, we found that a byte-by-byte comparison of each file still produces significant changes, beyond the signature. We then considered an advanced comparison technique that attempts to identify matching blocks of bytes, even if their sizes slightly differ. Such comparison generally yields a score that represents more accurately the real content difference. With respect to the official file sizes and using such a comparison technique,[3] we find 60,049 (4.0%) of mismatching bytes in TrueCrypt.exe, 34,746 bytes (2.2%) in TrueCrypt Format.exe, 7,673 bytes (3.3%) in truecrypt.sys, and 7,878 bytes (3.4%) in truecrypt-x64.sys.

We further investigated additional patches available for Visual Studio 2008 SP1 that do not change the version number of the software but lead to major changes in the output.

[3]We used Beyond Compare v3 (scootersoftware.com).

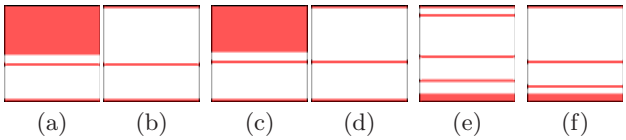


Figure 1: Visual differences on the binary files between our build and the official one for TrueCrypt 7.1a using an advanced comparison technique. (a), (c), (e) and (f) represent binaries compiled without any updates for VS2008 SP1 in our build while others are compiled with the proper ones. Files in (e) and (f) are not influenced by such updates. (a) and (b) represent TrueCrypt.exe; (c) and (d) TrueCrypt Format.exe; (e) truecrypt.sys; (f) truecrypt-x64.sys. Red areas represent approximate differences with coarse granularity (the thinnest red line represents up to 2.8% differences).

In other words, we should restore an environment that contains the same updates for the compiler as what the authors had installed when they compiled the official release. As there is no mention about which patches were installed in their environment, we needed to manually go by trial and error. To reduce the differences between our build and the official binaries, we figured that we needed to install all the updates that were available when a particular version of TrueCrypt was released. The core updates are KB971092, KB973675, and KB972222 for versions 6.1 to 7.0a, with the addition of KB2538241 for 7.1 and 7.1a. With these only updates installed, we minimize the differences. Figure 1 shows a visual representation of the differences between our build and the official files in the case where no updates are installed, or the proper updates are installed. This indicates that the developers had their system up-to-date, however such observation may not stand in general.

For versions 5.0 to 5.1a that were compiled with Visual Studio 2005 SP1, a single core update (KB937061) is available, which was released five months before the release of TrueCrypt v5.0. We hence installed it before carrying our experiments, unlike our first tries with more recent versions of TrueCrypt. For versions 6.0 and 6.0a, no update was available for Visual Studio 2008 at the time of their release.

Thus, for the purpose of verifying official builds, it is important to use the same exact version of the compilers and tools installed on the developers' build machine, since a slight difference can significantly change the output binaries.

3.2.3 Build path

The files truecrypt.sys and truecrypt-x64.sys are the 32-bit and 64-bit Windows drivers, respectively, which take care of all features related to the OS, such as providing virtual disks and supporting full disk or system partition encryption. The number of differences between our build and the official one remains significant. While investigating them, we found that these files contain various debug information, including the full project path. These debug parameters are references to a Program DataBase (PDB) debug info file and are inserted by the linker to match the corresponding debug file [32]. The short debug section in the binaries contains the path of the PDB debug file, located under the project directory. The project folder in our test environment was located on the user's desktop while the developers had it apparently in c:\truecrypt-7.1a. The difference in the path length leads to a shift of the remaining data in this debug section, plus an offset in some addresses that point to locations in the file. By compiling the software again after

moving the project directory to the same location as in the official build, these differences were gone. Note that, in the other files, there is no debug information and hence no link to the build path. Hence, the build path can sometimes influence the output and should be taken into account while setting up the environment. Throughout the 16 versions we analyzed, the project path usually contains the version number and is located in the root of the C drive; however some project paths are found without a reference to the version number and/or are located in an E or T drive. As we found, the project paths are c:\truecrypt-7.1 for version 7.1, c:\truecrypt-7.0a for 7.0a, c:\truecrypt for 6.3a, t:\truecrypt for 6.3, c:\truecrypt-6.2a for 6.2a and similar down to 5.1a, e:\testworkspace\truecrypt\main for 5.1, 5.0a and 5.0.

3.3 Sources of non-determinism

To understand the differences between our compiled binaries and the original ones, we perform a byte-by-byte comparison. Below we analyze each file individually. We mainly focus on the latest version of TrueCrypt (v7.1a) with all addresses and offsets cited referring to this version, unless otherwise specified.

3.3.1 TrueCrypt.exe and TrueCrypt Format.exe

Visually from Figures 1 (b) and (d), there are three main regions that host differences in TrueCrypt.exe.

First region. The first region is located in the file header, and contains three differing elements. The first one is at file offset 0x000000F8 that corresponds to the time/datestamp in the COFF/PE file header. The second is located at file offset 0x00000148 that corresponds to the checksum in the PE optional header. Finally, the third one at offset 0x00000188 that corresponds to the certificate table in the optional data directories header.

According to Microsoft documentation [30], time/datestamp corresponds to the time and date the file was created. Since we compiled TrueCrypt at a different time than the developers, this difference is legitimate. Then, the checksum corresponds to the image file checksum based on a Microsoft-specific algorithm [47]. This checksum is different because our compiled executable has slight differences that are covered by this checksum, resulting in different values, for which we verified the correctness. Hence, this difference is also legitimate. Finally, the certificate table contains a certificate data field, which is a pointer to a certificate in the file; along with field about the certificate size. This table provides information regarding the X.509 signature over the file that is included in the official binaries. Because we do not have certified binaries, the certificate table is all zeros, whereas the original file points to some certificate data at offset 0x00170600 in the file (see the third region).

Second region. The second region of differences is located at about two thirds of the file. It corresponds to a readable time and date written in English and what seems to be also a timestamp encoded as a 32-bit integer. Converting this alleged timestamp to a readable date and time, we find that it matches the written date, confirming it is simply a timestamp. Interestingly, we can deduce the time zone of the compiling machine, since the time written is interpreted with respect to the machine settings while the timestamp is a POSIX time representing the number of elapsed seconds relative to a reference UTC time. In this case, the timestamp of the official binary converts to 2012-02-07 09:08:50 UTC while the written date and time reads "Tue Feb 07 10:08:49 2012". This informs that the time zone of the machine which built the executable was set to UTC+1 (CET).

In the 11 versions from 5.0 to 6.3, more differences arise in this section. Further investigations reveal that they are located in the resource section of the executable, and correspond to an Interface Definition Language (IDL) file compiled by the Microsoft IDL compiler. The OLE-COM Object Viewer in Windows SDK can decompile this resource. By comparing our decompiled IDL resource to the decompiled official one, only the timestamp and written date remain different, meaning both resources are functionally identical. Despite our efforts to recreate many possible build environments, we were unable to reproduce an IDL resource that only includes differences in the timestamps in a way that is visible from the compiled format directly, as it is the case in later versions of TrueCrypt.

For version 6.3, using the SDK for .NET Framework 3.5 SP1 (v7.0) that was available at the time of release of this version of TrueCrypt, yields a different Rich signature (see [36]) in the DOS stub part of the PE header of TrueCrypt's executables. The difference resides in the *comp.id* row number 148 that corresponds to a subversion 30729 in our case, while the original file shows a subversion 21022. This indicates that a component in the toolchain differs. Once we switched to the latest version (v7.1), this problem was solved and only the difference in the IDL resource remained. However, this later version of the SDK was released 7 months after the release of TrueCrypt 6.3, and thus could not have been used by the developers. One possible explanation is that the authors had Visual Studio 2010 installed on their system, which comes with the intermediate version 7.0A of the SDK, and configured Visual Studio 2008 to use it. However, Visual Studio 2010 was also released 6 months after this version of TrueCrypt. The only realistic possibility is that the authors were using Visual Studio 2010 beta 1 or 2 on their machine, which was released 5 months and 2 days, respectively, before TrueCrypt 6.3. They would have had to configure Visual Studio 2008 accordingly. We were unfortunately unable to find this beta version for further tests. This problem demonstrates that the sources of non-reproducibility can be difficult to identify and thus difficult to troubleshoot.

Third region. Finally, at the end of the file, the third region of differences starts at 0x00170600 and shows us that the original file contains more information. Recall from the first region that this offset is directly pointed to by the certificate table and hence is related to the digital signature. We can safely ignore the presence of the certificate in the official binaries, since a signature and certificate are normally harmless (cf. Microsoft's documentation: "[t]hese certificates are not loaded into memory as part of the image." [30]) Thus, if this section contains malicious code, it has to be loaded by the program first, which would be seen in the source code (its review is out of our scope, though).

It is to be noted that apart from these three unimportant mismatches (timestamps, checksum, presence of certificate), the rest of the files are identical. This indicates that our TrueCrypt.exe and the official one are identical. Also, in TrueCrypt Format.exe, we find the same patterns of difference as TrueCrypt.exe. As we explained the unimportance of these differences in the case of TrueCrypt.exe, we can con-

clude that our TrueCrypt Format.exe and the official one are also coming from the same source.

3.3.2 Truecrypt.sys and truecrypt-x64.sys

Apart from the build path issue that we solved when creating the build environment, other differences remain.

In the 32-bit driver, a difference at file offset 0x00000270 corresponds to the time/date-stamp in the PE headers. File offsets 0x0001EA44 and 0x00034184 show the same timestamp difference. More specifically, the one at 0x0001EA44 matches the timestamp location in the debug directory structure [30], which contains the address of the debug section. In turns, this debug section is located at file offset 0x0002CBA8 in the 32-bit driver and offset 0x0002F490 in the 64-bit driver, where another difference is found (see the next paragraph). Similarly, the timestamp at offset 0x00034184 matches the location of the timestamp in the export directory table. File offset 0x000002C0 is the optional PE checksum header, which also differs for the same reason as in the previous files: the file has slight differences that lead to a different checksum. File offset 0x00000300 represents the certificate table difference, which as we explained is expected. The end of the original file contains the certificate.

In the PDB debug section, pointed by the debug directory previously mentioned, we still find one difference of 16 bytes starting at 0x0002CBAC. The format of this section is undocumented, however it is explained in [19] as containing a signature (the string "RSDS"), followed by a GUID (Globally Unique IDentifier) that is regenerated in each build and is used by the debugger to link a binary file to its PDB debug file. Our 16-byte difference matches the location of the GUID. Next to it is an "age" field followed by the path of the debug file, which led to a difference we had previously resolved by using the same project path as in the official build. As the project was compiled in release configuration (not in debug configuration), such information should not be present at all in the output files. Their presence remains unclear.

In versions 6.3 and 6.3a, the `readme` file suggests the use of NASM "version 2.06 or compatible". However, if we use version 2.06 during the build process, NASM will crash when assembling the 64-bit driver. While investigating this problem, we found a ticket on the NASM bug tracking system [40] mentioning this specific issue. The issue has been resolved in 2.08-rc1, which was released one week before TrueCrypt v6.3's realease. One may further correlate this bug ticket with the internal development of TrueCrypt.

The 64-bit version of the driver, truecrypt-x64.sys, shows the same patterns. However, in version 6.2 and 6.2a, we observe an additional difference spanning on 5 consecutive bytes at file offset 0x0001CFCB. Contrary to other differences that we identified so far, this one does not affect metadata but rather is located in the `.text` section of the binary file, which contains the logic of the driver. Our build reads `0F 1F 44 00 00` while the official build reads `66 66 90 66 90`. The disassembled binary files show a more comprehensive comparison shown below.

```
66 66 90          data32 xchg ax,ax
66 90             xchg   ax,ax
0F 1F 44 00 00    nop    DWORD PTR [rax+rax*1+0x0]
```

Functionally, both sequences are effectively realizing No Operation (NOP). These NOPs only serve alignment for the remaining code. We can find a partial explanation for this difference in Intel's documentation [23], which lists sequences of various lengths that realize a NOP (in Table 4-9 entitled "Recommended Multi-Byte Sequence of NOP Instruction"). Our NOP corresponds to a 5-byte NOP in this table. However, we cannot find the official build's NOP sequence in this table. The explanation actually lies in the type of processor used in the build environment. While Intel's documentation recommends `0F 1F 44 00 00` for a 5-byte NOP, AMD's documentation [1] recommends `66 66 90 66 90`. We can infer that the compiler determines the processor it is running on and adapts its output accordingly. This hypothesis is confirmed after we compiled this version of TrueCrypt on a machine with an AMD processor. The 5-byte NOP was present as in the official build. We can further deduce that the developers were using an AMD processor for the release of versions 6.2 and 6.2a.

3.3.3 TrueCrypt installer

Now that the remaining files have been analyzed and the differences between our build and the official one have been explained, we can package the original files with our compiled installer. After packaging the original files with our compiled installer, we obtain an installer of 3,458,614 bytes, which is close to the original installer's size (3,466,248 bytes). Again, the usual time/date-stamp, checksum and certificate table differ, and the original installer has a certificate at the end of its file. A new difference occurs at 0x0034C632 on 4 bytes that look like a checksum. By investigating TrueCrypt's source code, we can find that it is indeed a checksum. During the packaging of the files, the installer computes an integrity checksum over its complete version. At this point, the installer is not yet signed, which means it does not embed a certificate and its certificate table is all zeros. However, after the installer is signed, it itself needs to be able to recompute the checksum over its unsigned version. To achieve this, the installer computes the checksum by replacing bytes between file offsets 0x00000130 and 0x000001FF (inclusive) with zeros, so as to zero out the Certificate Table. This range however contains more information than just the certificate table, specifically half of the optional PE header and the whole data directories, including the Export, Import, Resource, Exception, Base Relation, and Import Address tables. In practice, erasing this byte range effectively deletes these fields in the header, and weakens the coverage of the integrity check. The installer also truncates the file after the magic word "TCINSCRC", which is located right before the checksum in question, effectively deleting the digital signature. The CRC32 computed over this modified file corresponds to the alleged checksum we are investigating.

Version 5.1a was released twice; the second edition was released the same day as version 6.0. This second edition repackages the original files with a new installer derived from the installer of version 6.0. The main apparent goal was to update the license shown during the installation. Comparing this new installer with the installer from v6.0 however shows significant differences. Once disassembled, many addresses differ, some part of the code is changed (mostly for alignment), and eight resources are completely different. As the exact source of this new installer was never made available, we are left with the option of reverse-engineering the installer or trying to recreate its source, based on the knowledge that it is an intermediate version between 5.1a and 6.0.

After several experiments, we concluded that this installer comes from v6.0 with changes made to VERSION_STRING and VERSION_NUM constants to simulate version 5.1a. Once compiled, the installer packages the original files from v5.1a with the new License.txt file. Incidentally, when using this new installer on a system-encrypted environment, the bootloader installed by the installer is in fact the one from v6.0 renamed as v5.1a (which caused the differences in the resource section as several versions of the bootloader exist and they are gzipped in that section). This brings another source of non-determinism, which is any (allegedly unimportant) code modification that the authors made to compile a build without releasing or documenting them.

3.4 Summary

In this analysis, we showed that the compiled versions 5.0 to 7.1a of TrueCrypt are directly compiled from the available source code. From a security perspective, this shows TrueCrypt is not backdoored by the developers in a way that is not visible from the sources, i.e., the provided executables for Windows are not added with any feature not visible from the sources; addressing the untrusted authors' threat.

From a software engineering perspective, our study helps identify sources of non-determinism in a simple Visual Studio project over the years, from which many findings can be generalized. We now summarize our experience throughout the 16 versions of TrueCrypt. The most important factor is the version of the compiler and of the additional resources required to build the project. We showed that a slight variation in the version number (or minor update) can significantly change the output. Then, we revealed a particular feature in the Microsoft Visual Studio compiler that optimizes the alignment of NOPs according to the recommendations of the manufacturer of the processor on which the compiler is running, making the build processor-dependent. We showed that the project path is sometimes stored in the output and results in cascading variations due to a shifting effect. Moreover, the instructions provided by the authors of the software may not be accurate and lead to incomplete builds as we observed with a bug in the version of NASM advertised in the `readme` file. These instructions may not be precise enough to avoid ambiguities in version numbers as we experienced with Visual Studio or Windows SDK; or, they may be incomplete altogether. In our case, the dd utility was missing. Incomplete code may also be an issue, as we saw with the second version of v5.1a's installer.

Furthermore, embedded timestamps proved to be a very common source of variations, leading to checksum differences. We also showed that the timestamps and checksums may not be included only once in a pre-defined location (as in the PE headers), but also in the resource section of the binary and they may follow application-specific algorithms (as in the case of TrueCrypt's installer). Debug information embeds unique identifiers that are randomly generated during every build. Finally, as embedded digital signatures are impossible to regenerate, they should either be copied as is from the official build after investigation of their content, or reside in a separate file to enable file-based comparison. We emphasize on the fact that recompiling a project twice on the same environment does not necessarily exhibit all sources of non-determinism. One needs to try on different environments with various configurations (e.g., heterogeneous CPUs and other hardware) to see all the differences, if any. Surprisingly, in our analysis, we did not encounter any differences caused with a different operating system (we compiled some versions on both Windows XP and 7).

4. SUMMARY OF CURRENT EFFORTS

In this section, we summarize the efforts from other projects involving deterministic or reproducible builds. Due to space limitations, we provide more details about each of them in an extended version of this work [9].

The most advanced work to date on deterministic builds is probably the one initiated by Bitcoin, named Gitian [17], and later adapted by the Tor project for the Tor Browser Bundle. Gitian provides a virtual environment for Ubuntu in which various sources of non-determinism can be fixed. The solution is however difficult to generalize for other open-source software, and presents several limitations. The adaptation by Tor, although not fully satisfactory, may be a step in the right direction.

Debian focuses on deterministic package build, as presented by Bobbio at DebConf'13 [4] and FOSDEM'14 [5]. Their approach is still experimental and is based on a special branch of the package management tools (dpkg). As of January 26, 2014, 67% of 6887 source packages were found reproducible. As an operating system, Debian can afford changing its own tools for the right purpose. Such liberty may however not exist in heterogeneous environment sharing both open- and closed-source components.

Other Linux distributions have also started their own process towards reproducible/verifiable builds. Fedora only focused on comparing two builds [13], with the underlying assumption that only predictable differences will be present. OpenSUSE also provides build-compare, a script to compare compiled software packages [34]. However, their approach remains preliminary as they do not provide a way to recompile a project for verification. As we observed in our study, it may not be always possible to automate the verification of non-deterministic builds. We note nonetheless that a framework for reconstructing the original environment might help the verification of current and past software.

5. RELATED WORK

Related projects about deterministic and verifiable builds have been discussed in Section 4. We present below other attempts to analyze TrueCrypt, which is the core part of our case study.

The Ubuntu Privacy Remix Team primarily assessed TrueCrypt 7.0a for Linux in 2011, based on previously unpublished review of past versions 4.2a, 6.1a and 6.3a [45]. They created a tool, tcanalyzer,[4] which helps the study of TrueCrypt containers' headers. This tool was used to verify the correctness of both the official build and their own build. They report that they could not find mistakes or backdoors in the encryption or the header format. They further reviewed the cryptographic algorithms in TrueCrypt. They argue about the possibility of hiding a container's cryptographic keys inside the salt value in the TrueCrypt headers in Linux or Windows; or, inside an unused section of the headers in Windows. They advise to recompile the source rather than using the official binaries to prevent such attacks. It is worth noting that in the Linux version, the unused space of the headers is filled with encrypted zeros,

[4] https://www.privacy-cd.org/en/using-upr/download

which can be verified; however, it contains encrypted random data in the Windows version, which is impossible to distinguish from a backdoor version of the keys. Our analysis proves, at least for the Windows version, that the binaries do not differ from the source, and hence do not include such backdoors. The team then presented the discovery of a weakness in the keyfile algorithm, by which it is possible to manipulate any file so that it has no effect when added as a keyfile to encrypt a container.

Sogeti, a French information technology consulting company, reviewed TrueCrypt 6.0a for Windows, Linux and Mac OS X, as part of a first-level security certification for information technologies (CSPN) for the French government in 2008 [38]. CSPN is a security certification formalized by the Central Information Systems Security Division (DCSSI), a government entity under the authority of the General Secretary for French National Defence. This test is to be performed in 30 man-days, and is meant to provide a reference opinion about the security of an application (TrueCrypt here). Sogeti's analysis of TrueCrypt reported that the cryptographic algorithms were implemented correctly, and provided a positive opinion about the application in general, even mentioning that "the product [TrueCrypt] inspires confidence". However, several vulnerabilities were found, regarding a BIOS memory leakage of the password size in case of a system-encrypted partition, memory leakage of the password after the creation of a volume, memory leakage of keyfiles path, memory leakage of the XTS secondary key of a volume after a backup of its header, and a denial-of-service attack against the TrueCrypt driver. Several best practices are also suggested to avoid the identified issues.

In 2013, Amossys proceeded with the same test against TrueCrypt 7.1a for Windows only [2], building up on the previous analysis. They also conclude that the implementation of the main functionalities is correct. They however point out few vulnerabilities that remained unfixed since the previous CSPN test, including the BIOS memory leakage, memory leakage of the last created volume's password, and memory leakage of keyfiles path after a volume is dismounted.

The Open Crypto Audit Project[5] mandated iSEC Research Labs for a security assessment of selected security-sensitive parts of TrueCrypt 7.1a for Windows in 2014 [24]. iSEC identified 11 vulnerabilities, including two integer overflow vulnerabilities, possible leakage of sensitive information from the pagefile, various internal information leakage, and lack of security checks in the bootloader. They also propose corrections. However, no serious flaws were found. Adding our results about the past 16 versions helps to build trust around TrueCrypt.

6. CONCLUSION

As few users compile security-critical open-source software themselves from the source, there should be a way to guarantee that the official build is indeed compiled from the published source. This guarantee would prevent malicious/coerced authors from inserting backdoors in the compiled version only, and would also defend against targeted attacks. It can be offered thanks to a verifiable build that enables reproducing the official build. Verifiable build can be achieved either by a semantic comparison between the official and recompiled files, an approach taken by Fedora and openSUSE; or through a deterministic build process, which can be repeated and would always provide the same output, hence exactly matching the official build. However, sources of non-determinism can be difficult to isolate and reproduce. A perfectly deterministic build can only be achieved if all variables can be controlled. In our case study, we encountered a source of non-determinism based on the brand of the CPU of the building machine, which only showed up after we dug into several versions of the same application, and found no documentation about it. This problem leads to the following conclusion: it is not possible to ensure deterministic builds over time if the build process relies on closed-source software for which an exact documentation is unavailable. Also, through our analysis of 16 versions of TrueCrypt for Windows, we can conclude that verifying old software packages that inherently do not provide a guaranteed deterministic build, can turn into a forensic case in which one needs to gather all the appropriate tools that may have impacted the authors' build, and explain the reasons behind any oddities. In the end, we concluded that the binary files of TrueCrypt for Windows from version 5.0 to and 7.1a match the available source code. Our hope is that the challenges as uncovered through our TrueCrypt case study, and other concurrent projects, would eventually help guide designing future deterministic/verifiable build processes, which are critical for trusting security-critical software.

[5]http://opencryptoaudit.org/

7. ACKNOWLEDGMENTS

We are grateful to anonymous ACSAC2014 reviewers for their suggestions. We also wish to thank the anonymous contributors for providing copies of old Microsoft development kits and pointing few additional details in our analysis. The second author is supported in part by an NSERC Discovery Grant and an FRQNT nouveau chercheur grant.

8. REFERENCES

[1] AMD. Software optimization guide for AMD64 processors, Sept. 2005.

[2] Amossys. Rapport de certification DCSSI-CSPN-2013/09, Oct. 2013. http://www.ssi.gouv.fr/IMG/cspn/anssi-cspn_2013-09fr.pdf.

[3] Bitcoin project. Bitcoin Core, 2014. https://bitcoin.org/en/download.

[4] J. Bobbio. Reproducible builds for Debian. In *DebConf'13*, Vaumarcus, Switzerland, Aug. 2013.

[5] J. Bobbio. Byte-for-byte identical reproducible builds? In *FOSDEM'14*, Brussels, Belgium, Feb. 2014.

[6] Conifer Systems. Build determinism. Blog article (Oct. 17, 2008). http://www.conifersystems.com/2008/10/17/build-determinism/.

[7] Conifer Systems. Machine-independent builds. Blog article (Sept. 15, 2008). http://www.conifersystems.com/2008/09/15/machine-independent-builds/.

[8] X. de Carné de Carnavalet. How I compiled TrueCrypt 7.1a for Win32 and matched the official binaries. Blog article (Oct. 21, 2013). https://madiba.encs.concordia.ca/~x_decarn/truecrypt-binaries-analysis/.

[9] X. de Carné de Carnavalet and M. Mannan. Challenges and implications of verifiable builds for security-critical open-source software. Technical Report 979115, Concordia University, Oct. 2014. http://spectrum.library.concordia.ca/979115/.

[10] Debian Wiki. ReproducibleBuilds. Wiki article visited on May 21, 2014. https://wiki.debian.org/ReproducibleBuilds.

[11] Debian Wiki. SameKernel. Wiki article visited on May 21, 2014. https://wiki.debian.org/SameKernel.

[12] E. Egorochkin. Deterministic (bit-perfect) builds, June 2013. nix-dev mailing list. http://lists.science.uu.nl/pipermail/nix-dev/2013-June/011357.html.

[13] Fedora Project. Reproducible builds for Fedora. https://github.com/kholia/ReproducibleBuilds.

[14] S. Forrest, A. Somayaji, and D. H. Ackley. Building diverse computer systems. In *HotOS'97*, Cape Cod, MA, USA, May 1997.

[15] M. Franz. E unibus pluram: Massive-scale software diversity as a defense mechanism. In *NSPW'10*, Concord, MA, USA, Sept. 2010.

[16] R. Gallagher and G. Greenwald. How the NSA plans to infect 'millions' of computers with malware. News article (Mar. 12, 2014). https://firstlook.org/theintercept/article/2014/03/12/nsa-plans-infect-millions-computers-malware/.

[17] Gitian.org. Gitian: a secure software distribution method, Oct. 2009.

[18] GnuWin project. CoreUtils and gzip for Windows. http://sourceforge.net/projects/gnuwin32/.

[19] J. Gordon. The RSDS pdb format, 2010. http://www.godevtool.com/Other/pdb.htm.

[20] A. Gostev. 'Gadget' in the middle: Flame malware spreading vector identified. Blog article (June 4, 2012). https://securelist.com/blog/incidents/33081/gadget-in-the-middle-flame-malware-spreading-vector-identified-22/.

[21] Hex-Rays.com. Fast library identification and recognition technology, Feb. 2012. https://www.hex-rays.com/products/ida/tech/flirt/index.shtml.

[22] IEEE and The Open Group. dd. The Open Group Base Specifications Issue 7.

[23] Intel. Intel 64 and IA-32 architectures software developer's manual, Feb. 2014.

[24] iSEC. Open Crypto Audit Project - TrueCrypt - Security assessment, Apr. 2014.

[25] S. T. King, J. Tucek, A. Cozzie, C. Grier, W. Jiang, and Y. Zhou. Designing and implementing malicious hardware. In *USENIX LEET'08*, San Francisco, CA, USA, Aug. 2008.

[26] H. Kirsch. The theory of build systems, Sept. 2013. http://www.pifpafpuf.de/BuildTheory.html.

[27] P. Larsen, A. Homescu, S. Brunthaler, and M. Franz. SoK: Automated software diversity. In *IEEE Symposium on Security and Privacy*, San Jose, CA, USA, May 2014.

[28] E. Lippert. Past performance is no guarantee of future results. Blog article (May 31, 2012). http://ericlippert.com/2012/05/31/past-performance-is-no-guarantee-of-future-results/.

[29] J. Menn. Exclusive: NSA infiltrated RSA security more deeply than thought - study. Reuters news article (Mar. 31, 2014). http://www.reuters.com/article/2014/03/31/us-usa-security-nsa-rsa-idUSBREA2U0TY20140331.

[30] Microsoft. Microsoft Portable Executable and Common Object File Format specification v8.3, 2013.

[31] Microsoft. Profile-guided optimizations, 2013. http://msdn.microsoft.com/en-us/library/vstudio/e7k32f4k.aspx.

[32] Microsoft. Specify symbol (.pdb) and source files in the Visual Studio Debugger, 2013. http://msdn.microsoft.com/en-us/library/ms241613.aspx.

[33] Mozilla Developer Network. Building with profile-guided optimization, Aug. 2013. https://developer.mozilla.org/en/docs/Building_with_Profile-Guided_Optimization.

[34] openSUSE Build Service. Build result compare script. https://build.opensuse.org/package/show/openSUSE:Factory/build-compare.

[35] M. Perry. Deterministic builds part one: Cyberwar and global compromise. Tor Project article (Aug. 20, 2013). https://blog.torproject.org/blog/deterministic-builds-part-one-cyberwar-and-global-compromise.

[36] D. Pistelli. Microsoft's Rich signature (undocumented). Blog article (Nov. 11, 2010). http://www.ntcore.com/files/richsign.htm.

[37] PrivacyLover.com. Analysis: Is there a backdoor in Truecrypt? Is Truecrypt a CIA honeypot? Blog article (Aug. 14, 2010). http://www.privacylover.com/.

[38] SOGETI Infrastructure Services. Rapport de certification DCSSI-CSPN-2008/03, Dec. 2008. http://www.ssi.gouv.fr/IMG/cspn/dcssi-cspn_2008-03fr.pdf.

[39] StackOverflow.com. How do I build TrueCrypt on Windows?, Nov. 2012. http://stackoverflow.com/questions/13379644/how-do-i-build-truecrypt-on-windows/13414137#13414137.

[40] The Netwide Assembler bug tracking system. NASM crashes when building x64 .obj file. Bug ticket (Oct. 6, 2009). http://sourceforge.net/p/nasm/bugs/469/.

[41] The PaX Team. Address space layout randomization, Mar. 2003. http://pax.grsecurity.net/docs/aslr.txt.

[42] K. Thompson. Reflections on trusting trust. *Commun. ACM*, 27(8):761–763, Aug. 1984.

[43] Tor Project bug track system. Improve software assurance. Bug ticket (May 31, 2012). https://trac.torproject.org/projects/tor/ticket/6008.

[44] TrueCrypt Foundation. TrueCrypt.

[45] Ubuntu Privacy Remix Team. Security analysis of TrueCrypt 7.0a with an attack on the keyfile algorithm. Technical report (Aug. 14, 2011). https://www.privacy-cd.org/downloads/truecrypt_7.0a-analysis-en.pdf.

[46] M. Uecker. Building packages three times in a row. Debian mailing list. https://lists.debian.org/debian-devel/2007/09/msg00746.html.

[47] J. Walton. An analysis of the Windows PE checksum algorithm, Mar. 2008. http://www.codeproject.com/Articles/19326/An-Analysis-of-the-Windows-PE-Checksum-Algorithm.

[48] D. Wendlandt, D. G. Andersen, and A. Perrig. Perspectives: Improving SSH-style host authentication with multi-path probing. In *USENIX Annual Technical Conference*, Boston, MA, USA, June 2008.

[49] D. A. Wheeler. *Fully Countering Trusting Trust through Diverse Double-Compiling*. PhD thesis, George Mason University, Oct. 2009.

Cloud Radar: Near Real-Time Detection of Security Failures in Dynamic Virtualized Infrastructures

Sören Bleikertz Carsten Vogel
IBM Research - Zurich
{sbl,ten}@zurich.ibm.com

Thomas Groß
University of Newcastle upon Tyne
thomas.gross@newcastle.ac.uk

ABSTRACT

Cloud infrastructures are designed to share physical resources among many different tenants while ensuring overall security and tenant isolation. The complexity of dynamically changing and growing cloud environments, as well as insider attacks, can lead to misconfigurations that ultimately result in security failures. The detection of these misconfigurations and subsequent failures is a crucial challenge for cloud providers—an insurmountable challenge without tools.

We establish an automated security analysis of dynamic virtualized infrastructures that detects misconfigurations and security failures in near real-time. The key is a systematic, differential approach that detects changes in the infrastructure and uses those changes to update its analysis, rather than performing one from scratch. Our system, called *Cloud Radar*, monitors virtualized infrastructures for changes, updates a graph model representation of the infrastructure, and also maintains a dynamic information flow graph to determine isolation properties. Whereas existing research in this area performs analyses on static snapshots of such infrastructures, our change-based approach yields significant performance improvements as demonstrated with our prototype for VMware environments.

1. INTRODUCTION

Infrastructure clouds are rapidly and dynamically changing systems due to self-service provisioning and on-demand scalability. Tenant as well as provider administrators frequently adapt the configuration of the sub-system they control, constituting in dynamic changes for the entire configuration. These changes may create security vulnerabilities with respect to the tenants' individual or the provider's overall security policies. According to studies by ENISA [5] as well as CSA [4], isolation failures as well as operational complexity leading to misconfiguration and security failures are among the top threats in cloud computing. Those vulnerabilities can be introduced as a non-deliberate fault or be the deliberate act of an insider attacker, affecting all the pillars of infrastructure clouds: computing, networking, and storage. While misconfiguration of network components (e.g., subnets and VLAN IDs) are recognised as the faults of isolation breaches, unwanted co-location of computing resources [11] or misconfigured storage isolation have been observed, too. While configurations of multi-tenant infrastructure clouds are complex in themselves, overseeing the security consequences of many configuration changes by multiple administrators can easily be beyond the grasp of human operators.

Indeed, the configuration complexity we observe in dynamically changing infrastructure clouds calls for tool-support. Existing research in this space is mostly focused on dynamic infrastructure analysis of non-security properties [14], node integrity monitoring [13] or establishing security analyses of static systems given by a configuration snapshot [2]. While the latter results give us confidence about reasoning on security consequences of infrastructure cloud topology and configurations, they suffer from blind spots due to transient security failures as well as from efficiency problems. In fact, an isolation case-study [3] using this approach showed that the analysis of a mid-sized virtualized infrastructure required about *seven minutes* for extracting the configuration and building up a model and *one minute* on the actual analysis of the model. Performing such an analysis in a dynamic environment will lead to a backlog of changes that need to be analysed, an increase in the response times in case of security incidents, as well as scalability and efficiency problems.

Consequently, it is our primary goal to reduce the times for configuration extraction, model building and analysis by establishing a systematic *differential* approach that does not require a full extraction and analysis on each configuration change, but still maintains strong security foundations all the way. We realise this goal with a practical security system that uses a model-based security analysis. It maintains a graph representation synchronised with the actual configuration of the virtualized infrastructure and accepts change events produced by cloud management hosts to update its own model. The model and its updates form the foundation for a differential security analysis that maintains an information flow graph for analysing isolation properties and which tries to find violations of specified security policies.

Our Contributions. With the overall research goal to establish a differential security analysis of dynamic infrastructure clouds, we make the following contributions: **1)** We establish an architecture that caters for near to real-time detection of configuration changes in heterogeneous virtualized infrastructures. **2)** In order to maintain a synchronised graph model of these infrastructures, we propose a set of algo-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ACSAC'14, December 08–12, 2014, New Orleans, LA, USA.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-3005-3/14/12 ...$15.00.
http://dx.doi.org/10.1145/2664243.2664274.

rithms for the computation of graph deltas (added/removed nodes and edges, changed node attributes) applicable to a graph model based on change events. **3)** We propose a novel approach that soundly maintains an information flow graph for the dynamic graph model of the infrastructure. **4)** We offer a practical implementation of our system, called *Cloud Radar* (*CR*), for VMware environments. Our comprehensive evaluation shows that the differential approach reduces the overall analysis time significantly, putting near-to-real-time analysis in our reach. For a broad spectrum of cloud operations and even for large infrastructures, we measure model update times in the order of milliseconds, which renders our approach several orders of magnitude more efficient than previous static analysis approaches. **5)** We establish a security analysis showing that *Cloud Radar* can be set up as security monitoring of insider adversaries.

2. SYSTEM AND SECURITY MODEL

An infrastructure cloud consists of (virtualized) computing, networking and storage resources, which are configured through a management host and its well-defined interface.

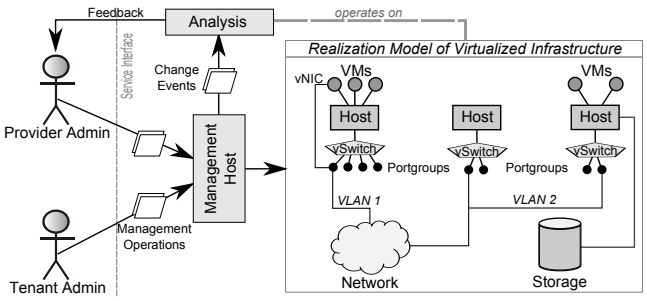


Figure 1: System model of the differential security monitoring covering compute, network, and storage.

As shown in Figure 1, the system model of this work is poised towards a differential analysis based on change events issued by the management hosts when the infrastructure is re-configured. The analysis system uses these change events to continuously update a graph representation of the infrastructure, the *Realization* model, which is used for subsequent analysis. As long as the management host issues the events correctly, the model covers malicious adversaries, insiders and externals alike.

2.1 System Model

We represent the virtualized infrastructure in a graph model, called *Realization* model (cf. [3]): The model is an undirected, vertex typed and attributed graph. The vertices of the graph represent the components of the virtualized infrastructure, which may be entire sub-systems, such as physical servers or virtual machines, or low-level components, such as virtualized network interfaces. Vertices are typed, e.g., type vm denotes a virtual machine, and annotated with name/value attributes. The attributes encode detailed properties of the components and capture their configuration. The edges of the graph represent the connections and relationships among components of the virtualized infrastructure, therefore encoding its topology. The vertex types of our model are organised in a hierarchy graph, i.e., a directed acyclic graph (DAG) where the edges represent a parent-to-child relation. The hierarchy graph reflects the inherent hierarchy found in the infrastructure. For example, a virtual machine belongs to a physical host, and therefore a physical host has a directed edge to a virtual machine.

Considering the example from Figure 1, we see that the Realization model captures all areas of the virtualized infrastructure: computing, networking and storage. While the actual model encodes fine-grained components of all these areas, e.g., storage being represented as virtual disks, file backend objects and storage pools, we focus our explanation on the networking components to prepare the ground for examples in subsequent sections. Physical hosts and their hypervisors provide networking to VMs by virtual switches that connect the VMs to the network. A virtual switch contains virtual ports, to which the VMs are connected via a virtual network card (vNIC). Virtual ports are aggregated into *port groups*, which apply a common configuration to a group of virtual ports. Virtual LANs (VLANs) allow a logical separation of network traffic between VMs by assigning distinct VLAN IDs to the associated port groups.

2.2 Threat Model

We establish a threat model based on the dependability taxonomy [1]. Users and administrators can be malicious or non-malicious. Thereby, we cover all classes of human-made faults, independent from *intent* or *capability*, that is, faults can be introduced deliberately as result of a harmful decision or without awareness; faults can be introduced by accident or by incompetence. These fault classes include misconfigurations as well as malicious insider administrators and, thereby, constitute a strong adversary model. Agents that operate on behalf of a human, e.g., as part of cloud automation, are also covered by this threat model, because we do not differentiate between the issuers of changes.

We only place one constraint on how the adversary can exert threats upon the virtualized infrastructure: The adversary is bound to the well-defined cloud manager API and cannot subvert the communication channel between the management hosts and the analysis system. In §5 we discuss and assess multiple deployment approaches to realise such a constraint in a practical environment. For example, based on isolation of the monitoring and management networks from the administrators, as well as using mandatory access control. Note that we consider the security of the software for the management host and the hypervisors as out of scope.

3. DESIGN AND IMPLEMENTATION

In this section, we describe the design and implementation of our *Cloud Radar* system. The goal of our system is to detect – in near real-time – configuration changes that impact the security of virtualized infrastructures. On a high-level, the system works in the following way.

The start of the system's workflow is an initial snapshot of the configuration and topology of the entire virtualized infrastructure represented as a graph model. An initial information flow analysis determines how information may flow within the infrastructure, in order to determine isolation properties. Isolation is critical in multi-tenant virtualized infrastructure and the concern of many security policies. The crucial part of our approach is that we operate on *change events*, which are the result of a change in the infrastructure, and which needs to be represented in the model by transforming it according to the event. The transformation of the model may result in new or changed information flow in the system, and the information flow analysis is differentially up-

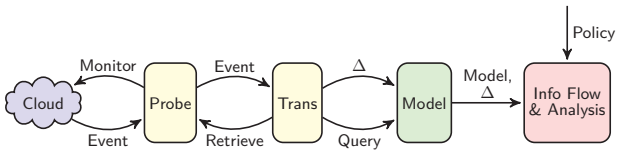


Figure 2: System Architecture and Components.

dated based on the change event. Finally, after each change the resulting model will be analysed with regard to a given security policy or a set of policies.

We depict the system architecture and components that implement such an analysis workflow in Figure 2. The system is composed of the following components. A **Probe** knows how to monitor the virtualized infrastructure and how to obtain change events. It is tied to a specific virtualization technology, e.g. VMware. For heterogeneous environments multiple probes are instantiated. For each probe a translation component (**Trans**) exists that knows how to convert from change events into model transformations, i.e., a graph delta (Δ). Depending on the information richness of the events, further information may need to be retrieved from either the probe or queried from the existing model.

The **Model** component contains the graph model of the virtualized infrastructure as introduced in §2.1. By obtaining graph deltas (Δ) from the *Trans* component, the model is updated in accordance to the change event. A graph delta contains the nodes and edges that should be added or removed, as well as attribute changes for nodes. As a pre-processor for the analysis, the **Info Flow** component determines the information flow implications in the infrastructure and updates the graph model with information flow edges. In a differential analysis, the information flow is updated based on the graph delta. The **Analysis** component analyses the infrastructure given as the graph model with regard to a security policy, which expresses desired or undesired properties of the infrastructure topology or configuration.

3.1 Obtaining Infrastructure Change Events

We follow a similar architecture as presented in [3] where a set of *probes* extracts the configuration of different virtualized systems. Instead of periodically extracting the entire configuration, we extend and improve the existing approach with probes that obtain events of changes in the infrastructure.

The format and level of information of change events can largely vary between different virtualization technologies. For example, VMware and Xen provide rich events on changes in their inventories, Libvirt provides change events on a VM level, and OpenStack as a management platform only provides coarse-grained events. In our design, we cater for the variety of formats and information richness of events among different virtualization technologies. In this paper, we focus on the VMware event probe.

3.1.1 VMware Probe

VMware maintains an internal relational inventory of the virtualized infrastructure that is composed out of *Managed Entities*, such as virtual machines or physical hosts. Managed entities can have properties that describe further configuration aspects of that entity. Each entity can be addressed using a *Managed Object Reference (MOR)*.

We are using the method WaitForUpdates of the VMware API [16] to obtain notifications on updates and property changes for managed entities. This method is part of the *Property Collector* component, which also handles retrieval of properties of entities in the API. The method returns an UpdateSet object that contains an incremental version number, which is used in repeated calls to only obtain the latest changes. Further, the update contains a set of ObjectUpdate objects with an *Object* attribute stating the *MOR* of the updated object, as well as a *Kind* attribute to indicate the kind of update. The update kind can be i) *Enter* for a new object, ii) *Leave* for a removed object, and iii) *Modify* for property changes of that object.

Essentially, the updates state which objects have been added or removed from the inventory, and which have been modified. For new or modified objects, a set of PropertyChanges describe the property changes of the objects in the following form. **Operation:** The type of property change. It can be i) *Add* a value to a collection property, ii) *Remove* a value from a collection, or iii) *Assign* a value to the property. **Name:** The name of the property that is changed. **Value:** Only for Add and Assign, the value that is added or assigned to the property.

Consider the operation UpdatePortGroup that allows an administrator to change the virtual networking configuration for virtual machines, including changes to the network isolation property. In the case that an administrator changes the VLAN ID associated with a port group *PG-1* to a new value of *123*, we obtain the following event from VMware.

```
[modify] HostSystem (host-159)
  assign: config.network.portgroup["key-vim.host.↩
      PortGroup-PG-1"].spec.vlanId <- 123
```

The event indicates that the host object host-159 has been modified. In particular, as part of the network configuration of that host, the property spec.vlanId of the port group *PG-1* has been assigned the value *123*.

3.2 From Change Events to Model Updates

From a high-level perspective, the *Trans* component translates from a change event to a model update in the form of a graph delta, as illustrated in Figure 2. The translation has to differentiate between three kind of change events. First, a new object appeared that may result in new nodes and edges in the graph model. Second, an object was removed and the corresponding nodes and edges in the model have to be removed, too. Finally, an object has been modified, i.e., attributes of that object have changed. This may result in attribute changes of nodes in the model too, but it can also leads to the creation or deletion of nodes and/or edges in the model. This categorisation aligns well with the events produced by the VMware probe since they contain an attribute indicating the kind of change.

A translation is typically bound to a specific probe and its produced event format, i.e., a VMware translation knows how to translate VMware change events. Therefore, we focus in the following on describing the event translation design with the concrete example of translating VMware change events. The translation needs to handle the three different change events, but also has to deal with the ordering of events due to their dependencies, and with incomplete new objects. Therefore, the output of the translation is either a graph delta, dependency requirements, or a notification that the translation encountered an incomplete, ignored, or unsupported object. If an change event consists of multiple object updates, we merge the produced graph deltas to form a single graph delta for that change event.

3.2.1 Translation of an Object Update

We explain and propose a set of algorithms for the successful translation of an Object Update into a graph delta. The handling of a failed translation due to cases such as incomplete objects or dependency ordering will be discussed in §3.2.3 and §3.2.4 respectively.

We define a graph delta as $\Delta = (V^+, V^-, E^+, E^-, M)$, where we differentiate between *creator* nodes and edges (V^+, E^+), *eraser* nodes and edges (V^-, E^-), and a set M of node attribute modifier in the form of $(node, attribute, value)$. Creators lead to new elements in the graph, erasers remove existing elements from the graph, and node attribute modifiers change attributes of existing nodes to new values.

Enter Object: Creating New Nodes and Edges.

We obtain an *Enter* Object Update for a new object that has been created in the inventory as well as for all the existing objects in the inventory during the initial probe connection. The goal of the translation component is to produce new nodes and edges for the model based on the update.

The fundamental idea is to employ a recursive algorithm that starts at a newly created object and traverses through all its connected neighbour objects. For each object, the technology-specific parts of the translation creates a corresponding model node and populates its attributes with values of the object. It further establishes relations to other created nodes due to the recursion. The output of this algorithm is a set of newly created model nodes and edges, where the edges not only connect to new nodes, but also to existing nodes in the model.

The translation of object updates has to handle two corner cases: Incomplete objects, where the attributes of an object have not been populated fully yet, and the ordering of object updates within the same update set. We will describe the handling of these cases in §3.2.3 and §3.2.4, respectively.

Leave Object: Deleting Nodes and Edges.

For each object that has been removed from the inventory, we obtain an *Leave* Object Update. The update contains the MOR of the removed object, and we lookup the corresponding model node identifier and obtain the node by querying the model component. Since in our model a managed entity might have resulted in the creation of multiple nodes, we have to perform a recursive deletion of the dependent nodes of the removed node.

The recursive deletion works as the following. First, given an object that was removed from the inventory, we lookup the corresponding node in the model, and add the node to the eraser node set. For all the deleted node's neighbours, we place the connecting edges in the eraser edge set. Further, we continue the recursive deletion at the neighbour node if i) the neighbour's type is a child type in the hierarchy (cf. §2.1); and ii) the neighbour node is not a managed entity.

Modify Object: Creating an Entire Graph Delta.

Finally, we consider the case that an object has been modified. A *Modify* Object Update consists of a Property Change and the modified object reference. This property change indicates the type of change, the attribute that has changed, and potentially a new value.

Our algorithm consumes such a property change and produces a graph delta that consists of creator/eraser nodes and edges, as well as a set of attribute modifier in the form of $(object, attribute, value)$. The algorithm has a similar structure as the algorithms for creating or removing objects, and in fact builds upon them. For each object type, we further differentiate between the changed attribute, as well as the operation performed on that attribute. For attribute assignments, we construct attribute modifiers that change the corresponding node's attribute. For added managed entities or data objects, we rely on the algorithm that handles *Enter* objects. Similarly, we construct erasers for deleted objects based on the *Leave* object algorithm.

An example to illustrate a modified object is the creation of a new virtual device, such as a virtual ethernet adapter, for a VM. In this case, we have to translate the new virtual device into a new model node, and connect it to the existing VM node with an edge. Such an event produces the following creator nodes and edges: $V^+ = \{vnic, vport\}$ and $E^+ = \{(vm, vnic), (vnic, vport), (pg, vport)\}$, where vm corresponds to the existing VM, $vnic$ and $vport$ are created, and the virtual port is connected to the port group pg.

3.2.2 Applying the Model Update

Given a graph delta as produced by our set of algorithms based on a change event, updating the graph model is expressed as updating the node and edge sets of a given graph $G = (V, E)$: $V' = (V \setminus V^-) \cup V^+$ and $E' = (E \setminus E^-) \cup E^+$. The updated model graph is $G' = (V', E')$. For each node attribute modifier $(node, attribute, value)$ in M we change the attribute of the node in V' with the new value.

3.2.3 Postponing Incomplete Objects

Resolving further information of an object during the translation may fail when not all relevant attributes have been set yet. For example in VMware, when a VM is created its configuration is only later fully populated. In that case, we are dealing with an incomplete object. We maintain a set of incomplete objects and monitor updates for these objects. If an incomplete object receives an update, we try to handle it as a new object rather than a modified one. If the translation succeeds, i.e., the object was complete and could be translated, we remove the object from the incomplete set. Otherwise, it remains in the incomplete object set. In practice and during our evaluation, incomplete objects always received modify events when further attributes were populated, usually within a sub-second time span. We may also employ periodic translation attempts for incomplete objects, in case they receive no further modify events.

3.2.4 Ordering of Updates based on Dependencies

We are dealing with an asynchronous system and we have to take care about the ordering of the change events. However, two aspects of the VMware probe supports the ordering of events. First, the probe connects over TCP which provides packet ordering for us. Second, VMware employs version numbers for the event discovery, which provides an ordering of events between different versions. However, within one UpdateSet, we may encounter a wrong ordering of changes. We can order the changes by using a dependency graph, i.e., a directed acyclic graph (DAG) where vertices are changes and directed edges describe dependencies such that the source node fulfils the dependency of the target node.

We construct such a dependency graph in the following way. A successful translation of an event returns a graph

Table 1: Subset of Information Flow Rules Relevant for Portgroup VLAN Isolation.

#	Type	Flow	Node Pair	Condition(s)	Edge Dependency
1	Simple	noflow	$VSwitch \Leftrightarrow PortGroup$	Portgroup's VLAN ID != 0	Attribute VLAN ID
2	Complex	flow	$PortGroup \Leftrightarrow PortGroup$	Portgroups' VLAN IDs equal and their virtual switches connected	Attribute VLAN ID, Connectivity of VSwitches
3	Simple	flow	$Any \Leftrightarrow Any$	None	

delta of new or modified nodes. In the case of an unsuccessful translation due to a missing dependency, i.e., a node was not found in the current model, the translation returns a *requirement* in the form of a node type and a predicate on its attributes. Further, the translation of an event may return *potential* nodes, which become available once other requirements are fulfilled. Based on the translation attempts, we try to match new or modified nodes with requirements, and introduce a directed dependency edge from the fulfilling event to the requirement. Potential nodes may also satisfy requirements, thereby building up a dependency graph. A topological sort of the dependency graph will yield an evaluation order.

3.3 Differential Information Flow Analysis

The information flow analysis determines how information may flow in the virtualized infrastructure by computing an information flow graph. The graph forms the foundation for analysing isolation properties in the infrastructure. The analysis works in two phases: first, it takes the realization model graph that represents the infrastructure and computes an overlay directed information flow graph. Second, it computes for the information flow graph the strongly connected components (SCC), i.e., the sets of graph nodes that are mutually reachable, and constructs a reachability graph of the SCCs. Our analysis is inspired by [3], but has been extended and improved to be more efficient and operate in a differential way, i.e., it operates on changes of the realization model rather than computing the information flow from scratch after each change. In order to compute the SCCs, we employ Tarjan's algorithm [15], but variants also exists that operate in a differential way [12].

3.3.1 Specifying Flow and Trust Assumptions

The core of the information flow analysis is a set of *traversal rules* that specify which elements in the infrastructure are trusted and provide isolation, and which elements constitute to information flows. For example in VMware, port groups provide isolation if they have been configured with a VLAN ID different than zero. The traversal rules for this example are listed in Table 1. A first-matching application of traversal rules results in new information flow edges, which either describe a *flow* or *noflow* (isolation) between a pair of nodes. Further, an information flow edge can be dependent on a specific node's attribute, the connectivity of nodes, as well as a combination of both. We differentiate between *simple* rules, which determine flows between an adjacent pair of nodes, and *complex* rules, which work on a pair of nodes that are not necessarily adjacent, but that fulfil a common condition such as equality of an attribute.

In our example, a simple rule introduces a *noflow* edge between a virtual switch and a adjacent port group in case the VLAN ID is not zero. The edge is attribute-dependent on the port group's VLAN ID attribute. Further, a complex rule introduces *flow* edges between non-adjacent port groups

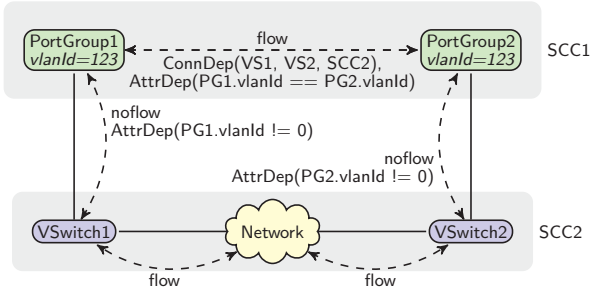


Figure 3: Graph model annotated with dashed information flow edges of different kinds (simple, attribute-dependent, connectivity dependent).

of the same VLAN ID (attribute dependency) if the virtual switches of the port groups are connected (connectivity dependency). A default rule introduces flow between any node pair that has not been covered by a previous traversal rule. Figure 3 illustrates the resulting topology graph that is annotated with information flow edges, as well as grouping nodes into SCCs, i.e., SCC1 and SCC2. In the case of the connectivity-dependent flow edge between the port groups, we see that the virtual switches are recorded as *connectivity endpoints* and SCC2 is part of the *connectivity path*.

3.3.2 Maintaining an Information Flow Graph

The challenge we solve is to maintain an information flow graph, which is build from simple as well as attribute and/or connectivity-dependent information flow edges, even when connectivity or attributes change. Our differential analysis works in two phases: 1) Given a graph delta (cf. §3.2.1), we compute the insertion and deletion of nodes and edges from the information flow graph by identifying affected graph elements and selectively apply traversal rules; 2) Based on the previous insertion and deletion of information flow edges we compute the reachability graph of strongly connected components. In case of changes in the connectivity, we also introduce or remove connectivity-dependent edges.

Processing Realization Model Graph Deltas.

In the first phase we process the realization model graph delta, and for each of the following elements of the graph delta compute information flow graph changes.

Creator Nodes: We insert the new nodes in the information flow graph. For each new node, evaluate the complex traversal rules, which may create new information flow edges, and insert the created edges in the information flow graph.

Creator Edges: We evaluate the simple rules for each node pair of the new edges and insert the resulting information flow edges in the graph.

Eraser Nodes: We remove the nodes as well as all their incoming and outgoing edges from the information flow graph. Further, we find all connectivity-dependent edges that require connectivity of a removed node, and remove those edges too. For example if VSwitch is removed from Fig. 3, the edges to

PortGroup1 and Network are removed. Additionally, the edge between the port groups is removed, because it is dependent on the connectivity of the vswitches.

Eraser Edges: For each undirected removed realization model edge, we find the directed information flow edges and remove them from the information flow graph.

Node Attribute Changes: Finally, we find all affected attribute-dependent edges and remove them if their attribute condition does not hold anymore. The simple rules are re-evaluated for the invalid edges and the complex rules are evaluated for the changed nodes. The resulting information flow edges are inserted in the graph. For example, if PortGroup1's VLAN ID changes to zero, the edges between the vswitch as well as the other port group are removed, but a new flow edge is introduced between the vswitch.

Processing Connectivity Changes.

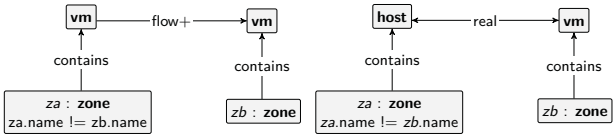
In the second phase of our analysis, we compute a new SCC reachability graph when information flow edges have been inserted or deleted in the first phase. We detect changes in connectivity by comparing the previous with the new SCC reachability graph. In particular, we operate on new and removed SCCs as well as inter-SCC edges, and either create or remove connectivity-dependent information flow edges.

Removed SCCs or inter-SCC edges: In the case of *reduced* connectivity, we find all connectivity edges that contain a removed SCC or removed inter-SCC edge in their connectivity path, and remove such edges. For example, if the Network node is removed from the example of Fig. 3, SCC2 splits into two new SCCs, i.e., SCC2 is removed and two new SCCs are added. In the example, the connectivity-dependent edge between the port groups is affected and removed, because SCC2 appears in its connectivity path. Since another connectivity path could exists for the connectivity endpoints, we re-evaluate the complex rules for the removed edges' node pairs. In the example no other connectivity path exists.

New SCCs or inter-SCC edges: In the case of *increased* connectivity, we re-evaluate connectivity candidates, i.e., information flow edges that previously have been missing connectivity, if they are affected by the new SCCs or new inter-SCC edges. If their connectivity is now fulfilled, we add them to the information flow graph.

3.4 Specification of Security Policies and Detection of Policy Violations

For the detection of security failures, we define the following security policies, in the form of *attack states*, with their graphical representation shown in Figure 4. *Cloud Radar* tries to match the policies on the dynamic realization model and information flow graph. Once a policy's attack state matches, we have found a security failure. A set of security administrators is notified about a security violation, in order to mitigate the problem.



(a) Network Isolation **(b) VM Placement**

Figure 4: Graphical Representation of Network and Compute Security Policies.

Network Isolation: Virtual machines are grouped into "security zones", e.g., production and test zone, and these zones must be isolated on the network level, e.g., through different virtual networks. This policy is violated if we find a potential connection (flow+) between two VMs of different security zones (za and zb).

VM Placement: A group of virtual machines should run on one or multiple designated physical hosts, e.g., for performance, availability, or also data privacy reasons. This policy is violated if a VM runs on a different host than the ones designated. Preventing VM co-location, e.g., due to side-channel attacks [11], is a variant of this policy.

Storage Isolation: VMs of different security zones must not be able to exchange information over a shared storage device, e.g., by using the same file as backing of the VMs' virtual disks.

We have two implementations to find a policy violation by matching the policy's attack state against the current realization model and information flow graph. The first one is a native implementation in Java/Scala that iterates through the nodes in the model graph. It benefits from a fast execution time and uses the SCC reachability graph to efficiently determine if two model nodes are connected. Either the two nodes are in the same SCC or there exists a path between their corresponding SCCs in the reachability graph. Otherwise, they are not connected. However, implementing new policies requires a native implementation, which makes it less extensible by end-users, such as a cloud administrator.

The second implementation uses a general-purpose graph matching tool called Groove [6], which tries to match a given sub-graph in a larger graph. In fact the policies in Fig. 4 are valid sub-graphs that Groove can match against our realization model graph. The main benefit of this approach is its extensibility, since end-users can implement new policies in a graphical and intuitive way. However, as a general purpose tool it bears a higher execution overhead and for determining connectivity it uses an equivalent but less efficient path-finding algorithm, compared to the SCC approach.

4. PERFORMANCE EVALUATION

In this section we empirically evaluate and discuss the performance of *Cloud Radar* in the case-study of a semi-production environment as well as in simulated environments of different sizes. The performance evaluation focuses on the processes of building and maintaining the models in sync with changes in the infrastructure.

4.1 Methodology and Environments

Our evaluation is performed with different environments: a real, semi-production environment (R_{150}) with 2 hosts and 150 VMs, and a simulated environment that uses an infrastructure simulator incorporated in the VMware management hosts. We vary the size of the simulated environment ($S_{\#VMs}$) between 150 and 30,000 VMs with a host-VM ratio of 1 : 50. This allows us to evaluate the scalability of our approach. *Cloud Radar* itself runs in a Linux VM with 12 vCPUs, 12 GB RAM, and Java 1.7.

We differentiate between the two approaches of obtaining and maintaining the model of a virtualized infrastructure: *Static Snapshot* is the existing approach that always extracts the full configuration. In order to deal with a dynamic and

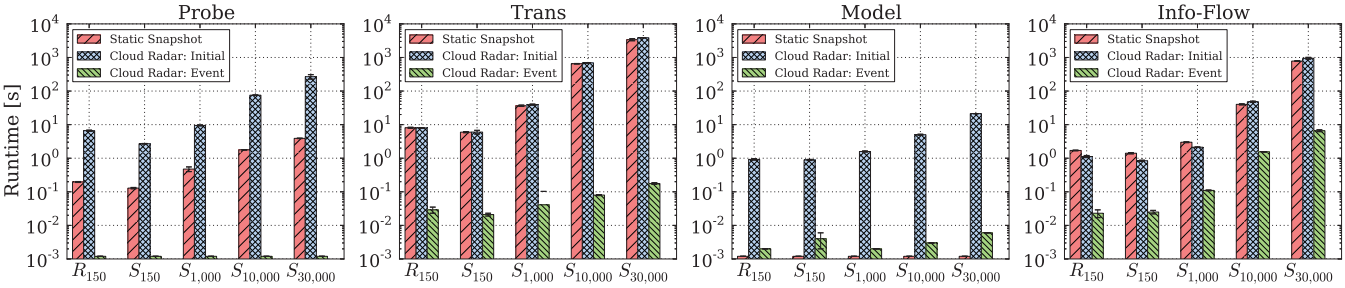


Figure 5: Runtime measurements (in seconds, log scale) of the existing approach "*Static Snapshot*" and the new *Cloud Radar* approach ("*Init*" for initialisation and "*Event*" for change events) for the four system components in relation to the infrastructure size (number of VMs).

constantly changing infrastructure, the extraction has to be executed periodically. On the other hand, *Cloud Radar* obtains an initial event containing the full configuration of the infrastructure, followed by events for infrastructure changes. In order to compare the performance of the two different modes, we measure the runtime of the Probe, the time needed to translate the Probe output into a model (sub)-graph, to initialise or update the graph model, as well as to construct or maintain the info flow graph. For the new event-based approach, we measure these aspects for both the initial event and subsequent change events. In order to trigger change events, we automatically perform a variety of operations on the virtualized infrastructure. In the particular measurement of Fig. 5 we used the CreateVM operation.

4.2 Results and Discussion

Figure 5 illustrates the main results of our performance evaluation with the runtime in seconds on the logarithmic y-axis, and the different environments and sizes in terms of number of VMs on the x-axis. We break down the results into measurements for the four system components. The measurements for the existing approach are shown as the *Static Snapshot* bars, and the new approach is broken down into measurements for initialisation (*Cloud Radar: Init*) and events (*Cloud Radar: Event*). Our measurement resolution is $1ms$ and measurements such as the runtime of the probe for events is equal or below that resolution.

How does the new event-based approach compare to the existing full extraction approach? How does the system scale with the size of the virtualized infrastructure?

First of all, comparing the results of the 150 VM sized realistic (R_{150}) and simulated (S_{150}) environments show equivalent results, which indicates that the infrastructure simulator in fact behaves accurately and provides a suitable environment for performing our measurements. Of course, in a realistic environment our absolute measurements could differ, but the scalability of the system would remain equivalent.

Probe: We observe that both approaches initially scale linearly, although the runtime of the existing approach is lower compared to the initialisation phase of our new one. We suspect this behaviour to be rooted in the more complex construction of the probe output. The existing approach traverses the VMware inventory and obtains a list of all the managed entities. In contrary, the new approach sets up a filter and VMware is required to find all entities that match the filter and have not been reported previously. However, after the costly initialisation, the probe reports events instantly and below our measurement resolution of $1ms$, independently of the infrastructure size.

Trans: This is the dominating factor in the initialisation of the model, in particular for large-scale infrastructures. Both the existing approach as well as our new approach perform almost identical for initialising the model, and scale linearly with the size of the infrastructure. This is unsurprisingly, as both the existing approach as well as the new approach with *Enter* events perform a similar translation of creating new objects. The significant performance improvements lies in translating change events into updates of the model with our new approach. To highlight this, consider the $S_{30,000}$ environment with 30,000 VMs: After the comparable initialisation time by both approaches, the existing approach would require a periodic translation of the entire environment taking *56 minutes*, whereas in the new approach each change event can be translated in *176 milliseconds* (the worst-case we measured for our set of operations). This is an improvement of four orders of magnitude.

Model: Both the model initialisation of the existing approach as well as the model update based on events are almost instantly. In the first case, a new full model is constructed all the time and can override the existing one, i.e., a simple reference assignment. In the latter case for the operations we tested, the graph delta remains small and is merged into the existing graph model. For the initial large event in our new approach, we have to merge a set of graph deltas together, where the size depends on the infrastructure size, resulting in a linear scalability. This also indicates the worst-case scenario, in case an operation results in an event that changes the entire infrastructure.

Info Flow: If we break down the information flow analysis (cf. Table 2), we observe for the full analysis a linear scalable evaluation of simple traversal rules and SCC computation. We also see a quadratic complexity for evaluating the complex traversal rule, which needs to evaluate pairs of port groups in our example rule set (cf. Table 1). The differential approach provides significant improvements with a rules evaluation that only depends on the size of the event and a linear SCC computation, for which we see even further potential for optimisation.

Analysis: We measured a runtime of $19ms$ for finding violations of the network isolation policy in the real environment. This includes finding all violations of the policy, although one could terminate after the first violation. The VMware infrastructure simulator does not support operations that trigger such policy violation, therefore our performance measurement is limited to the real environment.

In summary and in the light of the more expensive initialisation of the new approach, when does it actually pay off? Consider the 10,000 VM environment $S_{10,000}$ and the

Table 2: Breakdown of Info Flow Runtime (in ms) into Simple/Complex Traversal Rule and SCC Computation for Static- and Event-based Approaches.

	Simple		Complex		SCC	
	Static	*Event*	*Static*	*Event*	*Static*	*Event*
R_{150}	153	4	151	1	153	48
S_{150}	157	4	90	2	139	52
$S_{1,000}$	269	4	592	2	465	138
$S_{10,000}$	755	6	31,172	2	2,776	1,095
$S_{30,000}$	1,681	4	926,708	2	11,346	4,769

cumulative runtimes of both approaches. The initialisation in the existing approach overall takes $693s \pm 14$ and for the new approach $819s \pm 20$. While follow up model updates require a full periodic execution of the entire workflow in the existing approach, the new one only requires $1.8s$ for each change event. Although the new approach is slightly more expensive in the initialisation, even after two event it pays off due to the much more efficient event processing.

How many events can be processed per second until we run into a backlog? Considering the simulated 10,000 VM environment ($S_{10,000}$), we can observe and translate approximately *33* VM creation operations per minute, bounded by the dominating translation time of $1.8s$ per CreateVM operation and assuming a serialised processing.

5. SECURITY EVALUATION

We evaluate the security of *Cloud Radar* in two ways. First, a security analysis argues that all change events are received with integrity, in face of the given adversary model. Further, we discuss various approaches how our system can be deployed securely in practice. Second, we test the system's ability to detect policy violations for compute, network, and storage resources using randomised operations.

5.1 Security Analysis

The security analysis considers the management host creating events and the *CR* host as separate entities and considers multiple attack vectors including manipulating network communication through VLAN re-configuration or denial of service or dropping communication sessions in the Session Manager. We propose a practical deployment of *Cloud Radar* that is secure in the face of an insider adversary, based on a small set of assumptions and a deployment pattern, which includes isolation of the reporting network, a heartbeat signal, and mandatory access control for regular administrators.

5.1.1 Assumptions and Deployment Pattern

The threat model of §2 already introduces that software attacks are out of scope, which includes that the management host software cannot be manipulated by the adversary. Our analysis is built on the following explicit assumptions, which form the backdrop of the deployment pattern.

[secchan] TLS offers a secure channel providing channel confidentiality and integrity, with server authentication based on a dedicated PKI. The adversary does not have capabilities to establish host certificates in the certificate tree of the root CA $\mathsf{CA}_{\mathsf{sec}}$ trusted by CR.

[access] The adversary accesses the virtualized infrastructure through the management interface only. This implies that the adversary does neither have physical or root access on the physical hosts, direct access to the hypervisor nor physical access to network and storage. The adversary does not have access as super_admin.

The assumption [access] is motivated by vSphere Security [17] best practice, which states that hypervisor hosts should only be managed through the central management host. This can also be enforced by putting the hypervisor into *lockdown mode*, by which no other users than vpxuser, the vCenter management user, have authentication privileges nor can perform operations on the host directly.

Network Isolation: We need to establish the condition [netisolation] that the reporting network (between the management host and Cloud Radar) is isolated from the networks accessible by the adversary to protect the event channel from interference. A dedicated reporting network $\mathsf{net}_{\mathsf{sec}}$ is created for the event reporting between management host and *Cloud Radar*. The network isolation is enforced 1) as dedicated physical networks (building upon the assumption [access]), 2) with a VLAN in the physical switch, where hypervisor or virtualization administrators do not have privileges, or 3) as a virtual network with a dedicated VLAN ID, where the administrators do not have privileges to change the VLAN configuration. The event channel is established as a secure channel ([secchan]) to the management host via $\mathsf{net}_{\mathsf{sec}}$.

Heartbeat Signal: The condition [heartbeat] models the realisation of a heartbeat signal sent in time intervals t_{hb}. A heartbeat can be realised by 1) opening the CR probe filter to background noise events, such as machine utilisation, including them into the event stream, 2) a periodic task changing managed entities scheduled by the super_admin, or 3) CR exercising write access on the managed entities, e.g., VMs, to obtain change events directly. It is necessary that the heartbeat signal will be in the event channel observed by the CR probe. Whereas the first approach is least invasive and does not require write privileges, it may yield false positives. The two other approaches give a reliable heartbeat signal, yet require partial write access, either under control of the super_admin or *Cloud Radar* itself.

Mandatory Access Control: The super_admin sets privileges such that regular administrators only gain privileges on the management host, but not on the hypervisors according to [access]. The following privileges are set on the management host: **1)** No administrator has rights to revoke a lockdown mode of a host. **2)** No administrator has rights to manipulate $\mathsf{net}_{\mathsf{sec}}$. **3)** The administrator privileges for session manipulation on Sessions are restricted, in particular Sessions.TerminateSession is controlled.

5.1.2 Security Argument

The foundation of *Cloud Radar* (*CR*) to detect security failures is the ability of obtaining all change events of the infrastructure in an unmodified form. Therefore, we establish the requirements integrity and availability, and argue that our secure deployment of *CR* fulfils these requirements.

Integrity.

For any n-th event e_n received at *CR* holds that the event is correct, fresh, in order and as it has been sent by the management host.

According to §3.1.1, if *CR* requests version n, the management host is guaranteed to produce an event e_n, which contains all changes after e_{n-1} up to reception of the request for version number n. Thereby, the event chain is complete. We obtain the order and weak freshness properties from the

version number, as an event e_n must have been generated after any event $e_{<n}$.

The event e_n is received at *CR* over a secure channel according to condition [netisolation]. The channel is established over the dedicated network $\mathsf{net_{sec}}$ and server-authenticated on cert that is in the certificate chain of trusted $\mathsf{CA_{sec}}$, which is inaccessible to the adversary according to [secchan]. Thereby, the connection is with the correct management host. Further, based on the secure channel of [secchan], we obtain channel confidentiality and integrity on the event e_n, which is thereby as sent by the management host. As the adversary can neither interfere with the management host event reporting by the exclusion of software attacks nor with the network configuration for $\mathsf{net_{sec}}$, the event e_n is the correct event sent as intended by the management host. From [access], we obtain that the adversary could not have changed the event at the management host or any subordinate host.

Weak Availability.

Either all events sent by the management host are received by *CR* eventually and latest within a channel timeout t_{timeout} or an alarm is raised after t_{timeout} is elapsed.

The network $\mathsf{net_{sec}}$ is modelled as an asynchronous channel, through which messages arrive eventually, the secure channel is established over it by *CR*. Observe that even though underlying TCP/IP offers reliable, ordered and error-corrected communication, it does not give strong timeliness guarantees. Whereas the secure channel enforces integrity and ordering, it does not offer availability. Because of the in-order delivery of $\mathsf{net_{sec}}$, it follows that if e_n is received, then all previous events $e_{<n}$ must have been received already, yielding that, if the channel is intact, all events sent by the management host are received by *CR* eventually and latest within a set timeout t_{timeout}. The condition [netisolation] isolates the network $\mathsf{net_{sec}}$ from interference by the adversary on the network, while [access] prevents interference on the subordinate hosts, however this does not rule out availability failures from other root sources, e.g., a cable fault.

The Weak Availability clause, i.e., an alarm is raised after t_{timeout} is elapsed, is obtained from the condition [heartbeat]. If the channel waits for a packet or the channel is interrupted, then we have that eventually t_{timeout} will be reached without a packet having arrived at *CR*. According to [heartbeat], the management host produces a heartbeat signal after each time window $t_{\mathsf{hb}} < t_{\mathsf{timeout}}$. Therefore, we have that the after t_{timeout} without a message, *CR* can conclude that the channel is interrupted and raise an alarm. It follows that availability failures are detected within t_{timeout}. The system will try to reestablish the connection after an interrupt.

5.2 Security Testing

For each policy (cf. §3.4) we determine the operation that may cause a policy violation if used with a specific parameter. We execute these operations several hundred times with a parameter from a known set of violating parameters or a random parameter, similar to *Fuzzing* from software security testing. *Cloud Radar* is required to detect a policy violation in the case of a parameter from the violating set, and otherwise no violation should be detected.

In the case of network isolation, a critical operation is UpdatePortGroup that changes the VLAN identifier of a port group to a given one. If the new VLAN identifier is conflicting with an existing identifier of a different tenant or security zone, the policy is violated. A violating VLAN identifier was chosen with a probability of 1/3. For VM placement, the critical operation is CreateVM that creates a new VM on a given host. The policy is violated if the given host is not part of the same placement zone as the new VM. Finally, storage isolation is violated if a VM is reconfigured (ReconfigVM) with a virtual disk that uses as backend a file already in use by another VM of a different zone.

The security testing uses the real environment described in §4.1, because the simulated one does not support all management operations and its networking configuration is not suited for the network isolation policy. This is not problematic as analysis performance and scalability are not a concern in this security testing, and a real environment yields more realistic behaviour as a simulated environment.

For the network isolation policy, *Cloud Radar* in fact detected all expected violating operations as policy violations, and operations with random operations as non-violations. Overall we issued 254 violating operations and 746 non-violating ones. For the VM placement policy, the system exhibits correct behaviour by detecting 491 violating VM creation operations and reported no violations for 509 non-critical operations. Finally, 505 operations out of 777 VM storage operations have been correctly identified as violations, and for the others the tool correctly reported no violations.

6. RELATED WORK

Configuration Changes in Networks: Misconfigurations in networks have been a problem in the operation of IT environments for a long time and solutions to mitigate such misconfiguration by the means of change monitoring and analysis have been proposed. Mahajan et al. [10] studied misconfigurations in BGP routing configuration changes by listening to changes and assess these. Kim et al. [9] analysed the evolution of network configurations by mining a repository of network configuration files. With the rise of software-defined networking, real-time monitoring and policy checking have been achieved in these environments [7, 8]. In virtualized infrastructures, misconfiguration may not only happen to the network configuration, but span the entire field of compute, network, and storage resources, which we tackle with *Cloud Radar*.

Security of Static Virtualized Infrastructures: The modelling and security analysis of *static* virtualized infrastructures have been subject of existing works [2, 3]. However, these efforts lack the ability to handle dynamic behaviour of such environments. With *Cloud Radar* we are closing this gap by introducing the continuous monitoring of dynamic infrastructures and maintaining an up-to-date model. In fact, our performance evaluation showed significant performance improvements compared to the existing approach.

Monitoring of Dynamic Virtualized Infrastructures: Closest to our work is *vQuery* [14] which monitors configuration changes in VMware environments and assess these changes with regard to performance implications. The different goals and motivations of vQuery and *Cloud Radar* (performance versus security) are reflected in the model and configuration translation, where vQuery models many performance metrics and *Cloud Radar* focuses on capturing the topology and its security. Schiffman et al. [13] proposed a monitoring system called *Cloud Verifier* that allows to monitor hosts and virtual machines with regard to integrity requirements, e.g., based on trusted computing mechanisms.

Cloud Radar on the other hand focuses on topological properties of the virtualized infrastructure and allows a wide variety of infrastructure security policies to be analysed.

7. CONCLUSIONS AND FUTURE WORK

In this paper we presented *Cloud Radar*, a system that detects security failures in virtualized infrastructures in near real-time. The system monitors virtualized infrastructures for changes and based on these changes maintains a graph model of the infrastructure. The model is the input to a model-based security analysis on the infrastructure's topology. The analysis computes and maintains an information flow graph for the dynamic infrastructure, in order to determine isolation properties, and tries find violations of specified security policies. We implemented a prototype of *Cloud Radar* for VMware environments and our performance evaluation shows a significant performance improvement of our event-based approach compared to an existing one that uses static configuration snapshots. The snapshot approach requires $693s$ in a 10,000 VM simulated infrastructure for extracting the configuration and building the models, whereas our event-based one only requires $1.8s$ for each change event after an initialisation of $819s$.

As future work, we aim for further optimisations of the differential information flow analysis by implementing a dynamic SCC computation algorithm. We perceive interesting opportunities along the lines of correlating events with operations and the historical analysis over the evolution of the model. Further, we also plan to extend the model to cope with dynamic access control configurations.

Acknowledgments

This research has been partially supported by the TREsPASS project funded by the European Union's Seventh Framework Programme (FP7/2007-2013) under grant agreement number ICT-318003.

Linux is a registered trademark of Linus Torvalds in the United States, other countries, or both. Java is a registered trademark of Oracle and/or its affiliates. Other product and service names might be trademarks of IBM or other companies.

8. REFERENCES

[1] Avizienis, A., Laprie, J.-C., Randell, B., and Landwehr, C. Basic Concepts and Taxonomy of Dependable and Secure Computing. *IEEE Transactions on Dependable and Secure Computing 1*, 1 (jan.-march 2004), 11 – 33.

[2] Bleikertz, S., Gross, T., and Mödersheim, S. Automated Verification of Virtualized Infrastructures. In *ACM Cloud Computing Security Workshop (CCSW'11)* (Oct 2011), ACM.

[3] Bleikertz, S., Gross, T., Schunter, M., and Eriksson, K. Automated Information Flow Analysis of Virtualized Infrastructures. In *16th European Symposium on Research in Computer Security (ESORICS'11)* (Sep 2011), Springer.

[4] CSA. The Notorious Nine: Cloud Computing Top Threats in 2013. Tech. rep., Cloud Security Alliance (CSA), feb 2013.

[5] ENISA. Cloud computing: Benefits, risks and recommendations for information security. Tech. rep., European Network and Information Security Agency (ENISA), nov 2009.

[6] Ghamarian, A. H., de, M. M., Rensink, A., Zambon, E., and Zimakova, M. Modelling and analysis using GROOVE. *International Journal on Software Tools for Technology Transfer (STTT)* (March 2011).

[7] Kazemian, P., Chang, M., Zeng, H., Varghese, G., McKeown, N., and Whyte, S. Real Time Network Policy Checking Using Header Space Analysis. In *10th USENIX Symposium on Networked Systems Design and Implementation* (2013), USENIX, pp. 99–111.

[8] Khurshid, A., Zou, X., Zhou, W., Caesar, M., and Godfrey, P. B. VeriFlow: Verifying Network-Wide Invariants in Real Time. In *10th USENIX Symposium on Networked Systems Design and Implementation* (2013), USENIX, pp. 15–27.

[9] Kim, H., Benson, T., Akella, A., and Feamster, N. The Evolution of Network Configuration: A Tale of Two Campuses. In *Proceedings of the 2011 ACM SIGCOMM Conference on Internet Measurement Conference* (2011), IMC '11, ACM, pp. 499–514.

[10] Mahajan, R., Wetherall, D., and Anderson, T. Understanding BGP Misconfiguration. In *Proceedings of the 2002 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications* (2002), SIGCOMM '02, ACM, pp. 3–16.

[11] 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.

[12] Roditty, L., and Zwick, U. A Fully Dynamic Reachability Algorithm for Directed Graphs with an Almost Linear Update Time. In *Proceedings of the Thirty-sixth Annual ACM Symposium on Theory of Computing* (2004), STOC '04, ACM, pp. 184–191.

[13] Schiffman, J., Sun, Y., Vijayakumar, H., and Jaeger, T. Cloud Verifier: Verifiable Auditing Service for IaaS Clouds. In *Proceedings of the IEEE 1st International Workshop on Cloud Security Auditing (CSA 2013)* (June 2013).

[14] Shafer, I., Gylfason, S., and Ganger, G. R. vQuery: a Platform for Connecting Configuration and Performance. *VMware Technical Journal 1*, 2 (Dec. 2012).

[15] Tarjan, R. Depth-First Search and Linear Graph Algorithms. *SIAM Journal on Computing* (1972).

[16] VMware. vSphere 5.5 API Reference, Sep 2013. `http://pubs.vmware.com/vsphere-55/index.jsp#com.vmware.wssdk.apiref.doc/right-pane.html`.

[17] VMware. vSphere Security, ESXi 5.5, vCenter Server 5.5 (EN-001164-04), 2013.

Scippa: System-Centric IPC Provenance on Android

Michael Backes, Sven Bugiel, Sebastian Gerling
{backes,bugiel,sgerling}@cs.uni-saarland.de

Saarland University/CISPA, Germany

Abstract

Google's Android OS provides a lightweight IPC mechanism called Binder, which enables the development of feature-rich apps that seamlessly integrate services and data of other apps. Whenever apps can act both as service consumers and service providers, it is inevitable that the IPC mechanism provides message receivers with message provenance information to establish trust. However, the Android OS currently fails in providing sufficient provenance information, which has led to a number of attacks.

We present an extension to the Android IPC mechanism, called Scippa, that establishes IPC call-chains across application processes. Scippa provides provenance information required to effectively prevent recent attacks such as confused deputy attacks. Our solution constitutes a system-centric approach that extends the Binder kernel module and Android's message handlers. Scippa integrates seamlessly into the system architecture and our evaluation shows a performance overhead of only 2.23% on Android OS v4.2.2.

1. INTRODUCTION

Smartphone operating systems allow end-user customization of the phone's functionality with 3rd party apps. To make this extensibility possible and to simultaneously protect the end-user's privacy, current designs of smartphone operating systems exhibit a complex combination of sandboxing-based privilege-separation and extensive message-based data-sharing. The popular Android OS—the focus of this paper—facilitates the integration of remote services and data into an app using a very lightweight IPC mechanism called *Binder*, which forms the primary channel for inter-app communication. To realize privilege-separation between apps, Android implements app sandboxing by assigning each app a distinct user ID under which the app's processes are executed. To implement the least-privilege principle, *Permissions*, i.e., privileges, are assigned to UIDs. Android ships with a set of pre-defined permissions to protect the Android application framework API, for example, reading the user's address book or retrieving the device's geolocation. It further allows app developers to define custom permissions to protect their apps' interfaces. Protecting an app interface with permissions is realized by 1) statically declaring in a manifest file which permissions are required from a caller to successfully access each app component; or by 2) performing runtime checks in the app components using IPC provenance information provided by Binder, i.e., the calling process' UID.

However, Android's current solution to protecting app interfaces is unsatisfactory. Statically declaring the required (custom) permissions for interacting with app components reduces the scope of access checks to the permissions each app holds and excludes more app-specific information such as the developer ID or package name. This solution is not scalable and makes it virtually impossible to flexibly endorse specific apps for component access: permissions can be either requested by any app or require apps to be signed with the same key, which in turn would require a more flexible public key infrastructure that does not exist. Performing runtime checks to protect app components, on the other hand, is more flexible and allows more fine-grained access control, but requires that the IPC mechanism provides message provenance information to app components. Android's system model does *not* fulfill this requirement for sufficient IPC provenance information for all app component types. While it provides the caller UID to *ContentProvider* and *Service* components, it fails in providing this information to components that are receivers of Intent messages–the most prominent inter-app communication mechanism. Prior work [22, 2, 3, 7, 4] has identified different attacks that can occur as a result of this shortcoming, most prominently *confused deputy attacks* [11].

Contributions. In this paper, we identify the technical root-cause for this shortcoming in providing IPC provenance information—a mismatch between Android's concept for inter-application communication at its middleware layer and at its kernel layer. In this multi-layered communication framework, the Binder kernel module is responsible for providing the sending process' user ID to the receiving process. However, *logical* communication occurs between app components—in the literature commonly referred to as *Inter Component Communication* (ICC) [7, 8]—using different abstraction levels of Binder IPC at Android's middleware layer. These abstractions introduce indirections and message dispatching that cause Binder's IPC provenance information (i.e., the sender UID) to be lost along the ICC control flow between app components.

We then present Scippa, our extension to Android's inter-application communication framework, to remedy this short-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.
ACSAC'14, December 08–12, 2014, New Orleans, LA, USA
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-3005-6/14/12 ...$15.00.
http://dx.doi.org/10.1145/2664243.2664264.

coming of Android's architecture. SCIPPA builds Binder IPC call-chains for ICC control flows and thus provides the required provenance information to apps. Although Quire [4] first identified the need for provenance information on Android, SCIPPA is, to the best of our knowledge, the first approach that *directly* addresses the mismatch between the middleware and the kernel-level security design in Android's multi-layered inter-application communication framework.

At the core of SCIPPA is an extension to the Binder kernel module, which constructs and forwards IPC call-chains across distinct application processes. The kernel module extension is complemented by extensions to the message handling routines in Android's application libraries to propagate call-chains across all components of an app. In contrast to Quire's prototypical implementation, which requires app developers to explicitly pass on call-chain information during ICC, our extensions integrate seamlessly into the system architecture and call-chains are established transparently to apps and app developers. Only when developers want to retrieve call-chains, they must be aware of a new system API.

SCIPPA enables for the first time determining the ICC caller ID within all types of Android app components. For instance, apps can now identify the sender of received broadcast Intents and thus distinguish spurious notifications from benign ones; they can also detect if a security-sensitive Activity was invoked from a trustworthy caller. This allows apps in general a more fine-grained, flexible self-governing of their interaction with other apps and provides the means to effectively mitigate recently reported attacks such as confused deputy attacks [11, 22, 3]. Additionally, we present and discuss changes to Android's app model to enable the return of finalized call-chains to the sending app. Providing information about how their messages were distributed, both by the system and other apps, gives senders the means to *detect* spurious or malignant distribution of their messages (e.g., message hijacking [2]).

The evaluation of our prototype implementation for Android v4.2.2 shows that the performance overhead imposed on Binder IPC messages is only 2.23% and thus does not impede the overall system performance.

2. BINDER-BASED INTER-APP COMMUNICATION ON ANDROID

We first provide the necessary technical background for SCIPPA. We introduce the Android OS and describe how it uses Binder-based IPC for different types of inter-app communication. In particular, we explain how Binder transactions are integrated into Android's security design.

2.1 Primer on Android

Android is an open-source software stack for embedded devices. The lowest level of this stack consists of a slightly modified Linux kernel that is responsible for basic services such as memory management, device drivers, or inter-process communication (IPC). On top of the Linux kernel lies the extensive Android middleware: it consists of native libraries (e.g., SSL), the Android runtime with the Dalvik Virtual Machine, and the application framework. The middleware implements the majority of Android's application API, which is complemented by pre-installed system apps at the application layer. The API can be extended with 3rd party apps on top of the software stack.

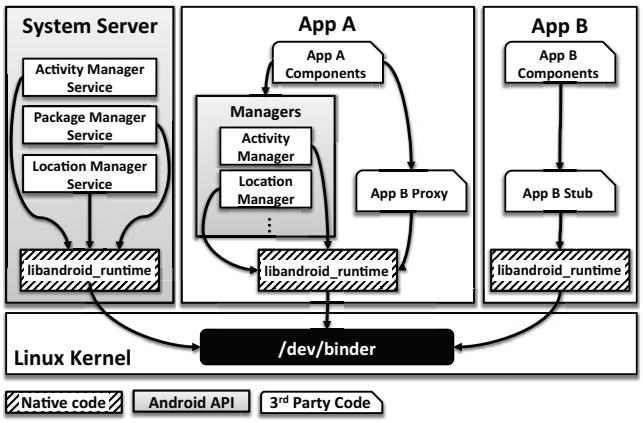


Figure 1: Binder-based inter-component communication.

Android apps are composed of different components and each app is sandboxed by executing it in a separate process with a distinct user ID (*UID*) and assigning it a private data directory on the filesystem. The four basic app components are *Activities* (GUI for user interaction), *BroadcastReceivers* (mailbox for broadcast *Intent* messages), *ContentProviders* (SQL-like data management), and *Services* (long term operations without user interaction). All components can be interconnected remotely across process boundaries by using different abstractions of Binder IPC. These interconnections are commonly referred to as *Inter-Component Communication (ICC) [8]*. To achieve privilege separation between app and realize the least-privilege principle, Android introduces *Permissions*, i.e., privileges that an app must have been granted by the user at install-time to access security- and privacy-sensitive resources.

2.2 Binder-based ICC

Although Android builds on top of a Linux kernel that provides *"classical"* channels such as files or sockets, the primary IPC mechanism on Android is *Binder*. In the following we take a top-down approach to ICC in Android. We show how Android uses ICC and explain afterwards how ICC is implemented as Binder transactions. We refer to external documentation [25] for more details on Binder.

2.2.1 Using Binder-Based ICC on Android

Figure 1 provides a high-level overview of standard Binder IPC in Android when used for connecting components of different apps. Apps can, for instance, either contact system services such as the *Location Manager Service* or communicate directly with each other. All Inter-Component Communication (ICC) builds on top of Binder IPC. User space processes can communicate with each other over Binder IPC via the Binder kernel module that is exposed through the `/dev/binder` sysfs entry. For inter-component communication, the *libandroid_runtime* library, included in all apps, includes an implementation of the Binder communication protocol. Moreover, since application developers usually do not want to deal directly with the low-level mechanics of inter-process communication, Android's design provides different levels of abstraction for Binder IPC. These allow developers to easily make use of Binder IPC at the application level to connect different apps' components (cf. Figure 1 for STUBS, PROXIES and MANAGERS).

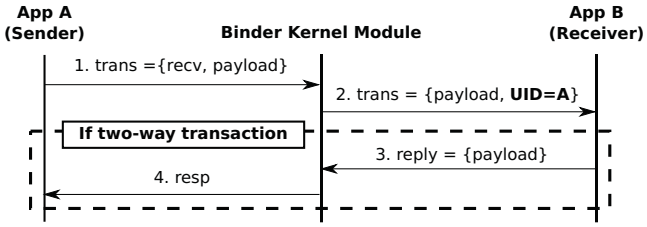


Figure 2: Binder transaction protocol.

Stubs and Proxies. The most basic level of abstraction of Binder IPC are Stubs and Proxies, which implement remote procedure calls (RPC) via Binder IPC. A Proxy at the caller-side marshals the method parameters into primitive data types and transfers them via IPC to the recipient, where Stub unmarshals the primitives into the original parameters and calls the actual method.

System Services and Managers. Managers are part of the SDK and encapsulate pre-compiled Proxies for system apps and services like the Location Manager Service that implement the Android application framework API.

Intents. The highest level of abstraction are so-called *Intent* messages. An Intent is a data structure used to provide an abstract description of an operation to be performed by its receiver(s). Common usages of Intents include starting Activity components or broadcasting notifications to apps. Since the sender of an Intent can both explicitly state the target component and implicitly define potential receivers through a description of the intended action, the actual target app(s) must be resolved at runtime. This is the task of the *ActivityManagerService*, which relays all Intents.

2.2.2 Binder Transactions and Integration Into Android's Security Design

Before we explain how Binder acts as a building block in Android's security architecture, we first explain how Binder conducts transactions between two app processes. Figure 2 illustrates abstractly a transaction between *App A* (sender) and *App B* (receiver). To initiate the transaction, *A* writes its transaction data via `/dev/binder` to the Binder kernel module (step 1). The transaction data contains a token (`recv`) identifying the communication peer (i.e., *B*) as well as some payload containing, e.g., the method ID to be executed by the receiver plus the method arguments (e.g., an Intent object). The kernel module then resolves the token to identify the recipient of the transaction, i.e., *B*, and extends the transaction data with the sender UID, i.e., UID of *A*. Afterwards, the module copies the transaction data into the user space of an *IPC Thread* selected from the IPC thread pool of *B* (step 2). If the caller expected a reply (*two-way transaction*), the reply is sent back to the caller via the kernel module (steps 3 and 4). A two-way transaction is implemented as a *closed wait*, i.e., the sender thread blocks until it receives a response and the kernel module ensures that this response originates from the receiver thread.

Providing the sender UID to the receiving component is pivotal for enforcing permissions in Android's security design. First, system services and apps, which implement the application framework API, use this information to perform runtime checks (i.e., `PackageManager.checkPermission(permission, uid)`) whether calling apps hold the required permissions to access their interfaces. Using runtime checks enables these services to enforce permissions flexible and at the granularity of interface functions instead of app components. For instance, the Location Manager Service requires either the Permission `FINE_LOCATION` or `COARSE_LOCATION` depending on the parameters passed to the API call. Second, system services like the *Activity Manager Service* mediate between caller and callee applications (e.g., when an app queries a remote ContentProvider or when delivering an Intent) and these system services use the caller UID when mediating to check whether the caller is allowed to access the callee.

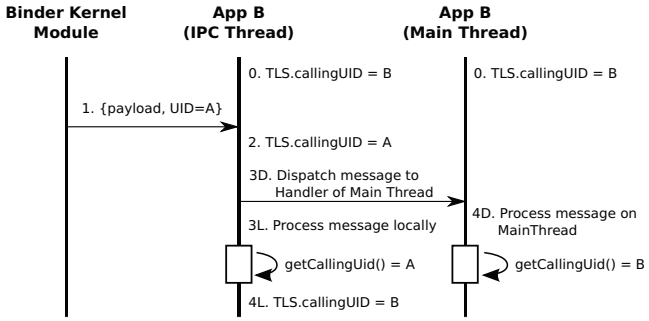


Figure 3: Handling of Binder transactions by the receiver and availability of caller UID.

3. PROBLEM DESCRIPTION

As explained in Section 2.2.2, Binder provides IPC message recipients with the UID of their *direct* caller. However, as shown in Section 2.2.1, Android introduced different abstraction layers for Binder IPC to enable an *inter-component* communication on top of the inter-process communication. These abstractions introduce *indirections* and *message dispatches* that cause the caller UID provided by Binder to be lost along ICC control flows or to be insufficient.

3.1 Message Dispatching

To understand how the caller UID provided by Binder can be dropped in ICC, one first needs to examine how an incoming IPC transaction is handled at the receiver side. Figure 3 extends Figure 2 after the receiver (*App B*) has received the transaction (Step 2 in Figure 2 and Step 1 in Figure 3). The IPC thread of *App B* that is selected to handle this incoming transaction copies the sender UID of the transaction (i.e., *UID=A*) into its thread-local storage (TLS; step 2). Any application code executed on this thread can query this sender UID through the `Binder.getCallingUid` function. However, if the thread is *not* processing a Binder transaction such that the TLS gets never updated, this function call defaults to the threads own information (e.g., `Binder.getCallingUid` would return the UID of the thread itself, as set in step 0).

A received transaction can be processed in one of two possible ways and depends on the targeted component type: First, the message could be handled locally in the context of the IPC Thread (step 3L), which preserves the IPC provenance information. A *Service* or *ContentProvider* component, for instance, would be executed by default in this context. As a consequence, these components can call `Binder.getCallingUid` at any time and retrieve the process UID that triggered their current execution. As explained in Section 2.2.2, this is pivotal for enforcing the default permissions in Android's

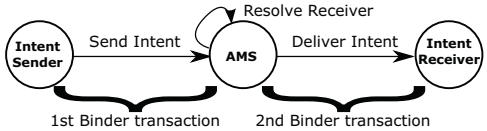


Figure 4: Indirection in Intent-based ICC.

system services. Once the execution of app code on the IPC thread has finished, this thread resets its TLS (step 4L).

Alternatively, the IPC Thread can dispatch the processing of a received message to a worker thread (by default the application's *Main Thread*[1]). The message is dispatched by means of Android's *Handler* mechanism[2] (step 3D). For instance, this worker thread typically performs the handling of received Intents, which includes the processing of received Broadcast Intents, executing IntentServices, as well as Activity-related Intents (step 4D). However, the worker thread is executed in a different context and naturally with a different TLS. Thus, calling `Binder.getCallingUid` on the worker thread will always result in the retrieval of the worker thread's process UID. Dispatching the message will, therefore, effectively drop IPC provenance information: Components executed on the application's worker thread will have no information about which process has triggered their current execution and, hence, cannot distinguish whether their execution is legitimate and whether they can trust any received payload.

3.2 Indirect Communication

Android's model for ICC also introduced indirect communication between components, which renders Binder's IPC provenance information insufficient, in particular in the case of Intent-based ICC. As mentioned in Section 2.2.1 and illustrated in more detail in Figure 4, the *ActivityManagerService* is responsible for relaying Intents between apps. Thus, the actual communication between Intent senders and Intent receivers consists of two distinct Binder transactions. As a consequence, Binder's IPC provenance mechanism will at the receiver's side always identify the *ActivityManagerService* as the IPC caller, instead of the actual origin of the received Intent (i.e., the UID of the Intent sender). An exception from this shortcoming are Activity-related Intents that require a return value from the receiver. In that specific case, the Intent receiver can request the sender ID from the *ActivityManagerService*.

3.3 Provenance Information vs. Permissions

Many attacks [22, 7, 3] that have been reported in the Android security literature can be effectively mitigated if the callee is provided with comprehensive IPC provenance information. In rare cases [7], the cause for the discovered vulnerability was simply a forgotten permission check. In general, however, the situation is more complex. The confused deputy attacks presented in [22] relied on a privileged *BroadcastReceiver* that modified the system state (e.g., Wi-Fi or GPS state) on behalf of any broadcast sender. The receiver did not check whether the sender was entitled to send this command, since the current Android design does not provide the technical means that enable the *BroadcastReceiver* to retrieve the caller's UID and to endorse the caller for this privileged command. As a potential fix, a new permission could be introduced to protect this receiver from receiving Intents from unprivileged apps. However, this approach would be very inflexible: it would require a new permission for every distinct privileged receiver, especially when a receiver holds multiple privileges (e.g., Wi-Fi *and* GPS) or the exact privileged operation depends further on message parameters. We assume these to be the reasons why the vulnerabilities mentioned above are still not fixed in Android v4.2.2, although they have been known since v2.2 [22]. In contrast, provisioning comprehensive IPC provenance information as in Scippa provides the means for a flexible and fine-grained access control to (system) app developers and is, thus, preferable to effectively prevent attacks, such as confused deputy attacks [22, 3].

[1]Also referred to as *UI Thread* or *Activity Thread*.

[2]`http://developer.android.com/reference/android/os/Handler.html`

4. REQUIREMENTS ANALYSIS

In this Section, we define our adversary model, derive requirements and discuss challenges for a comprehensive Binder IPC call provenance on Android.

4.1 Adversary Model

The attacker model for our design of Scippa considers a strong attacker that is able to mount confused deputy, Intent hijacking and Intent spoofing attacks.

Confused Deputy Attacks. We adopt the confused deputy attacker model [11] that was adapted to the specific scenario of Android [4, 22]. In this model, a malicious app with an insufficient set of permissions for its malign purpose tricks a privileged app into executing its privileges on behalf of the malicious app. For instance, Enck *et al.* [7] reported the possibility of sending an Intent to the *Phone* app in order to start a phone call without holding the corresponding `CALL_PHONE` permission. Porter Felt *et al.* [22] discovered several *BroadcastReceiver* components in system services that acted as confused deputies and allowed any app to change, for example, the Wi-Fi or GPS status.

Intent Hijacking and Spoofing. In addition to confused deputy attacks, we consider Intent hijacking and Intent spoofing attacks [2]. To mount an Intent hijacking attack, the attacker registers an app in the system such that (ordered) broadcast Intents or certain Activity-related Intents are delivered to this registered app instead of to the actually intended recipient. Thus, the malicious app is able to receive any information contained in the Broadcast. Additionally, in the context of Activities, the attacker can use this technique to mount phishing attacks.

Restrictions. We focus in this work exclusively on *Binder IPC* and exclude other IPC channels such as files or sockets. A solution for these alternate channels would require additional extensions to low-level services such as the filesystem. Moreover, we limit our solution to *direct* IPC and do not consider covert channels as leveraged in collusion attacks [24, 16]. Finally, we do not explicitly consider attacks that compromise the system integrity such as root exploits or attacks against system components. However, we consider 3rd party apps to be in full control of their sandbox. Apps can include native code that is able rewrite the application code.

4.2 Requirements and Challenges

In this section we derive the necessary requirements for Scippa and elaborate on technical and conceptual challenges in context of the Android system model.

Availability of Provenance Information. Due to message dispatching, provenance information is currently only available to app components that are executed in the context of a receiving IPC thread. Thus, the first requirement for a comprehensive solution is to extend Android's app model to propagate IPC provenance information to all app components. A particular challenge to be addressed in this context is to identify the correct IPC context of each thread. The Main Thread, for instance, usually handles workloads dispatched by multiple IPC threads. Hence, its current IPC context depends on the IPC thread it is currently serving.

Building System-Centric IPC Call-chains. By default, Binder provides apps with the UID of their direct caller. However, when considering the indirections in Android's Intent-based ICC, this information is insufficient for callees to identify the initiator of incoming requests. This leads to the second requirement: it is necessary to establish system-centric call-chains for Binder IPC so that we can provide receivers of Binder transactions with provenance information. This would enable receivers to answer questions like *"Who sent this Intent?"*.

Returning Call-Chains to Senders. While Android provides the receiver of a Binder transaction with limited means to retrieve the sender's UID, it does not provide any feedback to the sender on how their message was handled in the system. This missing feedback makes the senders unaware of, e.g., Intent hijacking and phishing attacks [2]. The third requirement is, therefore, to establish a feedback mechanism for IPC senders by returning already established, finalized call-chains to them. This enables the senders to analyze how their message was handled both by the system and other applications and, thus, to *detect* potential hijacking and phishing attacks. A technical challenge is to efficiently address branching of call-chains, which leads to a *1:N* communication (e.g., when broadcasting an Intent).

Tagging Asynchronous Messages. Although Binder transactions are synchronous, the protocols and mechanisms Android deploys on top of Binder can be asynchronous. For instance, *sticky* Broadcast Intents are kept in the system and are delivered even to recipients that register *after* the broadcast was sent. Thus, to effectively fulfill the first three requirements, this asynchronicity needs to be addressed, e.g., by tainting asynchronously delivered messages with their associated IPC provenance information.

5. SYSTEM-CENTRIC IPC CALL-CHAINS

In this section, we describe our extensions to Android's Binder IPC at the kernel and user space to establish call-chains along *direct* ICC control flows. Further, we elaborate on extensions to Android's message handling mechanism to propagate those call-chains between the threads of an app.

5.1 Establishing Call-Chains

At the core of our solution are extensions to the Binder kernel module. In Binder, IPC messages are passed between processes as Binder transactions. Our extensions construct call-chains across app processes by linking the transactions along a direct thread of control for inter-application communication. Figure 5 illustrates recursive Binder transactions between three apps *A*, *B*, and *C*. For instance, *App A* could be an Intent sender, *App B* the *ActivityManagerService*, and *App C* the Intent receiver (cf. Figure 4). Moreover, Figure 5 shows that we differentiate in our design between *two-way* and *one-way* transactions. In recursive two-way transactions the second transaction `trans#2` is nested in the first transaction `trans#1`. In contrast, in one-way transactions `trans#1` is finished before `trans#2` is triggered, as illustrated by the pseudo reply `*reply#1`.

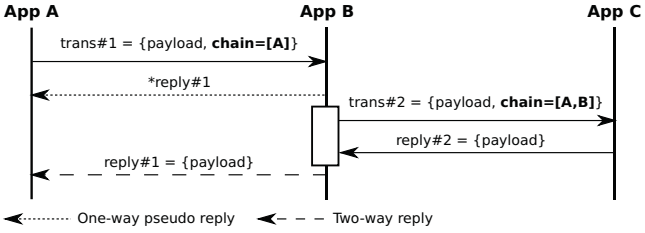


Figure 5: Call-chains during recursive Binder IPC calls.

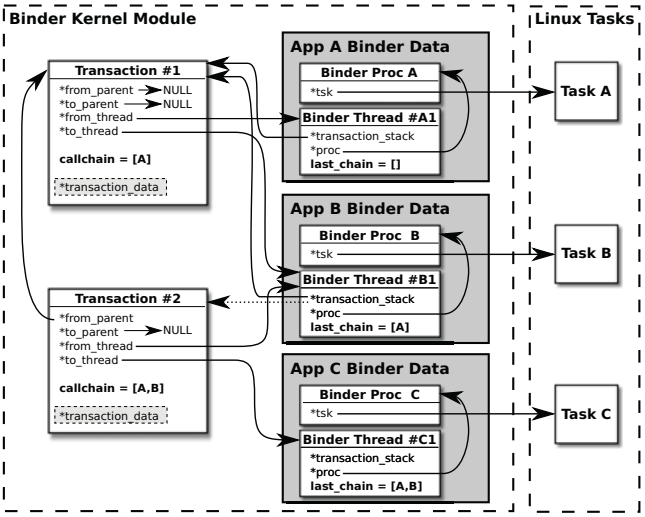


Figure 6: Constructing call-chains in recursive transactions. The dotted line represents the pointer *after* Transaction#2.

Binder transactions. The operations performed within the Binder kernel module in this scenario are depicted in Figure 6. In general, for each process that registers with Binder as a sender/receiver, the kernel module sets up `Binder Proc` information associated with the process as a whole and `Binder Thread` information associated with a particular thread of the application process (i.e., threads from the app's Binder IPC thread pool and the main thread). When *App A* sends a message to *App B*, the Binder kernel module creates new transaction data `Transaction#1`, where `transaction_data` contains the message, `from_parent` and `to_parent` point to parent transactions at the sender's and receiver's side in case of recursive transactions, and `from_thread` and `to_thread` point to the sender's and receiver's involved `Binder Threads`. The `transaction_stack` attribute of `Binder Thread` points to the last processed (i.e., last sent or received) transaction of the associated thread. When an IPC thread of *App B* is ready to receive this transaction, the message is copied from the kernel to the thread's user space. At this point the kernel module provides the receiver thread with the sender UID (cf. Step 4 in Figure 2), which is retrieved from the kernel task information associated with the transaction sender's `Binder Proc`.

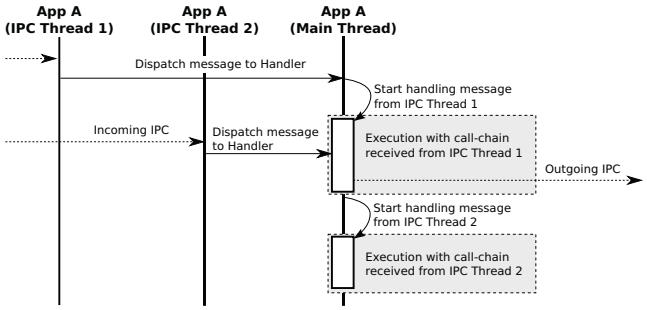


Figure 7: Message processing by Handler on Main Thread.

Constructing Call-Chains in Two-Way Transactions. As shown in Figure 5, *App B* issues a recursive call to *App C* while handling the call from *App A*. In Figure 6 this is depicted as `Transaction#2`. The attributes of `Transaction#2` are set the same as those of `Transaction#1`. However, since `Transaction#2` is the most recent transaction for *App B*, its `transaction_stack` pointer is adjusted accordingly. The dotted line represents the pointer *after* `Transaction#2`.

In Scippa, we attach the current state of the call-chain to each transaction data. To this end, we extended the `transaction` data structure with new attributes to hold a call-chain. Every time a recursive call is made, the kernel module copies the call-chain from the direct predecessor transaction to the new transaction (or starts a new call-chain if no predecessor exists) and appends the UID of the current sender to the call-chain. In addition to the UID, we also save the sender's process and thread ID in the call-chain and assign every new call-chain a unique ID.

Constructing Call-Chains in One-Way Transactions. For one-way transactions, the mechanism depicted in Figure 6 is not applicable, since `Transaction#1` is freed before `Transaction#2` is allocated. Thus, when continuing the chain, the data structure for the direct predecessor of `Transaction#2` no longer exists. To establish this missing link between transactions, we save the last received call-chain as part of the `Binder Thread`. When the application code executing on the IPC thread sends the next transaction (i.e., transaction *B*-*C*), the saved call-chain is continued. This solution is based on the observation that IPC threads are at any given time either executing app code as direct consequence of the last received IPC call, or, when this execution has finished, return to a state in which they can receive the next call and hence next call-chain.

5.2 Intra-App Call-Chain Propagation

As explained in Section 3, certain types of IPC messages are dispatched by the receiving IPC thread to the Main Thread (or a dedicated worker thread) by adding it to the Main Thread's Handler message queue. The Main Thread handles queued messages sequentially (cf. Figure 7). Thus, the current IPC context of the Main Thread directly depends on the message currently processed. To propagate the call-chain received by the IPC thread to the Main Thread, we extended the `Message` and `Looper` classes of Android's application libraries to attach the received call-chain to new messages for the Handler. The Main Thread's `Handler` class is extended to always update its current IPC context with the call-chain of the message it processes next. Thus all outgoing IPC during this processing continue the call-chain correctly.

5.3 Asynchronous Call-Chain Propagation

One particular challenge for establishing call-chains that include broadcast Intents are *sticky* broadcasts. As long as the sender app does not cancel the broadcast and does not get uninstalled, sticky broadcasts are stored by the *ActivityManagerService*. These broadcasts are even delivered to relevant Broadcast Receivers that register in the system *after* the broadcast Intent has been sent. To address this asynchrony within the control flow from the Intent sender to the receiver, we modified the *ActivityManagerService* to tag stored sticky broadcasts with the call-chain at the time the broadcast Intent was stored. Additionally, we modified the *ActivityManagerService*'s logic for delivering sticky broadcasts to adjust its current IPC context according to the call-chain stored with the sticky broadcast before sending and to restore its original IPC context afterwards. As a result, sticky broadcasts continue the call-chain so that its receivers can now identify the original broadcast sender.

5.4 Accessing Call-Chains from User Space

To provide the current call-chain information to the user space, we pass this information as part of the `binder_transaction_data` from the kernel to the IPC thread that receives the transaction. We extended the Android runtime library (cf. Section 2.2.1) to extract the call-chain from the transaction data and to subsequently store it in the thread local storage (TLS). From there it can be retrieved by any application code that is executed on the same thread. Similar to the default `getCallingUid` function, we introduce a new API function `getCallChainUids` to retrieve the call-chain. App developers can then retrieve information about those chained UIDs from the system—including the UIDs' package names, developer signatures, or permissions—and implement a fine-grained access control based on that information.

As explained earlier, some scenarios require that the user space is able to set its current call-chain. Therefore, we extended the `Binder` classes with functions to set the call-chain of the current thread. When a new Binder transaction is triggered, the set call-chain is passed to the Binder kernel module as part of the send transaction data. To prevent the user space from forging or modifying call-chains, a token-based approach—i.e., user space processes only hold a *read-only* reference, bound to their UID, to retrieve the call-chain from the kernel space—could enable the kernel to verify that call-chains retrieved from user space were originally created by the kernel and hence to discard illicit chains.

5.5 Returning Call-Chains to Message Senders

Our extension to the kernel module propagates finished branches of call-chains back to the initial sender. We extended the Binder protocol with a new flag `BR_CALLCHAIN` to send a finalized branch of a call-chain back to the app that started this chain. To distinguish branches of different call-chains, the kernel module additionally provides the call-chain ID to the user space. While this mechanism returns all finalized branches to the sender, ongoing work extends Android's application model to efficiently store and manage this information. We are in the process of implementing a new application component type dedicated to this task.

6. EVALUATION

In this section we evaluate and discuss the implementation of Scippa in terms of effectiveness and performance impact.

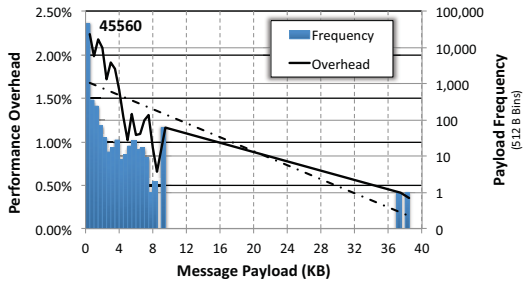


Figure 8: Performance overhead of Binder transactions vs. payload size and frequency breakdown of payload sizes.

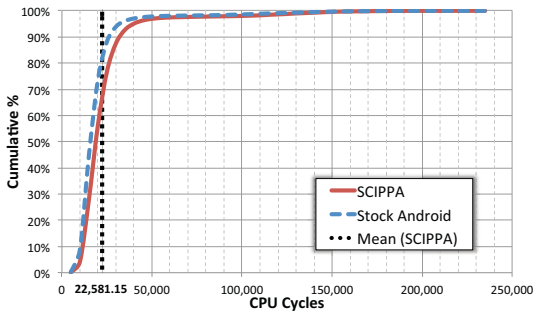


Figure 9: CDF of CPU cycles for Binder transactions on Scippa and stock Android.

6.1 Experimental Methodology

All our experiments were performed on a standard Galaxy Nexus development phone (Dual-core 1.2GHz Cortex-A9 CPU, 1GB RAM). We implemented Scippa as a modification to the Android OS code base of v4.2.2_r1.2b and the Android Linux kernel in branch *android-omap-tuna-3.0-jb-mr1.1*. Since the resolution of the Linux time facilities were too coarse grained for our microbenchmarks of Binder transactions, we leveraged the ARM *Performance Monitoring Unit* to calculate the performance overhead in CPU cycles. Since the PMU counts cycles per CPU core on our dual-core platform, we modified the Binder kernel module to acquire a spinlock for the duration of each measurement, thus eliminating the possibility that the current Binder thread is re-scheduled on the other CPU core while executing the measured code segment. Additionally, to be able to estimate the performance overhead in seconds, we adjusted the CPU frequency governor to always clock the CPUs at the maximum rate of 1.2GHz. Finally, to reduce the level of white noise in our measurements, we did not run any other apps except for our benchmark apps. Thus, all measurements approximate the *lower* bound for the actual overhead.

6.2 Performance Impact

To evaluate the performance impact of Scippa, we performed i) microbenchmarks of transactions in the Binder kernel module and ii) reimplemented relevant parts of the user space benchmarks of the closely related work *Quire* [4].

Transaction microbenchmarks. We performed microbenchmarks within the Binder kernel module for building and continuing call chains as described in Section 5. The results of our benchmarks are based on the measurements of 52,777 Binder transactions. Figure 8 presents the relative overhead vs. the data payload of the transaction. The maximum overhead was 2.23% and this overhead decreased with increasing payload size, where the memory copy operations for the data buffer outweigh the call-chain operations. However, when taking the frequencies of different payload sizes into consideration, the weighted average remains at 2.23%.

This small performance overhead is further illustrated in Figure 9, which shows the cumulative frequency distribution of the CPU cycles required for performing Binder transactions in Scippa and stock Android. On average, transactions on stock Android required 18,850 cycles and 99% of the measured transactions required less than 115,000 cycles. On Scippa this performance merely decreases to an average of 22,581.15 cycles per transaction, which translates to 18.82μs in our experimental setup, and 99% of the transactions required less than 130,000 cycles.

It should be noted that 0.58% of the measurements in our data set were extreme outliers that were in the range of 3 to 5 magnitudes higher than the rest of the measurements. These outliers occurred in both stock Android and Scippa benchmarks. We could trace these outliers back to thread blocking during parsing of the binder transaction header, i.e., *operations independent from our* Scippa *modifications*. However, since these rare outliers significantly distorted the mean and the margin of error in our measurements, we excluded them from the results presented here.

User space benchmark [4]. Since applications in Scippa have to retrieve, parse, and set the received call-chain as part of their Binder IPC thread operations (cf. Section 5), we measured the overhead of Scippa from the application layer perspective. To this end and to provide a better comparison with existing work, we re-implemented the test cases presented in the closest related work *Quire* [4]. In this test, the Service components of several test apps interact a) to pass a message with variable size payload roundtrip between two apps and b) to send a message without payload roundtrip between multiple apps to build call-chains of different lengths.

Figure 10 presents the average performance per roundtrip versus the message payload as computed from 11,000 measurements per payload size. The payload size ranges from 0 bytes to 6,336 bytes in 64 bytes increments. In our data set, Scippa imposed between 3.70-25.33% overhead, which is comparable to Quire's performance (21% slowdown).

Figure 11 shows the average performance per roundtrip vs. the call-chain length with a max length of 9 (i.e., 10 apps involved) and 11,000 measurements per length. Scippa's overhead is 12.70-26.73%, which is again comparable to Quire (20-25% slowdown).

6.3 Binder IPC Provenance

In this section, we provide statistics on the call-chains observed in Scippa during our tests and evaluate how well these call-chains provide the necessary IPC provenance information to efficiently mitigate the different attacks introduced in our requirements analysis (cf. Section 4).

Call-chain Statistics. Table 1 summarizes statistics on call-chains observed during our testing. All margins of error are for a 95% confidence. We logged in total 54,670 call-chains with an average length of 1.56. All chains had at least two branches with 2.59 being the average number of branches per chain. Figure 12 provides a breakdown of the branch

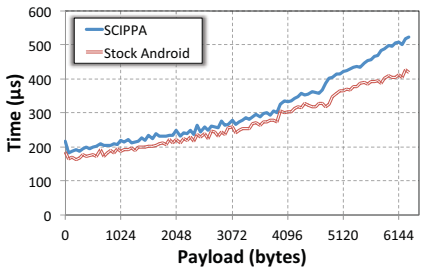


Figure 10: Single Service call roundtrip time vs. payload size.

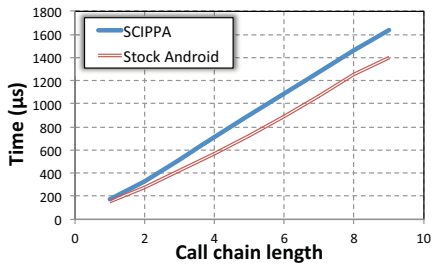


Figure 11: Service call roundtrip time vs. call chain length.

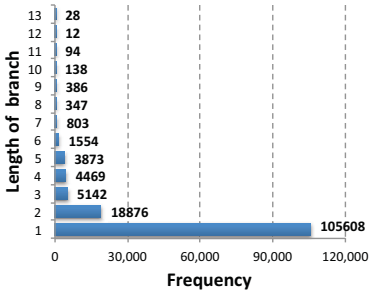


Figure 12: Break down of observed call-chain lengths.

General:		Branching:		Dispatching:	
#Call-Chains:	54,670	#Chains with branches:	54,760 (100%)	#Chains with dispatching:	3,237 (5.91%)
Chain length:	1.56 ± 0.01	#Branches (total):	141,330	#Dispatches (total):	24,966
Max. length:	13	#Branches (per chain):	2.59 ± 0.08	#Dispatches (per chain):	7.71 ± 1.92
		Max. number of branches:	1,194	Max. number of dispatches:	2,784

Table 1: Call-chain statistics.

lengths and shows that the maximum observed length is 13 and that chains with a length of one are most frequent. The max number of branches for one chain is 1,194. Additionally, 5.91% of all chains contained at least one dispatch between an IPC thread and the Main Thread. On average, each chain contained 7.71 dispatches with 2,784 being the highest number of dispatches observed for one chain.

Attack mitigation. To verify that SCIPPA fulfills the requirements stated in Section 4.2, we developed a set of interacting test applications, which implement different combinations of inter-component communication that model the scenarios that have been reported to be prone to attacks such as confused deputy attacks [22] or Intent hijacking [2]. This includes common inter-app communications such as starting Activities, Broadcast Intents, or binding and calling Services including IntentServices. In general, all called components were able to retrieve the call-chain of the direct thread of control that lead to their invocation and thus identify the initiator of the call-chain. Also, all senders were successfully notified by the kernel module about finalized call-chain branches. Based on this information, we were able to implement a per-UID access control that is more fine-grained and flexible than Android's static Permission system.

Since the most well-known reported confused deputy attacks rely on Broadcast Receivers [22], we briefly elaborate on call-chains for broadcast receivers in our testbed. Figure 13 shows an established call-chain for a single Broadcast Intent send by the app with UID 10043 (lower left corner). The Intent is sent to the *ActivityManagerService* as part of the system server with UID 1000, where the task to send this broadcast is dispatched to a dedicated thread (1000:403:777 → 1000:403:420). This thread delivers the Broadcast Intent in parallel to all *dynamically* registered receivers (upper left rectangle) and in *order*[3] to all receivers registered statically through the applications' manifests (lower right rectangle). Each app receives the Intent via an IPC thread and processing of the Intents by the Broadcast Receiver components is dispatched to the apps' Main Threads.

As a consequence, each Broadcast Receiver is able to retrieve the branch of the call-chain that lead to its invocation and, hence, to identify the sender of the broadcast. For instance, the receiver of UID 10047 retrieves the call-chain 10043 → 1000 and the receiver of UID 10045 retrieves the chain 10043 → 1000 → 10044 → 1000 that shows all receivers previous to itself in the ordered delivery. Using this information, Broadcast Receivers in SCIPPA can now efficiently evaluate their trust in received messages and their senders, which allows them to react accordingly by, e.g., refusing to accept spurious messages. In addition to our test cases, we also verified that the privileged Broadcast Receiver that was reported as a confused deputy in [22] is now able to identify the broadcast Intent sender and hence to apply fine-grained access control depending on the Intent payload (e.g., GPS vs. Wi-Fi control commands). That eliminates the confused deputy vulnerability without the need to split its component interface or to introduce new permissions.

Additionally, the sender received from the kernel module four `BR_CALLCHAIN` notifications about the call-chain branches that ended with the apps with UIDs 10045 through 10048. Thus, the sender is able to identify the receivers of its broadcast and to determine if its broadcast was potentially hijacked by an unintended receiver [2]. In case of ordered broadcasts, it can even determine which app was responsible for cancelling the further delivery of the broadcast.

6.4 Discussion and Limitations

The most important limitation for the effectiveness of our approach is that the call-chain can be lost if communication between threads occurs over channels currently not covered by SCIPPA. With message dispatching, SCIPPA covers one of the major communication channels between Android application threads, but other channels exist (e.g., `notify`). Future work has to address these channels through extensions to the Dalvik VM and Java language classes in the Android framework. For instance, initial experiments have shown

[3]Broadcast receivers registered through the manifest are *always* served in ordered fashion, but intermediary receivers only can stop further delivery when the *ordered* flag is set.

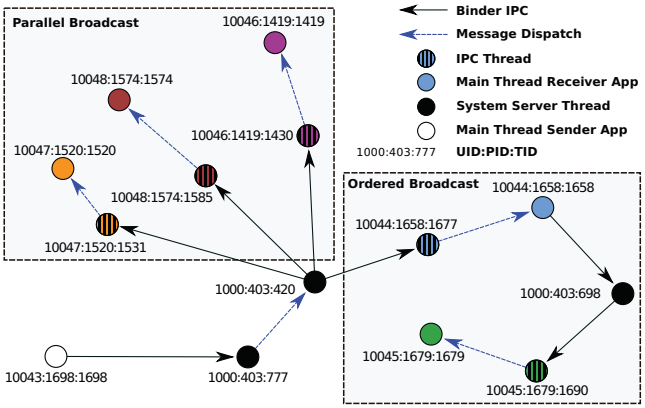


Figure 13: Call-chain for parallel and ordered broadcasts.

that it is possible to forward the call-chain to newly spawned threads, e.g., for *IntentService* components or *AsyncTasks*.

Similarly, Scippa currently only covers direct control flows for ICC. Hence, indirect control flows are an open problem. Providing an efficient solution to address indirect control flows is an orthogonal problem and affects other approaches such as dynamic taint-tracking [6] as well.

Since our solution relies on code within the app sandboxes to forward call-chains on message dispatching, this code base is prone to attacks by malicious apps. While forging and modifying call-chains can be prevented by implementing a token-based approach (cf. Section 5.4), the deliberate dropping of call-chains cannot be prevented. However, because UIDs are attached by the kernel to the call-chains during the sending of IPC calls, a malicious app can only hide previous hops in the call-chain but not itself. Thus, a malicious app cannot fool a receiver into trusting it by hiding its predecessors in the call-chain [4] and this is primarily a problem if multiple malicious apps collude [16, 24, 1].

7. RELATED WORK

IPC-based domain isolation. Thread-migrating IPC has been used in high-assurance systems [15, 28, 26] as building block for domain-based isolation by factoring applications into smaller domains. Domains are usually compartmentalized at process boundaries and IPC is used to connect them. The security of IPC has been investigated from different angles, e.g., for synchronous IPC [27] or language-based security [9]. Android's design borrows ideas from this literature (e.g., UID-based app sandboxes, connected via thread-migrating IPC). However, in this paper we identified and addressed misaligned security assumptions when *Inter-Component Communication* is built on top of IPC.

Provenance frameworks. Establishing provenance information has been primarily investigated for data provenance [29, 17] and in distributed systems [30]. Specific to smartphones, the *SPADE* framework [10] has been ported to Android [13]. It uses the Binder debug interfaces to profile the IPC on Android and generate useful traces for device auditing. Scippa provides very similar information, however, in contrast to SPADE on Android, Scippa also provides the links between IPC channels along direct ICC control flows and thus valuable information for a more detailed auditing.

A recent approach called *EPIC* [21] uses static analysis to detect all potential Intent-based communication channels between application components. The information created by Scippa includes this information as well. However, Scippa only reflects actual runtime behavior. A combination of these two approaches could lead to a more comprehensive app testing, in which static analysis shows all potential channels and Scippa fills the gaps in this analysis (e.g., when a target cannot be resolved statically).

Android security. Research has established a large body of literature on Android security. With respect to preventing confused deputy attacks, related work [22] has proposed the poli-instantiation of apps in ICC to reduce the callee's privileges to the ones of their caller. *XManDroid* [1] monitors all ICC communication and applies at runtime Chinese Wall security policies to prevent communication that could lead to a dangerous information flow. While poli-instantiation and XManDroid rely on strictly restricting privileges or communication channels, Scippa and closely related work (*Quire* [4]) rely on provisioning IPC provenance information to callees to enable them to apply fine-grained access control on their ICC interfaces. This allows them to securely provide APIs to other apps. In contrast to Scippa, however, Quire's prototypical implementation requires developers to explicitly extend all Stub and Proxy interfaces in order to construct call-chains. Scippa abstains from a developer-centric approach for establishing IPC provenance information and implements a system-centric solution that builds call-chains transparently to developers. Moreover, in Quire's developer-centric approach, call-chains are created and extended within the app code and, hence, this approach requires verifiable statements to establish trust in and authenticity of chains. In Scippa, the kernel creates and extends the call-chains and only when chains are propagated through the user-space back to the kernel, a lightweight cryptographic mechanism (e.g., tokens) is required to ensure authenticity of chains. Additionally, as a kernel-based solution, Scippa covers even cases in which apps do not use Stubs, but instead app code (e.g., native libs) communicates directly with the Binder kernel module.

Besides confused deputy attacks, related work has proposed a solution [14] to mitigate Intent hijacking by applying heuristics-based access control for Intents to prevent their unintended delivery. While this is a system-centric, preventive security extension, Scippa provides a different trade-off. Scippa adds measures that allow an application to detect (not prevent) such attacks, but in turn provides a higher precision than heuristics due to its call-chain information.

Information flow control. Concepts from decentralized information flow control [20, 19], e.g., as implemented in the DEFCON [18] and Asbestos [5] operating systems, have been applied within different solutions on Android. Most prominent solutions based on dynamic taint analysis include *TaintDroid* [6], *AppFence* [12], and *Paranoid Android* [23]. In contrast, Scippa does not aim at restricting information flows of sensitive data at information sinks, but instead aims at providing apps with IPC provenance information that enables them to effectively apply access control for sensitive data and functionality.

8. CONCLUSION

In this paper, we presented Scippa, our architecture for provisioning Binder IPC provenance information on Android. It allows app components to identify the sending app of incoming IPC messages despite indirections and message dispatching. Using this provenance information, apps are now

able to effectively apply per-sender access control to their interfaces. In contrast to related work, SCIPPA constitutes a system-centric approach that directly addresses conceptual shortcomings in Android's multi-layered inter-application communication. We presented an implementation of SCIPPA based for Android v4.2.2 and the evaluation of our prototype showed that SCIPPA imposes only minimal overhead when compared to stock Android. In addition, we deem the lessons learned from SCIPPA valuable for the design of future multi-layered OS security architectures that rely on thread-migration and that support liberal inter-app communication.

The source code of SCIPPA can be retrieved from `http://infsec.cs.uni-saarland.de/projects/scippa/`.

Acknowledgements

We thank the anonymous reviewers for their comments. 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). Sven Bugiel thanks Rose Hoberman for her valuable editorial comments.

9. REFERENCES

[1] S. Bugiel, L. Davi, A. Dmitrienko, T. Fischer, A.-R. Sadeghi, and B. Shastry. Towards taming privilege-escalation attacks on Android. In *NDSS'12*. Internet Society, 2012.

[2] E. Chin, A. Porter Felt, K. Greenwood, and D. Wagner. Analyzing inter-application communication in Android. In *MobiSys'11*. ACM, 2011.

[3] L. Davi, A. Dmitrienko, A.-R. Sadeghi, and M. Winandy. Privilege escalation attacks on Android. In *ISC'10*. Springer, 2010.

[4] M. Dietz, S. Shekhar, Y. Pisetsky, A. Shu, and D. S. Wallach. Quire: Lightweight provenance for smart phone operating systems. In *USENIX Security'11*, 2011.

[5] P. Efstathopoulos, M. Krohn, S. VanDeBogart, C. Frey, D. Ziegler, E. Kohler, D. Mazières, F. Kaashoek, and R. Morris. Labels and event processes in the Asbestos operating system. In *SOSP'05*. ACM, 2005.

[6] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *OSDI'10*, 2010.

[7] W. Enck, M. Ongtang, and P. McDaniel. On lightweight mobile phone application certification. In *CCS'09*. ACM, 2009.

[8] W. Enck, M. Ongtang, and P. McDaniel. Understanding Android security. *IEEE Security and Privacy*, 7(1):50–57, 2009.

[9] M. Fähndrich, M. Aiken, C. Hawblitzel, O. Hodson, G. Hunt, J. R. Larus, and S. Levi. Language support for fast and reliable message-based communication in Singularity OS. In *EuroSys'06*. ACM, 2006.

[10] A. Gehani and D. Tariq. Spade: Support for provenance auditing in distributed environments. In *Middleware 2012*, volume 7662 of *Lecture Notes in Computer Science*. Springer, 2012.

[11] N. Hardy. The confused deputy: (or why capabilities might have been invented). *SIGOPS Oper. Syst. Rev.*, 22(4):36–38, Oct. 1988.

[12] P. Hornyack, S. Han, J. Jung, S. Schechter, and D. Wetherall. These aren't the droids you're looking for: retrofitting Android to protect data from imperious applications. In *CCS'11*. ACM, 2011.

[13] N. Husted, S. Qureshi, D. Tariq, and A. Gehani. Android provenance: diagnosing device disorders. In *TAPP'13*. USENIX, 2013.

[14] D. Kantola, E. Chin, W. He, and D. Wagner. Reducing attack surfaces for intra-application communication in Android. In *SPSM'12*. ACM, 2012.

[15] B. W. Lampson and H. E. Sturgis. Reflections on an operating system design. *Commun. ACM*, 19(5):251–265, May 1976.

[16] C. Marforio, H. Ritzdorf, A. Francillon, and S. Čapkun. Analysis of the communication between colluding applications on modern smartphones. In *ACSAC'12*. ACM, 2012.

[17] P. McDaniel, K. Butler, S. McLaughlin, R. Sion, E. Zadok, and M. Winslett. Towards a secure and efficient system for end-to-end provenance. In *TAPP'10*. USENIX, 2010.

[18] M. Migliavacca, I. Papagiannis, D. M. Eyers, B. Shand, J. Bacon, and P. Pietzuch. Defcon: high-performance event processing with information security. In *USENIX ATC'10*, 2010.

[19] A. C. Myers and B. Liskov. A decentralized model for information flow control. In *SOSP'97*. ACM, 1997.

[20] A. C. Myers and B. Liskov. Protecting privacy using the decentralized label model. *ACM Trans. Softw. Eng. Methodol.*, 9(4):410–442, Oct. 2000.

[21] D. Octeau, P. McDaniel, S. Jha, A. Bartel, E. Bodden, J. Klein, and Y. L. Traon. Effective inter-component communication mapping in Android: An essential step towards holistic security analysis. In *USENIX Security'13*, 2013.

[22] A. Porter Felt, H. J. Wang, A. Moshchuk, S. Hanna, and E. Chin. Permission re-delegation: Attacks and defenses. In *USENIX Security'11*, 2011.

[23] G. Portokalidis, P. Homburg, K. Anagnostakis, and H. Bos. Paranoid Android: Versatile protection for smartphones. In *ACSAC'10*. ACM, 2010.

[24] R. Schlegel, K. Zhang, X. Zhou, M. Intwala, A. Kapadia, and X. Wang. Soundcomber: A stealthy and context-aware sound trojan for smartphones. In *NDSS'11*. Internet Society, 2011.

[25] T. Schreiber. Android Binder – Android Interprocess Communication. `https://www.nds.rub.de/media/attachments/files/2011/10/main.pdf`, 2011.

[26] M. D. Schroeder, D. D. Clark, and J. H. Saltzer. The Multics kernel design project. In *SOSP '77*. ACM, 1977.

[27] J. S. Shapiro. Vulnerabilities in synchronous ipc designs. In *IEEE SP'03*, 2003.

[28] J. S. Shapiro, J. M. Smith, and D. J. Farber. EROS: a fast capability system. In *SOSP '99*. ACM, 1999.

[29] Y. L. Simmhan, B. Plale, and D. Gannon. A survey of data provenance in e-science. *SIGMOD Rec.*, 34(3):31–36, Sept. 2005.

[30] E. Wobber, M. Abadi, M. Burrows, and B. Lampson. Authentication in the Taos operating system. *ACM Trans. Comput. Syst.*, 12(1):3–32, Feb. 1994.

Android Security Framework: Extensible Multi-Layered Access Control on Android

Michael Backes, Sven Bugiel, Sebastian Gerling, Philipp von Styp-Rekowsky
{backes,bugiel,sgerling,styp-rekowsky}@cs.uni-saarland.de

Saarland University/CISPA, Germany

ABSTRACT

We introduce the *Android Security Framework (ASF)*, a generic, extensible security framework for Android that enables the development and integration of a wide spectrum of security models in form of code-based security modules. The design of ASF reflects lessons learned from the literature on established security frameworks (such as Linux Security Modules or the BSD MAC Framework) and intertwines them with the particular requirements and challenges from the design of Android's software stack. ASF provides a novel security API that supports authors of Android security extensions in developing their modules. This overcomes the current unsatisfactory situation to provide security solutions as separate patches to the Android software stack or to embed them into Android's mainline codebase. This system security extensibility is of particular benefit for enterprise or government solutions that require deployment of advanced security models, not supported by vanilla Android. We present a prototypical implementation of ASF and demonstrate its effectiveness and efficiency by modularizing different security models from related work, such as dynamic permissions, inlined reference monitoring, and type enforcement.

1. INTRODUCTION

For several decades now, the need for operating system security mechanisms to provide strong security and privacy guarantees has been well understood [24, 34, 26, 5]. Yet, recent classes of attacks against smartphone end-user's privacy and security [19, 41, 29, 9] have shown that the fairly new smart device operating systems fail to provide these strong guarantees, for instance, with respect to access control or information flow control. To remedy this situation, security research has proposed a wide spectrum of security models and extensions for mobile operating systems, most of them for the popular open-source Android OS. These extensions include context-related access control [10], developer-centric security policies [28], and dynamic, fine-grained permissions [42, 21, 3]. They also comprise security models [7, 33, 36, 8] such as domain isolation and type enforcement, which are usually at the heart of enterprise and governmental security solutions.

However, the lack of a comprehensive security API for the development and modularization of security extensions on Android has created the unsatisfactory situation that all of these novel and warranted security models are either provided as model-specific patches to the Android software stack, or they became an integrated component of the Android OS design [36]. When considering the body of literature on established security frameworks, such as *Linux Security Modules* (LSM) [40] or the *BSD MAC Framework* [39], their history has taught that the need to patch the OS or the hardwiring of a specific security model impairs both the practical and theoretical benefits of security solutions. First, there is in general no consensus on the *"right"* security model, as demonstrated by the broad range of Android security extensions [10, 28, 3, 42, 7, 36]. Thus, OS security mechanisms should not limit policy authors to one specific security model by embedding it into the OS design. Second, providing security solutions as *"security-model-specific Android forks"* impedes their maintainability across different OS versions, because every update to the Android software stack has to be re-evaluated for and applied to each fork separately.

Contributions. In this paper, we propose the design and implementation of Android Security Framework (ASF), which allows security experts to develop and deploy their security models in form of modules as part of Android's platform security. This provides the means to easily extend the Android security mechanisms and avoids that security designers have to choose "the right Android security fork" or that the OS vendor has to impose a specific security model. In the design of ASF we transfer the lessons learned and guiding principles from the literature on established OS security infrastructures to Android and intertwine them with new requirements for efficient security policies for multi-tiered software stacks of smart devices. In contrast to concurrent, independent work [20], which introduced extensibility for security *apps* (i.e., add-ons), our design establishes a generic and extensible security framework that allows instantiating security models *by design* as part of Android's platform security and enables not only extending but also replacing Android's default security mechanisms. This is particularly beneficial when tailoring Android for higher-security deployments like enterprise phones, where the default mechanisms are insufficient or even obsolete (e.g., when the IT department is an additional stakeholder that decides on apps' privileges and installation). We make the following contributions:

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.
ACSAC'14, December 08–12, 2014, New Orleans, LA, USA
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-3005-6/14/12 ...$15.00.
http://dx.doi.org/10.1145/2664243.2664265.

1. Policy-agnostic, multi-tiered security infrastructure: The security infrastructure must avoid committing to one particular security model and enable authors of security extensions to develop as well as deploy their solutions in form of code. This requires special consideration of Android's multi-tiered software stack and the dominant programming languages at each layer. For ASF we solve this by integrating security-model-agnostic enforcement hooks into the Android kernel, middleware and application layer and exposing these hooks through a novel security API to module authors.

2. Enabling edit automata policies: Various Android security solutions realize edit automata policies that not only truncate but also modify control flows. In ASF, the application layer and middleware hooks are specifically designed to allow module authors to leverage the rich semantics of Android's application framework and to implement their security policies as edit automata. This required a re-thinking of the "classical" object manager design from the literature by shifting the edit automata logic from the infrastructure into the security modules.

3. Instantiation of existing security models: We demonstrate the efficiency and effectiveness of our ASF by instantiating different security models from related work on type enforcement [8, 36] and inlined access control [3] as well as from Android's default security architecture as modules.

4. Maintenance benefits for security extensions: Our ported security modules show how ASF simplifies maintainability of security extensions across different OS versions by shifting the bulk of effort to the security framework maintainer. This is similar to the maintenance of the application framework for regular apps. Hence, a comparable benefit to regular apps in adaption and stability across OS versions can be expected of security modules.

5. Research and development benefits: We postulate that developing security solutions against a well documented security API also greatly contributes to *a)* a better understanding and analysis of new security models that form a self-contained unit instead of being integrated to various components of the Android software stack, *b)* a better reproducibility and dissemination of new solutions since modules can be easily shared and instantiated, and *c)* a more convient application of security knowledge to the Android software stack without the requirement to be familiar with the deep technical internals of Android.

2. BACKGROUND ON ANDROID

In this section we provide necessary technical background information on Android.

2.1 Primer on Android

Android is an open-source software stack for embedded devices. The lowest level of this stack consists of a Linux kernel responsible for elementary services such as memory management, device drivers, and an Android-specific lightweight inter-process communication called Binder. On top of the kernel lies the extensive Android middleware, consisting of native libraries (e.g., SSL) and the application framework. System services in the middleware implement the bulk of Android's application API (e.g., the location service) and pre-installed system apps at the application layer, like Contacts, complement this API.

Although application layer and middleware apps and services are commonly written as Java code, they are compiled to *dex* bytecode and run inside the *Dalvik Virtual Machine* (DVM). In addition to dex bytecode, apps and services can use native code libraries (i.e., C/C++) for low-level interactions with the underlying Linux system. Native code can be seamlessly integrated into dex bytecode by means of the *Java Native Interface*.

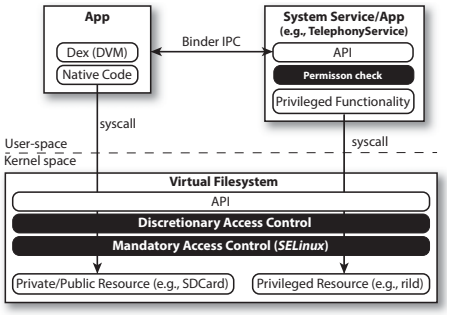


Figure 1: Android's default security architecture.

Android apps are generally composed of different components. The four basic app components are *Activities* (GUI for user interaction), *BroadcastReceivers* (mailbox for broadcast *Intent* messages), *ContentProviders* (SQL-like data management), and *Services* (long running operations without user interaction). All components can be interconnected remotely across application boundaries by using different abstractions of Android's Binder IPC mechanism, such as *Intent* messages.

2.2 Android's Security Philosophy

Android's security philosophy dictates that all apps are sandboxed by executing them in separate processes with distinct user IDs (*UID*) and assigning them private data directories on the filesystem.

To achieve privilege separation between apps, Android introduces *Permissions*, i.e., privileges that an app is granted by the user at install-time. In accordance with the least privilege principle, an app without permissions is not able to access security and privacy sensitive resources. Permissions are assigned to the app's UID and enforced at two different points in the system architecture (cf. Figure 1): First, every app sandbox can directly interact with the kernel through system calls, for instance, to edit files or open a network socket. These resources are either of private nature (i.e., are within the app's private directory) or public resources (e.g., SDCard). Access control in the filesystem ensures that the apps' processes have the necessary rights (i.e., Permissions) to issue particular syscalls, e.g., opening a file. The filesystem access control consists of the traditional Linux Discretionary Access Control, which is complemented (since Android v4.3) by SELinux based Mandatory Access Control (MAC).

Second, apps can interact through the Android API in a strictly controlled manner with highly privileged resources. To ensure system security and stability, apps are prohibited to access those resources directly. Instead, those resources are wrapped by system services and apps that implement the API. For instance, the *TelephonyService* communicates on behalf of apps with the radio interface layer daemon (*rild*) to initiate calls or send text messages. Whether an app is sufficiently privileged to successfully call the API is determined by a Permission check within the system services/apps. For this

check, the Binder mechanism provides to the callee (system service/app) the UID of the caller (app).

3. RELATED WORK

We first provide a synopsis of the development of extensible kernel security frameworks and discuss afterwards the current status of security extensions and frameworks for Android.

3.1 Extensible Kernel Access Control

The importance of the operating system in providing system security has been very well studied in the last decades [34, 24, 5, 26] and different approaches to extending operating systems with access control and security policies have been explored. These include system-call interposition [15, 30], software wrappers [16], and extensible access control frameworks like *DTE* [4], *GFAC* [1], and *Flask* [37]. All of these solutions have been provided as kernel patches for Linux or UNIX. However, this led to an intricate situation: On the one hand, maintaining these solutions as patches incurred high maintenance costs for adapting the patches to kernel changes. On the other hand, none of these solutions was included in the vanilla kernel because this would constrain security policy authors to one specific security model. This constrain would be unsatisfying since there exists in general no consensus on the "right" security model. To remedy this situation, extensible security frameworks have been proposed [40, 39] that allow the extension of the system with trusted code modules that implement specific security models. Module authors are supported with an API that exposes kernel abstractions as well as operations and facilitates the implementation of the desired security architecture and model. The results of this research have been integrated into the mainline kernels as the *Linux Security Modules* framework (LSM) [40] and the *BSD MAC Framework* [39].

3.2 Android Security

Closest to our approach is the independently and concurrently developed *ASM* [20], which also provides a programmable interface for security extensions. In contrast to ASF, however, it targets "security apps" added in addition to the default Android security architecture. As a consequence, ASM has to address the intricate problem of including untrusted code into highly-privileged context for access control enforcement and consolidating it with existing policies. It avoids this Gordian knot through a trade-off between policy expressiveness and sandboxing of security apps. In contrast, our ASF framework resides beneath the default Android security framework and hence allows instantiation of security models that complement or even substitute parts of the default platform security (cf. Section 6). Hence, ASM can even be implemented as a module in ASF. By definition, we must trust the developer of security solutions for ASF.

In recent years, Android's security has been quite scrutinized, and a wide spectrum of security extensions has been brought forward. To name a few: *CRePE* [10] provides a context-related access control, where the context can be, e.g., the device's location. *Saint* [28] enables developer-centric policies that allow app developers to ship their apps with rules that regulate the app's interactions with other apps. Different approaches to more dynamic and fine-grained permissions have been proposed based on system-centric enforcement (e.g., *TISSA* [42]) or inlined reference monitors (*Dr. Android and Mr. Hide* [21] or *AppGuard* [3]). *XManDroid* [6] enforces Chinese Wall policies to prevent confused deputy and collusion attacks. *TrustDroid* [7] and *MOSES* [33] isolate different domains such as "Work" and "Private" from each other. *SE Android* [36] and *FlaskDroid* [8] bring type enforcement to Android, where SE Android focuses on the kernel layer and has been partially included into the mainline Android source code, and FlaskDroid extends type enforcement to Android's middleware layer on top of SE Android.

4. REQUIREMENTS ANALYSIS

The current development of Android security extensions has strong parallels to the initial development of the above mentioned Linux and BSD security extensions, since current Android security extensions are provided as patches to the software stack or, in the case of SE Android [36], are embedded into the Android source tree. For the same, above mentioned reasons as for the early Linux and BSD security extensions, this impedes the applicability and adaption of Android security extensions and additionally precludes many of the benefits that a modular composition could offer in terms of maintenance: Embedding SE Android's security model into Android's source tree limits policy authors to the expressiveness and boundaries of type enforcement, whereas provisioning security models and architectures as patches to Android's software stack forces policy authors to chose a solution-specific Android fork. This requires for every version update to the Android OS a re-evaluation and port of each separate fork. Moreover, security solutions cannot be easily compared with each other, because their infrastructures are deeply embedded into the Android software stack.

In this paper, we develop in the spirit of the two de facto most established security frameworks, *Linux Security Modules* (LSM) [40] and the *BSD MAC Framework* [39], a generic and extensible Android Security Framework that allows the instantiation and deployment of different security models as modules at Android's application layer, middleware, and kernel. The two most important guiding principles from LSM and the BSD MAC framework that govern the design of our Android Security Framework are: *1)* provisioning of policies as code instead of data; and *2)* providing a policy-agnostic OS security infrastructure. In the remainder of this section, we analyze the requirements and challenges for their transfer to the Android software stack.

Policy as code and not data. The first guiding principle is that policies should be supported as code instead of data (such as rules written in one predetermined policy language). Providing an extensible security framework that supports integration of policy logic as code avoids committing to one particular security model or architecture. For Android, this removes the need to chose a particular extension-specific Android fork or to be limited to one specific security model in the mainline Android software stack. Additionally, developing modules against an OS security API provides the benefits of modularization for developing and maintaining security extensions. This includes, foremost, a higher functional cohesion of security modules and lower coupling with the Android software stack and, hence, can significantly reduce the maintenance overhead of modules, especially in case of OS changes. Moreover, it allows a better dissemination, comparison, and analysis of self-contained security modules.

Transferring this principle to an extensible security framework for Android poses the additional requirement to consider the semantics and dominant programming languages of the

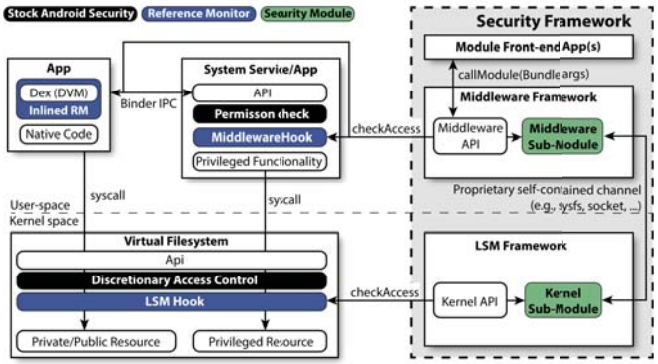


Figure 2: Android Security Framework architecture.

different layers of Android's software stack. LSM and the BSD MAC Framework, for instance, as part of the kernel, support modules written in C and operate on kernel data structures (e.g., filesystem inodes). While this applies to the Android Linux kernel as well, an Android security framework should additionally support modules written for Android's semantically-rich middleware and application layers. That means modules written in Java and operating on application framework classes (e.g., Intents or app components).

Policy-agnostic security infrastructure. The second principle is that the security framework and its API should be policy-agnostic. This means that policy-specific intrusions into the software stack are avoided and policy-specific data structures and logic are confined to security modules.

A particular additional requirement for a security framework on Android are enforcement hooks in the middleware and application layer that support *edit automata* [23] policies, as promoted by different solutions [42, 7, 21, 3]. Edit automata, in contrast to truncation automata, can not only abort control flows but also divert or manipulate them and, thus, give policy authors a higher degree of freedom in implementing their enforcement strategies. For instance, when querying a ContentProvider component, the policy could simply deny access by throwing a Java Exception (truncation), but also modify the return value to return filtered, empty, or fake data (edit). To technically enable security modules to implement edit automata, our design requires a re-thinking of the "classical" object manager vs. policy server design that is used, e.g., in LSM. Object managers (i.e., enforcement points) are responsible for assigning security labels to the objects that they manage and for both requesting and enforcing access control decisions from the policy server (i.e., policy decision point). Because this design embeds the enforcement logic into the system independently from the security model, it is unfit for realizing edit automata. Thus, our design requires hooks that generically support different enforcement strategies and shift the enforcement and object labelling logic from the object managers to the security modules.

5. ANDROID SECURITY FRAMEWORK

In the following we present ASF. We provide further details in an anonymous long version of this paper [2].

5.1 Framework Overview

The basic idea behind our Android Security Framework is to extend Android with a new security API that incorporates the design principles explained in Section 4. This API allows to easily author, integrate, and enforce generic security policies. Figure 2 provides an overview of our ASF and we explain its building blocks in the following.

5.1.1 Reference Monitors

In our design we differentiate between policy enforcing and policy decision making code. For enforcement we use reference monitors [22] at all layers of the Android software stack, i.e., at the application layer, the middleware layer, and the kernel layer. Each reference monitor protects one specific privileged resource and is placed such, that it mediates all access to the resource through the Android API. The benefit of this multi-tiered enforcement is that each reference monitor can operate with the semantics of its respective layer.

5.1.2 Security Modules

Security extensions are deployed in the form of code modules and loaded during boot into the security frameworks at the middleware and kernel level. Modules should be signed to ensure their integrity and trustworthiness, and the verification key is embedded in the kernel (or a secure location like a secure execution environment). Each module implements a policy engine that manages its own security policies and acts as policy decision making point. Security modules are integrated into the security frameworks through a security API that exposes objects and operations of the different software stack layers.

To provide a clear separation between policy decision logic using kernel level semantics and logic using middleware/application layer semantics, each module consists of two submodules: a Kernel Sub-Module leveraging the already existing Linux Security Module (LSM) infrastructure of the Linux kernel and a Middleware Sub-Module, for which we designed and implemented a novel security infrastructure at the application and middleware layers.

5.1.3 Front-end Apps

To enable user configurable policies or graphical event notifications, modules might want to include user interfaces. To this end, the module developers (or external parties being aware of the modules) can deploy standard Android apps that act as front-end and that communicate through the framework API with the module. We enable such proprietary module interfaces through a *Bundle* based communication protocol. A Bundle is a key-value store that supports heterogenous value types (e.g., Integer and String) and that can be transmitted via Binder IPC. It is the responsibility of the module to verify that the caller is sufficiently privileged.

5.2 Framework Infrastructure

We present now in a bottom-up approach details about the ASF infrastructure that has been prototypcially implemented for Android v4.3 and currently comprises 4606 lines of code.

5.2.1 Kernel Space

At kernel level we employ the existing Linux Security Module (LSM) [40] framework of the Linux kernel. LSM implements an infrastructure for mandatory access control and provides a number of enforcement hooks within kernel components such as the process management or the virtual filesystem. The Kernel Sub-Module is implemented as a standard Linux Security Module that registers through

Listing 1: Exemplary enforcement functions

```
1 public boolean deliverToRegisteredReceiver (Intent intent,
      ComponentName targetComp, String requiredPermission,
      int targetUid, int targetPid, String callerPackage,
      ApplicationInfo callerApp, int callingUid, int callingPid);
2 public Location getLastLocation (Location currentLocation,
      LocationRequest request, int callingUid, int calingPid);
```

the LSM API for the LSM hooks in the system and that operates with kernel level semantics. Kernel Sub-Module can be an existing Linux security module like SELinux or proprietary ones [20]. Kernel-level policies form truncation automata that terminate illegal control flows.

Since there might be operational inter-dependencies between the Kernel Sub-Module and user-space processes like the Middleware Sub-Module (e.g., propagation of access control decisions), the kernel module can implement proprietary channels for communication between kernel- and user-space (e.g., sysfs entries).

5.2.2 Middleware Layer

At the middleware layer we extended the system services and apps that implement the Android API with hooks that enforce access control decisions made by the Middleware Sub-Module. The middleware security framework is executed as a new Android system service and mediates between our hooks and the Middleware Sub-Module. The hooks are policy-agnostic and not tailored to one specific security model. Each hook takes as arguments all relevant, ambient information of the current control flow that led to the hook's invocation. For instance, Listing 1 presents two exemplary hooks in our system: one for the Intent broadcasting subsystem of the *ActivityManagerService* (line 1) and one for the *LocationManagerService* that implements the location API of Android (line 2). Both provide to the Middleware Sub-Module information about the current caller to the Android API, i.e., App in Figure 2 (parameters `callingUid` and `callingPid`). However, all other parameters are specific to the hooks' contexts, e.g., the hook in line 1 provides information about the Intent being broadcast and the app component that should receive this Intent (parameters `targetComp` through `targetPid`). Thus, the hooks support policies that use the rich middleware-specific semantics.

In general, all hooks support truncation automata as policies by either allowing the module to throw exceptions that terminate the control flow and that are returned to the caller of the Android API, or by explicitly requiring a boolean return value that indicates whether the hook truncates the control flow or not (line 1 in Listing 1). A subset of the hooks additionally supports edit automata policies, that is the module can modify or replace return values of the Android API function or modify/replace arguments that divert or affect the further control flow after the hook. For instance, the *LocationManagerService* hook in Listing 1 (line 2) allows the module to edit or replace the Location object that is returned to the app that requested the current device location.

5.2.3 Application Layer

At the application layer, our Android Security Framework provides a mechanism to inject access control hooks into apps themselves. This access control technique is based on the concept of *inlined reference monitors* (IRM) pioneered by Erlingsson and Schneider [14]. The basic idea is to rewrite an untrusted app such that the reference monitor is directly embedded into the app itself, yielding a "self-monitoring" app. Although using IRMs might seem counter-intuitive or redundant in our design, IRMs are the only way in Android's current app model to achieve privilege separation between the components within an app (e.g., ad libs [19]) or to enforce edit automata policies on file-system and network interfaces (e.g., HTTPS-everywhere). The former are DVM internal operations and the latter do not involve the middleware, but instead the app processes interact directly with the file-system and network API of the kernel, whose semantics are rather unsuitable for enforcing edit automata policies. Thus, until this app model has been retrofitted to enable a system-centric solution for such kind of policies, our design relies on IRM. ASF provides an instrumentation API that enables security modules to dynamically hook any Java function within an app's DVM. Hooked functions divert the control flow of the program to the reference monitor, which thereby not only gains access to all function arguments but can also modify or replace the function's return value. Furthermore, in contrast to the hooks placed in the Android middleware, application layer hooks are dynamic: Hooks are injected by directly modifying the target app's DVM memory when a new app process is started. This design enables security modules to dynamically create and remove hooks at runtime as well as to inject app-specific hooks.

5.3 Middleware Framework API

We elaborate now in more detail on our framework API and the interaction between modules and the security infrastructure. Since we use the existing LSM framework as is, we focus here on our newly introduced middleware security framework and refer to the kernel documentation [25] for details on the LSM API. The middleware framework API of our current implementation contains 168 callback functions; a full listing is provided in the long version of this paper [2]. This API can be broken down into the following categories:

Enforcement functions. These functions form the bulk of the API (136 methods) and are called by the framework whenever the enforcement hooks in system apps and services are triggered. Each hook has a corresponding callback function in the module API, which has the same method signature as the hook (cf. Listing 1) and which implements the policy decision logic for its hook.

Kernel Sub-Module Interface. To avoid policy-specific interfaces for the communication between middleware/application layer apps and the Kernel Sub-Module, we introduce a generic kernel module API as part of the middleware framework API. It allows apps and services a controlled access to Linux security modules. Each security module can implement this interface and internally translate the API calls to calls on the proprietary channel between the user-space and the Linux security module. Two particular challenges for establishing this interface were the self-contained security checks of the kernel module and the requirement that this interface is already available during system boot. To guarantee security, the kernel module is required to perform policy checks to verify that a user-space process is sufficiently privileged to issue commands to it. Additionally, the kernel module is called before the middleware framework can load any Middleware Sub-Module, e.g., it can be called by Zygote when spawning new app processes. To solve these challenges, our design avoids an additional layer of indirec-

tion (i.e., IPC) for communication with the kernel module and loads the interface implementations via the Java reflection API statically into the application framework when it is bootstrapped. This ensures that the calling processes communicate directly with the kernel module through our generic API and that the kernel module can be called independently of middleware services.

Life-cycle management. Every module must implement functions for life-cycle management, such as initialization or shutdown. This enables the framework to inform the module when the system has reached a state during the boot cycle from which on the module will be called or when the system shuts down. Modules should use these functions, e.g., to initiate their policy engines or to save internal states to persistent storage before the device turns off.

Event notifications. Event notification interfaces are used to propagate important system events to the module. For instance, modules should be immediately informed when an app was successfully installed, replaced, or removed. Although this information is usually propagated via a broadcast Intents, the time gap between package change and broadcast delivery might cause inconsistencies in module states. Hence these events must be delivered synchronously.

Framework Callbacks. The framework provides modules a callback interface for communicating in a more direct manner with system services, such as the *PackageManagerService*, and avoids the need to go through the Android API. This is desirable for policy authors that want to leverage the middleware internal information. Our current callback interface, for instance, includes functions that allow modules to efficiently resolve PIDs to application package names.

Proprietary protocols. We introduced in our framework API a *callModule()* function that allows modules to implement proprietary communication protocols with other apps that are aware of this specific module, e.g., the front-end apps (cf. Section 5.1). When using *callModule()*, these protocols are based on *Bundles* and enable a protocol similar to the Parcel-based Binder IPC: apps serialize function arguments to a Bundle and add an identifier for the proprietary function the module should execute with the deserialized arguments. It is the task of the module to verify that the sender is sufficiently privileged to send commands.

IRM Instrumentation. The framework provides an instrumentation API that enables security modules to hook any Java function within selected app processes. To the best of our knowledge, ASF is the first solution for Android to provide a generic instrumentation API. Hooks injected via the instrumentation API are local to the app process that the API is called from. Therefore, all calls to the instrumentation API need to be performed from within a target app's process. We solve this by placing an instrumentation hook in the *ActivityManagerService* that is triggered when a new app process is about to be launched. A module that implements this hook has to return a Java class for the instrumentation logic that will be executed within the app's process. To ensure that this code is executed before control flow is passed to the app itself, we modify the arguments passed to Zygote to start this new app process via a special wrapper class that loads and executes the instrumentation code first.

5.4 Middleware Security Modules

We elaborate in more detail on the structure of security modules. Again, we use Linux security modules as is [25]

Figure 3: Middleware security module structure.

and, thus, focus here on the Middleware Sub-Module. A middleware module is simply a *Jar* file that is created with an Android SDK that includes our new security API. It is deployed to a protected location on the file system, from where it is loaded during boot. This Jar file contains all the module's code, resources, and manifest file (cf. Figure 3):

Module Manifest. The manifest (formatted in XML) declares properties such as the module author or code version, and, more importantly, the name of the main Java class that forms the entry point for the module.

Classes.dex. The *classes.dex* file contains, as in regular Android apps, the Java code compiled to *Dalvik executable bytecode* (DEX). It contains all Java classes that implement the security module's logic. During the load process of the Middleware Sub-Module, the middleware framework uses the Java reflection API to load the module's main class (as specified in the manifest) from *classes.dex*. To ensure that the reflection works error-free, the main class must implement the API as described in Section 5.3. Since the API defines currently more than a hundred methods, but a security module very likely requires only a subset of those, our SDK provides an abstract class that implements the API. That abstract class can be sub-classed by the module's main class, which then only needs to override the required functions. The abstract class returns for each non-overridden enforcement function an allow decision.

LSM interface. The proprietary interface between the user-space processes and the Linux security module in the kernel is implemented through a native library *liblsm.so* and a corresponding Java class *LSM.java*, which exposes the native library via the Java Native Interface. *LSM.java* has to implement the generic interface for the communication with the kernel that was explained in the previous section. The generic kernel module interface of ASF loads *LSM.java* through the Java reflection API into Android's application framework. This allows apps and services to communicate with the kernel module and avoids a policy-specific interface. We exemplified this mechanism by integrating SELinux through API into Zygote (cf. Section 6.2).

Resources. Each module can ship with proprietary resources, such as initial configuration files or required binaries. During module instantiation, the framework informs the module about the filesystem location of its Jar file, enabling the module to extract these resources on-demand from it.

5.5 Stackable and Dynamic Loadable Modules

Finally, two desirable properties for implementing an extensible security framework such as our ASF are dynamically loadable policies and policy composition (i.e., stacking modules). In the following we explain why we chose, in contrast to closest related work [20], to *permit* these features by design, but *not* consider them a requirement for our solution.

Dynamically Loadable Modules. Being able to dynamically load and unload modules is desirable, for instance, to speed up the development and testing cycles of modules and, in fact, we used this feature during the development of our example use-cases (cf. Section 6). However, the ar-

guments to support dynamically loadable modules beyond development (e.g., for security add-ons [20]) are disputed: First, dynamic loading is not always technically possible. A small set of static policy models, such as type enforcement [36, 8], require that all subjects and objects are labeled with a security context. Supporting such extensive labeling operations at runtime is an intricate problem. Second, there exist security considerations. The loading and unloading of modules must be strictly controlled to ensure that only integrity protected, trusted modules are loaded. Otherwise, given the privileges of modules, this would open the way to powerful malware modules. In our design we agree with the conclusions of the various Linux security module authors [11] and consider the drawbacks of dynamically loadable modules to outweigh their benefits. Therefore, we load the module *once during the system boot* and *permit* users of our framework to additionally activate dynamic unloading and loading of modules. But we currently do not consider this feature a requirement for our solution.

Stackable Modules. Composing the overall policy from multiple, simultaneously loaded and independent policies is a desirable feature, since usually no "one-size-fits-all" policy exists. Android, for instance, implements currently a quadruple-policy approach consisting of Permissions, SE Android type enforcement, AppOps, and Linux capabilities—each being responsible for a different aspect of the overall access control strategy. Multiple policies will naturally conflict and thus require the security framework to support different policy composition and reconciliation strategies (e.g., consensus or priority based) [32, 27]. However, supporting fully generic policy composition is quite a challenge and has been shown to be intractable [18]. Thus, despite its benefits, we decided in our design to follow the lessons learned by the LSM developers [40] and to only *permit* module developers to implement stackable modules, but we do not provide explicit interfaces for stacked modules in our framework infrastructure. In module combinations where policy consolidation is known to be feasible, the approach to stacking modules would be to provide a "composition module" that implements policy reconciliation and composition logic and which in turn can load other modules and multiplex API calls between them.

6. EXAMPLE SECURITY MODULES

In this section, we demonstrate the efficiency and effectiveness of our Android Security Framework by instantiating different security models from related work. To illustrate the versatility of ASF, we chose models from the areas of inlined reference monitoring, Android's default security architecture, and type enforcement. We present further instantiations of other security models in our long version [2].

6.1 AppOps and IntentFirewall

Google introduced (unofficially) with Android v4.3 the *AppOps* infrastructure for dynamic, more fine-grained Permissions. It added hooks in different system services and apps, which query a central AppOpsService whether an application is allowed to perform an operation (e.g., retrieving the location of the device or querying a ContentProvider). The AppOps rules define a mapping from UID/package name to allowed operations. AppOps offers an interface to apps to retrieve the current configuration. Additionally, Google introduced (again unofficially) an *IntentFirewall*, which acts as a reference monitor for certain Intent-based operations

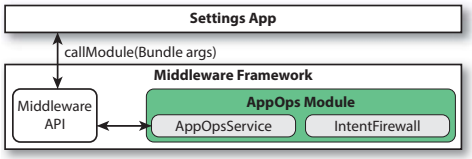


Figure 4: AppOps and IntentFirewall module

like starting an Activity. The IntentFirewall rules describe which caller is allowed to receive which kind of Intent object, using the Intent's attributes such as destination component.

Implementation as a module: We ported AppOps and IntentFirewall (from Android v4.3) to a security module for ASF (cf. Figure 4) by moving the AppOpsService and the IntentFirewall classes into a module. Our module comprises 2290 lines of code and differs in 33.71% of all LoC from the original implementation. The bulk of the changes (520 LoC), were required to move the hook logic of both services from the system apps and services of Android into the module by using our enforcement functions. For the IntentFirewall, this was straightforward and we only had to substitute a direct callback from IntentFirewall to the *ActivityManagerService* by our framework callback mechanism. For the AppOpsService, we had to add a mapping from caller PID to package name. By default the hooks of AppOps determine the caller's package name and pass this information to the AppOpsService for policy check. Since this is a policy-specific logic of the hooks, our framework hooks do not (by default) provide the caller's package name and we re-implemented this logic in our module by using our callback interface, which allows us to retrieve the package name for app PIDs. Moreover, we adapted the AppOpsService interface to retrieve/configure the current policies via a Bundle-based communication. AppOps is, furthermore, partially integrated into the *Settings* application to allow users to disable notifications from selected apps. We replaced this policy-specific channel between Settings and AppOps also with our policy-agnostic Bundle-based communication. Modules that support this Settings option, can return a value indicating whether notifications are disabled or not. If the module does not support this feature, Settings app by default allows notifications.

6.2 Type Enforcement [36, 8]

SE Android [36] brought SELinux type enforcement to the Android kernel and established the required user space support, e.g., it extended Zygote to label new app processes with a security type. FlaskDroid [8], developed for Android v4.0.3, extends SE Android's type enforcement to Android's middleware. Building on SEAndroid's kernel and low-level patches, it adds policy-specific hooks as policy enforcement points to various system services and apps in Android's middleware. The policy decisions at kernel level are made by the SELinux kernel module, while the decisions at middleware are made centrally in a policy server service. Both policy decision points decide based on subject type, object type, and object class reported by the hooks at their respective layer whether control flows should be truncated or not.

Implementation as module: We realized type enforcement with our ASF by porting FlaskDroid[1] as a module (cf. Figure 5), in this context porting the currently hard-

[1]Source code retrieved from `http://www.flaskdroid.org/`

Existing solution	LoC of module policy engine	LoC added/removed/edited (total delta)
AppGuard [3]	5059	+828/-79/∘13 (18.18%)
AppOps / Intent Firewall	2290	+627/-106/∘39 (33.71%)
FlaskDroid [8]	4968	+749/-32/∘40 (16.53%)

Table 1: Effort of porting different security extensions as module on our Android Security Framework.

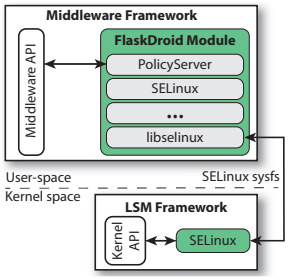


Figure 5: FlaskDroid module

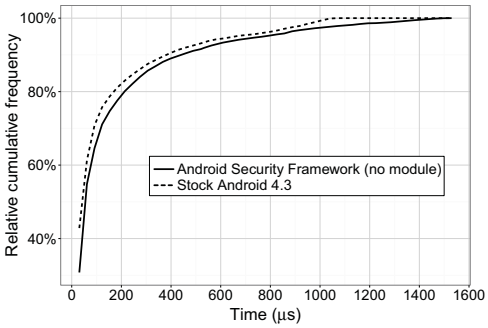


Figure 6: Cumulative frequency distribution of micro benchmarks in stock Android (dashed line) vs. ASF (solid line).

	Frequency	Mean (μs)
Stock Android 4.3	7320	116.182±4.550
ASF v4.3	6009	129.851±5.681

Table 2: Weighted average performance overhead of executing hooked functions in stock Android and in our ASF. The margin of error is given for the 95% confidence interval.

coded SELinux support of Android's middleware into an ASF module. At kernel level, we use the SE Android kernel and provide an SELinux-specific interface implementation for the kernel module. Further, we moved the middleware policy server and its dependencies into the middleware module. Using the enforcement functions of our API, we moved the policy-specific hook logic of FlaskDroid into the module as well. Additionally, we used SE Android's build system to label the file-system with security types.

Our port of FlaskDroid's middleware component as a security module consists of 4968 LoC (cf. Table 1) and differs in only 16.53% of all LoC from the original code. The bulk of these changes (550 LoC) is attributed to additions for implementing a mapping from the enforcement functions of our framework API to FlaskDroid's type checks. To confirm the correct enforcement of policies, we used the policies for middleware and kernel level that are provided with the FlaskDroid source code. Additionally, we noticed during our tests that the original implementation contains an error in assigning middlware security types to processes. Additional changes were necessary to fix this error in our module.

6.3 Inlined Reference Monitoring [3]

We use AppGuard [3] as the use-case to illustrate the applicability of our IRM instrumentation API, but similar application rewriting approaches [21] are also feasible. AppGuard is a privacy app for Android that enables end-users to enforce fine-grained access control policies on 3rd party apps by restricting their ability to access critical system resources. By injecting an IRM into apps, this approach supports security policies not easily enforceable by traditional external reference monitors in the Android middleware or kernel, e.g., to enforce the use of *https* over *http*.

Implementation as a module: We ported AppGuard[2] as a module for ASF by separating its privacy app into three components: We adapted the (1) AppGuard reference monitor with its dynamic hook placement and policy enforcement logic to use our IRM instrumentation API. The reference monitor is injected into selected app processes via our framework at app startup. The policy decision logic and persistent storage of policy settings was moved into (2) a middleware module. The middlware module selects the apps into which the IRM is injected. It also implements a Bundle-based communication protocol to exchange policy decisions and security events with the IRM component and with (3) a front-end app. The front-end app allows the user to adjust policy settings and to view logs of security-relevant events. We used the policies included in the original AppGuard implementation to confirm that policy enforcement by our module and by the original implementation are identical.

Our AppGuard security module consists of 5059 LoC in total (cf. Table 1), with 782 LoC residing in the middleware module and 4277 LoC in the IRM. Our module diverts in 18.18% of all LoC from the original code. The majority of the difference, 728 LoC, is attributed to moving the policy decision logic into the middleware module, while only 46 LoC were required to adapt the inlined reference monitor to use the provided instrumentation API.

[2]Source code provided by the original authors

7. EVALUATION AND DISCUSSION

We evaluate the performance of our ASF and discuss its current scope and prospective future work. An extended evaluation is provided in the long version of this paper [2].

7.1 Performance

Although the actual performance overhead strongly depends on the overhead imposed by the loaded module, we wanted to establish a baseline for the impact of our Android Security Framework on the system performance. The performance of LSM has been evaluated separately, e.g., for SEAndroid [36], and we are interested here in the effect of our new middleware security framework on the performance of instrumented middleware system services and apps.

Methodology. We implemented our ASF as a modification to the Android OS code base in version 4.3_r3.1 and used the Android Linux kernel in branch *android-omap-tuna-3.0-jb-mr1.1*. We performed micro-benchmarks for all execution paths on which a hook diverts the control flow to our middleware framework: We first measured the execution time of each hooked function with no security module loaded and allowing all access. Afterwards we repeated this test with hooks removed to measure the default performance of the same functions and thus operating identical to a stock Android. All our micro-benchmarks were performed on a standard Nexus 7 development tablet (Quad-core 1.51 GHz CPU and 2GB DDR3L RAM), which we booted and then used according to a testplan for different daily tasks such as browsing the Internet, sending text messages and e-mails, contacts management, or (un-)installing 3rd party apps.

Micro-benchmark results. Table 2 presents the number of measurements for each test case and their mean values. To eliminate extreme outliers, we excluded in both measurement series the highest decile of the measurements. For ASF the mean is the weighted mean value with consideration of the frequency of each single hook. In overall, our framework with no loaded module imposed with 129.851 μs approximately 11.8% overhead compared to stock Android. Figure 6 presents the relative cumulative frequency distribution of our measurements series and further illustrates this low performance overhead. Major contributor to this overhead is the marshalling, sending, and unmarshalling of the hooks' parameters. Thus, a future optimization of the baseline overhead would be a framework configuration that enables only the hooks that are actually used by the loaded module(s) and, hence, avoids irrelevant hook invocations.

7.2 Current Scope and Future Work

System setup. Certain security models require a preparatory system setup. For instance, type enforcement requires a pre-labelling of all subjects and objects. After the system has been setup, ASF supports modularization of these security models (cf. Section 6.2).

Module Integrity. As part of the kernel, the Kernel Sub-Module has the highest level of integrity. In contrast, the Middleware Sub-Module, as a user space process, can be circumvented or compromised by attacks against the underlying system (e.g., root exploits) and thus requires support by the kernel modules to prevent low-level privilege escalation attacks. Inlined reference monitors are inherently susceptible to attacks by malicious applications, because the reference monitor executes in the same process as the application that it monitors and no strong security boundary exists between the monitor and the app code. To remedy this situation, we are currently retrofitting Android's application model to combine the benefits of inlined and of system-centric reference monitors. By splitting apps into smaller units of trust (e.g., app components and ad libs), system-centric reference monitors are able to differentiate distinct trust levels within apps [31, 38, 35].

Completeness. It is crucial for the effectiveness of our security framework, that *all* access to security and privacy sensitive resources is mediated by the reference monitors. We consider it out of scope for this submission to formally verify the completeness of our prototype framework, but plan to use recent advances in static and dynamic analysis on Android to verify the placement of our hooks, similarly to how it was done for the LSM framework [12, 17].

Information flow control. Our framework provides modules with the control over which subject (e.g., app) has *access to* which objects (e.g., device location), but it cannot control how privileged subjects *distribute* this information. Controlling information flows is an orthogonal problem specifically addressed by different solutions [13, 33]. We plan to integrate such data flow solutions into our framework and to extend our security API with new generic calls for taint labeling and taint checking.

8. CONCLUSION

In this paper we presented the Android Security Framework (ASF), an extensible and policy-agnostic security infrastructure for Android. ASF allows security experts to develop Android security extensions against a novel Android security API and to deploy their solutions in form of modules or "security apps". Modularizing security extensions overcomes the current unsatisfactory situation that policy authors are either limited to one predetermined security model that is embedded in the Android software stack or that they are forced to confide in a security-model-specific Android fork instead of the mainline Android code base. Additionally, this modularization provides a number of benefits such as easier maintenance and direct comparison of security extensions. We demonstrated the effectiveness and efficiency of ASF by porting different security models from related work to ASF modules and by establishing a baseline for the impact of our infrastructure on the system performance.

Availability

The ASF source code and example modules can be retrieved from `http://infsec.cs.uni-saarland.de/projects/asf/`.

Acknowledgements

We thank the anonymous reviewers for their comments. 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).

9. REFERENCES

[1] M. D. Abrams, K. W. Eggers, L. J. LaPadula, and I. M. Olson. A generalized framework for access control: An informal description. In *NIST NCSC'90*, 1990.

[2] M. Backes, S. Bugiel, S. Gerling, and P. von Styp-Rekowsky. Android security framework: Enabling generic and extensible access control on android. Technical Report A/01/2014, Saarland University, April 2014.

[3] M. Backes, S. Gerling, C. Hammer, M. Maffei, and P. von Styp-Rekowsky. Appguard - enforcing user requirements on Android apps. In *TACAS'13*, 2013.

[4] L. Badger, D. F. Sterne, D. L. Sherman, K. M. Walker, and S. A. Haghighat. Practical domain and type enforcement for UNIX. In *IEEE SP'95*. IEEE, 1995.

[5] D. B. Baker. Fortresses built upon sand. In *NSPW'96*. ACM, 1996.

[6] S. Bugiel, L. Davi, A. Dmitrienko, T. Fischer, A.-R. Sadeghi, and B. Shastry. Towards taming

privilege-escalation attacks on Android. In *NDSS'12*. The Internet Society, 2012.

[7] S. Bugiel, L. Davi, A. Dmitrienko, S. Heuser, A.-R. Sadeghi, and B. Shastry. Practical and lightweight domain isolation on Android. In *SPSM '11*. ACM, 2011.

[8] S. Bugiel, S. Heuser, and A.-R. Sadeghi. Flexible and fine-grained mandatory access control on Android for diverse security and privacy policies. In *USENIX Security'13*. USENIX, 2013.

[9] E. Chin, A. Porter Felt, K. Greenwood, and D. Wagner. Analyzing inter-application communication in Android. In *MobiSys'11*. ACM, 2011.

[10] M. Conti, V. T. N. Nguyen, and B. Crispo. CRePE: Context-related policy enforcement for android. In *ISC'10*. Springer, 2010.

[11] J. Edge. The return of loadable security modules? Online: `http://lwn.net/Articles/526983/`, Nov. 2012.

[12] A. Edwards, T. Jaeger, and X. Zhang. Runtime verification of authorization hook placement for the Linux security modules framework. In *CCS'02*. ACM, 2002.

[13] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *OSDI'10*. USENIX, 2010.

[14] Ú. Erlingsson and F. B. Schneider. IRM enforcement of Java stack inspection. In *IEEE SP'00*. IEEE, 2000.

[15] T. Fraser. LOMAC: MAC you can live with. In *USENIX ATC'01*. USENIX, 2001.

[16] T. Fraser, L. Badger, and M. Feldman. Hardening COTS software with generic software wrappers. In *IEEE SP'99*, 1999.

[17] V. Ganapathy, T. Jaeger, and S. Jha. Automatic placement of authorization hooks in the Linux Security Modules framework. In *CCS'05*. ACM, 2005.

[18] V. Gligor, S. Gavrila, and D. Ferraiolo. On the formal definition of separation-of-duty policies and their composition. In *IEEE SP'98*. IEEE, 1998.

[19] M. Grace, W. Zhou, X. Jiang, and A.-R. Sadeghi. Unsafe exposure analysis of mobile in-app advertisements. In *WISEC'12*. ACM, 2012.

[20] S. Heuser, A. Nadkarni, W. Enck, and A.-R. Sadeghi. Asm: A programmable interface for extending android security. Technical Report TUD-CS-2014-0063, Intel CRI-SC at TU Darmstadt, North Carolina State University, CASED / TU Darmstadt, Mar. 2014. To appear at USENIX Security'14.

[21] J. Jeon, K. K. Micinski, J. A. Vaughan, A. Fogel, N. Reddy, J. S. Foster, and T. Millstein. Dr. Android and Mr. Hide: Fine-grained security policies on unmodified Android. In *SPSM '12*. ACM, 2012.

[22] B. W. Lampson. Protection. *ACM SIGOPS Operating Systems Review*, 8(1):18–24, Jan. 1974.

[23] J. Ligatti, L. Bauer, and D. Walker. Edit automata: Enforcement mechanisms for run-time security policies. *International Journal of Information Security*, 4(1–2):2–16, 2005.

[24] T. A. Linden. Operating system structures to support security and reliable software. *ACM Computer Surveys*, 8(4):409–445, Dec. 1976.

[25] Linux Cross Reference. Linux Security Module framework. Online: `http://lxr.free-electrons.com/source/Documentation/security/LSM.txt`.

[26] P. A. Loscocco, S. D. Smalley, P. A. Muckelbauer, R. C. Taylor, S. J. Turner, and J. F. Farrell. The inevitability of failure: The flawed assumption of security in modern computing environments. In *NISSC'98*, 1998.

[27] P. McDaniel and A. Prakash. Methods and limitations of security policy reconciliation. In *IEEE SP'02*. IEEE, 2002.

[28] M. Ongtang, S. E. McLaughlin, W. Enck, and P. McDaniel. Semantically rich application-centric security in Android. In *ACSAC'09*. ACM, 2009.

[29] A. Porter Felt, H. J. Wang, A. Moshchuk, S. Hanna, and E. Chin. Permission re-delegation: Attacks and defenses. In *USENIX Security'11*. USENIX, 2011.

[30] N. Provos. Improving host security with system call policies. In *USENIX Security'03*. USENIX, 2003.

[31] N. Provos, M. Friedl, and P. Honeyman. Preventing privilege escalation. In *USENIX Security'03*. USENIX, 2003.

[32] V. Rao and T. Jaeger. Dynamic mandatory access control for multiple stakeholders. In *SACMAT'09*. ACM, 2009.

[33] G. Russello, M. Conti, B. Crispo, and E. Fernandes. MOSES: supporting operation modes on smartphones. In *SACMAT'12*. ACM, 2012.

[34] J. Saltzer and M. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, 1975.

[35] S. Shekhar, M. Dietz, and D. S. Wallach. Adsplit: Separating smartphone advertising from applications. In *USENIX Security'12*. USENIX, 2012.

[36] S. Smalley and R. Craig. Security Enhanced (SE) Android: Bringing Flexible MAC to Android. In *NDSS'13*. The Internet Society, 2013.

[37] R. Spencer, S. Smalley, P. Loscocco, M. Hibler, D. Andersen, and J. Lepreau. The Flask security architecture: System support for diverse security policies. In *USENIX Security'99*. USENIX, 1999.

[38] Y. Wang, S. Hariharan, C. Zhao, J. Liu, and W. Du. Compac: Enforce component-level access control in Android. In *CODASPY'14*. ACM, 2014.

[39] R. Watson, W. Morrison, C. Vance, and B. Feldman. The TrustedBSD MAC Framework: Extensible kernel access control for FreeBSD 5.0. In *USENIX ATC'03*. USENIX, 2003.

[40] C. Wright, C. Cowan, S. Smalley, J. Morris, and G. Kroah-Hartman. Linux Security Modules: General security support for the Linux kernel. In *USENIX Security'02*. USENIX, 2002.

[41] Y. Zhou and X. Jiang. Dissecting Android malware: Characterization and evolution. In *IEEE SP'12*, 2012.

[42] Y. Zhou, X. Zhang, X. Jiang, and V. Freeh. Taming information-stealing smartphone applications (on Android). In *TRUST'11*. Springer, 2011.

Towards a Scalable Resource-driven Approach for Detecting Repackaged Android Applications

Yuru Shao[†], Xiapu Luo[†‡*], Chenxiong Qian[†], Pengfei Zhu[†], and Lei Zhang[†]
Department of Computing, The Hong Kong Polytechnic University[†]
The Hong Kong Polytechnic University Shenzhen Research Institute[‡]
shaoyuru@gmail.com, {csxluo, cscqian, cspzhu,cslzhang}@comp.polyu.edu.hk

ABSTRACT

Repackaged Android applications (or simply apps) are one of the major sources of mobile malware and also an important cause of severe revenue loss to app developers. Although a number of solutions have been proposed to detect repackaged apps, the majority of them heavily rely on code analysis, thus suffering from two limitations: (1) poor scalability due to the billion opcode problem; (2) unreliability to code obfuscation/app hardening techniques. In this paper, we explore an alternative approach that exploits core resources, which have close relationships with codes, to detect repackaged apps. More precisely, we define new features for characterizing apps, investigate two kinds of algorithms for searching similar apps, and propose a two-stage methodology to speed up the detection. We realize our approach in a system named ResDroid and conduct large scale evaluation on it. The results show that ResDroid can identify repackaged apps efficiently and effectively even if they are protected by obfuscation or hardening systems.

1. INTRODUCTION

Repackaged apps have been one of the major sources of mobile malware on Android for many years [16,18]. A recent study showed that 86% malware samples were repackaged version of legitimate apps [48]. BitDefender even found that 1.2% of apps on Google Play have been repackaged to deliver ads and collect information [2]. Apps repackaging has also become a major threat to app economy [9]. By modifying the embedded ad's client ID or replacing it with new ad libraries, an attacker can make profits through apps developed by others [24]. As another example, repackaging paid apps and uploading the modified versions to third-party markets will result in revenue losses to developers. Moreover, repackaged financial apps not only cause financial loss to customers [31], but also compromise companies' reputation and users' experiences [9].

*The corresponding author

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ACSAC '14, December 08 - 12 2014, New Orleans, LA, USA
Copyright 2014 ACM 978-1-4503-3005-3/14/12 ...$15.00.
http://dx.doi.org/10.1145/2664243.2664275

Although a number of systems have been proposed to identify repackaged apps, how to effectively and efficiently detect them remains a challenging problem. One possible reason is that the majority of existing systems heavily rely on codes' features to quantify the similarity between apps, such as diverse hash values [1, 27, 47], abstract syntax trees (AST) [37], control flow graph (CFG) and its variants [13, 22, 43], program dependency graph (PDG) [19, 20], etc., thus suffering from two limitations:

Poor scalability due to the billion opcode problem. Hanna et al. estimated that the total amount of opcodes in all apps is around 1.45 billion [27] not to mention the rapid increase in the number of new apps. Therefore, processing millions of apps in many Android markets demands scalable and efficient solutions.

Unreliability to code obfuscation/app hardening techniques. Since most of existing solutions borrowed methods from the area of code clone detection that has been studied for 20 years [11, 39, 41], evasion solutions are available to dedicated attackers, let alone new obfuscation and hardening techniques [17, 29, 43, 44].

We explore an alternative scheme, a novel resource-driven approach, to detect repackaged apps. Our approach is motivated by two observations. First, Android apps usually contain various resources, such as layout and style resources for graphical user interface (GUI). Although these resources are separated from the executable `.dex` file, there are close relationships between resources and codes. Second, attackers seldom modify original resources in order to let the repackaged apps work properly and have the same look-and-feel. Moreover, existing obfuscation/hardening systems rarely handle resources. Although not all resources are critical to an app, some core resources cannot be easily modified by attackers without affecting the app's functionality or quality-of-experience (QoE). Therefore, we exploit core resources and the related codes to quantify apps' similarity.

To scale up this approach, we propose a two-stage methodology for grouping similar apps. More precisely, statistical features are used in the first stage to quickly divide apps into groups and then at the second stage structural features are employed to further cluster apps in each group. Note that the second stage can be performed in parallel. Since the methodology is general, we investigate the performance of two kinds of algorithms to search for similar apps: nearest neighbor search (NNS) [8] and clustering [7]. The former may quickly locate apps that are very similar to a target app but may miss other similar apps. The latter can partition apps into different clusters from a global view but has higher

computational complexity.

We have realized our approach in a system named ResDroid and conducted extensive evaluation on it. The experimental results not only validate its effectiveness and efficiency but also reveal interesting observations. In summary, this paper makes the following contributions:

1. We propose a novel resource-driven approach to detect repackaged apps. To our best knowledge, it is the first systematic examination on leveraging resources for repackaged apps detection.

2. We propose a two-stage methodology to scale up the approach and investigate the performance of two kinds of algorithms for searching similar apps: nearest neighbor searching and clustering.

3. We realize our approach in a system named ResDroid with 2770 lines of Python code, 1157 lines of Java code, and 309 lines of C code. Although a simultaneous research, ViewDroid [45], also adopts GUI-related features, there are significant differences between it and ResDroid in terms of feature selection, comparison algorithms, and scalability, as detailed in Section 7.

4. We conduct the first study on the effect of commercial app hardening systems on detecting repackaged apps. We also develop DexDumper for dynamically dumping hardened apps from memory for detection.

5. We conduct extensive evaluation on ResDroid with 169,352 apps crawled from 10 markets and 200 real repackaged apps. The results show that ResDroid can detect repackaged apps effectively and efficiently.

The rest of this paper is organized as follows. Section 2 introduces the problem and background knowledge. Section 3 and Section 4 detail our methodology and the implementation of ResDroid, respectively. The experimental results are reported in Section 5. Section 6 discusses ResDroid's limitations and future work. After introducing the related work in Section 7, we conclude the paper in Section 8.

```
1  MainActivity.java:
2  public class MainActivity extends Activity {
3      protected void onCreate(Bundle savedInstanceState) {
4          super.onCreate(savedInstanceState);
5          setContentView(R.layout.main_act_layout);
6          ...
7      }
8  }
9
10 TextFragment.java:
11 public class TextFragment extends Fragment {
12     public View onCreateView(LayoutInflater inflater,
           ViewGroup container, Bundle ...) {
13
14         View view = inflater.inflate(
15             R.layout.text_fragment, container, false);
16
17         return view;
18     }
19 }
20
21 MyEditText.java:
22 public class MyEditText extends EditText {
23     ...
24 }
25
26 main_act_layout.xml:
27 <RelativeLayout ... >
28     <fragment
29         android:id="@+id/text_fragment"
30         android:name="com.example.TextFragment" />
31 </RelativeLayout>
32
33 text_fragment.xml:
34 <RelativeLayout ... >
35     <include layout="@layout/titlebar" />
36     <TextView
37         android:id="@+id/text_view1"
38         android:text="@string/tv_text"
39         ... />
40     <com.example.MyEditText
41         android:id="@+id/edit_text"
42         ... />
43 </RelativeLayout>
44
45 my_editor.xml:
46 <view
47     class="com.example.MyEditText"
48     ... />
```

Figure 1: Example of the interaction between app code and resources.

2. BACKGROUND

2.1 Problem Statement

The majority of existing approaches heavily depend on code-level features, thus suffering from two limitations: (1) poor scalability to process billions of opcodes; (2) unreliability to code obfuscation/app hardening techniques. The goal of this paper is to explore an alternative scheme, a resource-driven approach, to detect repackaged apps. Motivated by the observation that existing attacks and code obfuscation/app hardening techniques seldom handle resources, we investigate how to employ resources to detect repackaged apps from four aspects, including, feature selection, feature extraction, scalable approaches for searching similar apps, and limitations. Note that the new approach complements the existing code-level systems instead of replacing them.

2.2 Application Resources

Android developers are recommended to externalize resources from the codes so that they can be maintained independently [6]. Figure 1 shows an example of app codes and resources. `MainActivity`.java defines an activity that provides users a GUI for interaction. When an activity is launched, the Android runtime creates an activity object and invokes the `onCreate` method defined at lines 3-7.

An activity may contain other GUI components, which can be added to its `View` dynamically or defined by a layout file (in XML format). Lines 27-31 define the layout used by `MainActivity`. Line 5 indicates that `MainActivity` loads its view from main_act_layout.xml.

`Fragment` is a special component that represents a behavior or a portion of user interface within an activity. It can be considered as a modular section of an activity, which has its own layout. main_act_layout.xml defines a fragment associated with the class com.example.TextFramgment, whose content is shown in lines 10-19. When `MainActivity` is started, its layout file main_act_layout.xml will be loaded and the activity will present users the fragment, whose layout is defined in text_fragment.xml and loaded in line 14 via the `inflate` method. In this example, the final GUI presented to users is actually the view of the fragment defined in text_fragment.xml.

The GUI component text_fragment (lines 40-42) is a customized component com.example.MyEditText. It is defined in my_editor.xml (lines 46-48), and the corresponding codes are in com.example.MyEditText.java, as specified in line 47.

When interacting with an app, users can navigate between (i.e., transition between) different activities [10]. We define the transition among activities as *activity transition graph* (ATG), where each vertex represents an activity and an edge indicates the existence of transitions through Android methods startActivity() or startActivityForResult().

App resources are referenced through IDs. For example, after main_act_layout.xml is parsed, an ID will be signed to it. The auto-generated file `R.java` records all resources and their IDs. App codes access the resources through their IDs. For example, `MainActivity` sets its view defined in main_act_layout.xml by referencing `R.layout.main_act_layout`. Moreover, developers can use the scheme `@type/name` to reference other resources. For instance, line 38 references a string resource named "tv_text" using `@string/tv_text`.

2.3 Event Handlers

There are two kinds of event handlers in Android apps.

GUI event handlers. GUI objects can be associated with event handlers. For example, given a button, the execution will reach its `onClick` function, defined in the interface android.view.View.OnClickListener, after a user clicks it.

Lifecycle event handers. An activity instance may transition among different states in its lifecycle [5]. Developers can define how an activity behaves when it transitions from one state to another in callback methods. For example, when an activity is started, its `onCreate` and `onStart` callback methods will be invoked successively.

For apps without GUI/activities, we consider callbacks in their Services and Broadcast Receivers as event handlers.

3. METHODOLOGY

3.1 Overview

Figure. 2 depicts our resource-driven solution for detecting repackaged apps, which is realized in ResDroid and the implementation is detailed in Section 4.

The feature extraction module first identifies an app's major packages according to their importance measured by the PageRank algorithm [34] (Section 3.3). Then, core resources along with their statistical features and structural features will be determined according to major packages and the app's manifest file. The statistical features (Section 3.4) are lightweight in terms of computation and comparison but may not provide precise information about an app. In contrast, structural features (Section 3.5) can better characterize an app at the cost of the complexity of computation and comparison.

To scale up the detection for millions of apps, we propose a two-stage methodology that employs the divide-and-conquer strategy to identify similar apps within small groups of apps. More precisely, ResDroid first uses statistical features to divide apps into small classes in the coarse-grained processing module and then employs structural features to identify similar apps within each group in the fine-grained processing module. The output is a set of *potential repackaging groups* (PR-Groups) containing suspicious repackaged apps. Our approach is rational because repackaged apps are usually similar to original apps and the process of clustering apps in different groups can be parallelled.

Finally, ResDroid verifies whether those similar apps are repackaged apps according to their signatures. Since the percentage of suspicious apps is usually small, they could be further inspected by in-depth code analysis systems [19], dynamic analysis systems [23, 38], malware detection systems [25], or even manual verification if necessary.

Based on the selected features, our solution can adopt different algorithms to find similar apps. Here, we examine the performance of two kinds of popular algorithms: clustering algorithms and nearest neighbor search (NNS) algorithms in Section 3.6 and Section 3.7, respectively.

3.2 Hardening Detection

To secure Android apps, hardening techniques and services have emerged [3] (e.g., Bangcle[1], iJiaMi[2]). Typically, they encrypt an app's `classes.dex` and load it into memory through java native interfaces (JNI). Hardening not only raises the bar for attackers to repackage apps, but also thwarts ResDroid to extract apps' features. To tackle this problem, we design and implement DexDumper (Section 4.1) to dynamically restore the `classes.dex`.

Before extracting features, ResDroid will check whether the app is hardened or not by looking for patterns of hardening services. For example, since Bangcle inserts a shared library `libsecexe.so` into hardened apps, an app containing such file is considered as a hardened app. Moreover, ResDroid will invoke DexDumper to dump the original `classes.dex` from a hardened app for feature extraction.

3.3 Major Packages and Core Resources

Not all resources are critical to an app and/or have close relationship with major codes. To raise the bar for an attacker to evade the detection by modifying features, we define core resources, which are used by major packages, with the following requirements: (1) it should be difficult for an attacker to manipulate these resources. In particular, random manipulations by an attacker will impair an app's functionality and/or QoE, or such manipulations can be easily filtered out; (2) they are representative.

Major packages refer to important codes in an app excluding imported libraries. We created a blacklist to filter out frequently-used ad and third-party libraries. Motivated by the module decoupling technique in [46], ResDroid first constructs a package dependency graph, an undirected and weighted graph, where each vertex represents a package, and an edge between two vertexes indicates the existence of method invocations between them. The weight of an edge is increased by one if it spots a method invocation between the two packages. Then, ResDroid ranks the packages using the PageRank algorithm [34] and selects the top 5 packages as major packages.

3.4 Statistical Features

3.4.1 Definition

We define 15 statistical features, which can be easily retrieved, and use them in the coarse-gained processing module. Let $\mathcal{A} = \{a^{(i)}\}_{i=1...N}$ be a set of apps. The statistical features of each sample $a^{(i)}$ is represented as a vector

[1] `http://www.bangcle.com`
[2] `http://www.ijiami.cn`

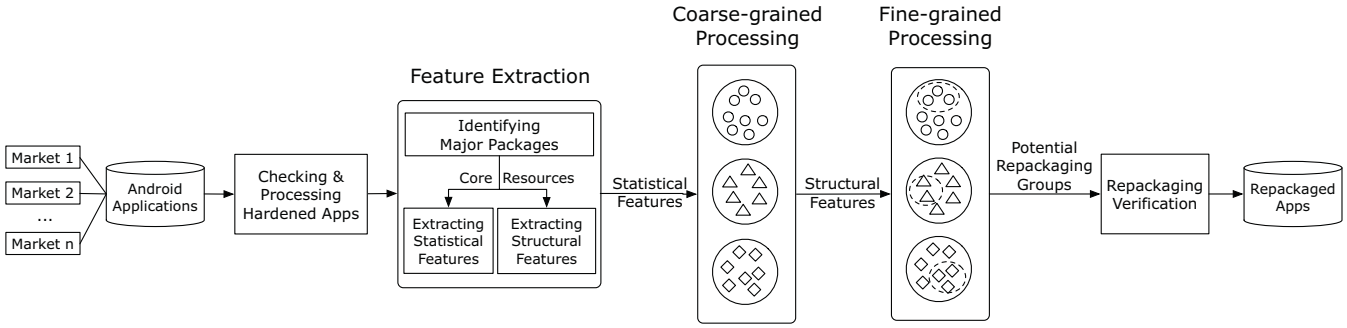


Figure 2: The procedure of our resource-driven approach.

$v^{(i)} = (f_1^{(i)}, ..., f_{15}^{(i)})$. The first five dimensions $(f_1^{(i)}, ..., f_5^{(i)})$ include (1) number of activities; (2) number of permissions, both system-defined and custom ones are included; (3) number of intent filters; (4) average number of `.png` files per `drawable*` directory; and (5) average number of `.xml` files per directory in `res`. The next 10 dimensions $(f_6^{(i)}, ..., f_{15}^{(i)})$ include the average number of references to the 10 most-referenced resources. Table 1 lists the top resources in 24,810 randomly selected apps. More features, such as number of services, can be added to better profile an app without dramatically increasing the complexity.

Note that we use the average number of `.png` and `.xml` files instead of the total amount, because they are more representative. For example, apps that support multiple display resolutions will have several set of drawables. For example, `drawable-hdpi` contains bitmap drawables for high screen densities while `drawable-ldpi` has objects for low-density displays. As another example, files in `values-zh` and `values-fr` provide Chinese and French supports, respectively. If an app is repackaged with extra language support, the total number of certain kinds of resources will increase but the average values will not be affected.

#	Type	Total	Average
1	id	6,631,598	267
2	drawable	3,616,963	146
3	string	1,964,378	79
4	color	1,024,896	41
5	style	823,848	33
6	dimen	623,906	25
7	layout	248,097	10
8	xml	136,730	6
9	integer	88,130	4
10	array	76,670	3

Table 1: 10 most referenced resource types in 24,810 apps randomly selected from our dataset.

3.4.2 Comparison

Since the features' ranges are quite different, we normalize $v^{(i)}$ and get a new vector $v_n^{(i)} = (F_1^{(i)}, ..., F_{15}^{(i)})$, where each dimension ranges in $[0, 1]$. We calculate $F_j^{(i)}$ using the following function:

$$F_j^{(i)} = norm(f_j^{(i)}) = \sqrt{\frac{f_j^{(i)} - min(f_j^{(1...N)})}{max(f_j^{(1...N)}) - min(f_j^{(1...N)})}} \quad (1)$$

Note that $min(f_j^{(1...N)})$ and $max(f_j^{(1...N)})$ are the minimal and maximal values of the jth feature for all apps in $\mathcal{A}$. The similarity between two apps $a^{(k)}$ and $a^{(h)}$ according to the statistical features is defined as:

$$s^{(k,h)} = e^{-D_c(v_n^{(k)}, v_n^{(h)})}, \quad (2)$$

where $D_c(v_n^{(k)}, v_n^{(h)})$ is the *Euclidean distance* of the two normalized feature vectors.

3.5 Structural Features

The structural features cover two types of information: (1) activity layout; (2) event handler. These features are reliable and representative because: (1) repackaged apps usually have the same GUI as the original apps to avoid affecting their functionality and QoE; (2) although an dedicated attacker can re-implement an existing layout, it takes time to achieve the completely same appearance and it is difficult to apply this process to all apps automatically; (3) since GUI objects and the associated *event handlers* determine the functionality of an app, attackers usually keep existing event handlers to avoid impairing the original app's functionality. Therefore, ResDroid compares two apps' event handlers instead of all codes.

As an app may have multiple activity layouts and different kinds of event handlers, features extracted from a single layout or event handlers should be well organized to form the final structural features. To achieve this goal, for each app $a^{(i)} \in \mathcal{A}$, we first perform static analysis to construct its ATG and then traverse the ATG using depth-first search algorithm starting from the main activity of $a^{(i)}$. If an app has no main activity, we choose the first activity defined in its `AndroidManifest.xml`. We remove the activities that do not belong to core resources from the generated activity sequence $AS(a^{(i)})$. The following layout features (Section 3.5.1) and event handler features (Section 3.5.2) are arranged with the order defined in $AS(a^{(i)})$.

3.5.1 Activity Layout Feature

For an activity act, we denote its layout as $L(act)$. We traverse $L(act)$ using the pre-order traversing algorithm and obtain an element sequence $\langle e_1, ..., e_m \rangle$, where e_i $(i = 1, ..., m)$

stands for a GUI object, such as `Button`, `EditText`. We define a function α for mapping e_i to an English letter ('a' to 'z', 'A' to 'Z') according to its type. For example, `Button` is labelled as 'b', `EditText` as 'e', and `View` as 'v', etc. Since customized components are usually derived from existing components and their names may be obfuscated, we use their ancestors' type to label them. For instance, a `PhotoView` extending `View` is also labeled as 'v'.

Following this rule, $L(act)$ is converted into a sequence $LS(act) = \langle \alpha(e_1), ..., \alpha(e_m) \rangle$. For an app $a^{(i)} \in \mathcal{A}$ and $AS(a^{(i)}) = \langle act_1, ..., act_n \rangle$, its layout feature will be:

$$LF(a^{(i)}) = \langle LS(act_1), ..., LS(act_n) \rangle \tag{3}$$

3.5.2 Event Handler Feature

Event handler feature refers to the corresponding callback methods. We perform static analysis on each event handler and extract their *method signatures* [21]. Let $\sigma(m)$ denote the signature of an event handler m. Since an activity act may have several event handlers, we represent its event handler feature as a sequence that consists of method signatures in *lexicographical order*: $EH(act) = \langle \sigma(m_1), ..., \sigma(m_k) \rangle$. Given an app $a^{(i)} \in \mathcal{A}$ and $AS(a^{(i)}) = \langle act_1, ..., act_n \rangle$, its event handler feature $EF(a^{(i)})$ is:

$$EF(a^{(i)}) = \langle EH(act_1), ..., EH(act_n) \rangle \tag{4}$$

3.5.3 Comparison

When quantifying the similarity between two apps $a^{(k)}$ and $a^{(h)}$ using the structure features, we define the distance between $a^{(k)}$ and $a^{(h)}$ as:

$$\begin{aligned} D_f(a^{(k)}, a^{(h)}) = w_l * d_s\Big(LF(a^{(k)}), LF(a^{(h)})\Big) \\ + w_e * d_s\Big(EF(a^{(k)}), EF(a^{(h)})\Big), \end{aligned} \tag{5}$$

where w_l and w_e are pre-defined weights (the actual values assigned to w_l and w_e are discussed in Section 5). $d_s(s_1, s_2)$ measures the distance between two sequence s_1 and s_2, which is defined as:

$$d_s(s_1, s_2) = 1 - \frac{length(LCS(s_1, s_2))}{min(length(s_1),\ length(s_2))}, \tag{6}$$

where $LCS(s_1, s_2)$ is the *longest common sequence* of s_1 and s_2, and $length(s)$ is the length of sequence s. The advantage of LCS is that even if noise is inserted into one sequence the final output of d_s may remain unchanged. Therefore, even though an attacker may insert junk resources to change an app's structural features, ResDroid will not be affected.

3.6 Clustering-based Processing

We adopt the *spectral clustering* [7] algorithm to cluster apps according to the normalized statistical features. Spectral clustering techniques leverage the spectrum (eigenvalues) of the similarity matrix of the input data to perform dimensionality reduction before performing the clustering. It allows us to add more features for further improving the performance in future work.

We apply the complete-linkage *hierarchical clustering* algorithm and the DB cluster validity index [26] to split each coarse-grained cluster into fine-grained clusters (i.e., PR-Groups). In hierarchical clustering, the linkage criterion determines the distance between sets of observations. We chose the complete-linkage criterion because it usually results in compact clusters with small diameters. Although it may be sensitive to outliers, we can filter out outliers before conducting the hierarchical clustering.

3.7 NNS-based Processing

Clustering algorithms may compare every pair of apps, thus leading to high computation complexity. NNS algorithms allow us to only compare apps that are likely to be similar and therefore dramatically reduce the comparisons. When applying NNS algorithms, we adopt the k-d tree technique [35]. In the coarse-grained processing, we select candidate app pairs with the help of k-d tree. In the fine-grained processing we compare apps in pair to obtain PR-Groups.

3.7.1 Selecting Candidate Pairs

We first build a k-d tree and insert statistical feature vectors $v^{(i)}$ $(i = 1, ..., N)$ to this tree ($k = 15$ because of the 15 dimensions). Since different apps may have identical statistical features, we cannot insert both of them into the k-d tree for the sake of avoiding duplicated nodes. For example, given two apps $a^{(i)}$ and $a^{(j)}$ with the same feature vectors (i.e., $v^{(i)}$ and $v^{(j)}$), if $a^{(i)}$ is in the k-d tree, we cannot insert $a^{(j)}$. To solve this problem, we sort the statistical feature vectors of all apps according to the *lexicographical order* and then group apps having identical feature vectors. As shown in Figure 3, $v^{(5635)}$ and $v^{(9440)}$ are identical, and therefore we group $a^{(5635)}$ and $a^{(9440)}$ into the same cluster c_n. Then, we insert clusters containing one or more apps into the k-d tree, where each node represents a cluster. When we query n nearest neighbors for each cluster C_i, we regard C_i and its neighbors as candidate cluster pairs. For two clusters C_i and C_j, the candidate app pairs include: (1) app pairs in C_i; (2) app pairs in C_j; and (3) app pairs between C_i and C_j.

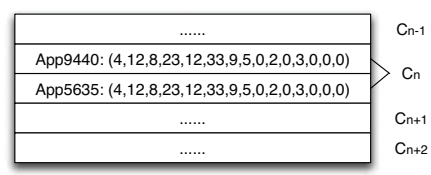


Figure 3: Statistical feature vectors are sorted in Lexicographical order. Apps that have identical feature vectors are grouped in the same cluster.

3.7.2 Pair-wise Comparison

The distance between two apps $a^{(k)}$ and $a^{(h)}$ is measured by D_f in Eqn.(5). Given a threshold θ, if $D_f(a^{(k)}, a^{(h)})$ is smaller than θ, they are in the same PR-Group.

3.8 Repackaging Verification

Apps in PR-Groups have similar appearance and functionalities. However, since developers may create a bunch of similar apps that cannot be considered as repackaged apps, we perform a verification on apps in each RP-Groups. More precisely, we extract developer certificates from apps and distinguish each certificate with their MD5 checksum. If all apps in a PR-Group share the same certificate, they are not repackaged. Otherwise, apps repackaging is detected.

4. IMPLEMENTATION

We have implemented Resdroid with 2770 lines of Python code, 1157 lines of Java code, and 309 lines of C code.

4.1 DexDumper

To handle hardened apps whose original `classes.dex` is encrypted or hidden in shared libraries, we design and implement *DexDumper* that dumps the original `classes.dex` of a hardened app from memory during runtime. It is motivated by the observation that the Dalvik virtual machine (DVM) cannot run encrypted `classes.dex` and therefore the original `classes.dex` will be restored before being executed.

DexDumper first invokes the `ptrace` system call to attach to the process of a running hardened app, and then reads the app's memory and searchs for the dex files. As the memory space of a process is very large, DexDumper manages to narrow the searching scope to efficiently locate the target dex code. More precisely, it reads the process mapping file of the hardened app, namely `/proc/PID/maps` (`PID` stands for the app's process ID), where we can get the start address, the end address, and the attributes of each memory piece. Note that the memory where dex files are mapped into has attributes "r-xp", meaning that the area is readable (r), executable (x) and private (p). From memory pieces with such attributes, we can dump a set of `dex` format files since some pre-loaded runtime libraries (e.g., `ext.jar@classes.dex`) are also located.

For each dumped dex files, DexDumper looks for the app's package name (defined in the app's manifest) in their string constants pool. If found, it is considered as the original `classes.dex`. Since hardening will not alter the app's resources, ResDroid takes in the dumped dex file and the resources from the hardened apps for further process.

4.2 Feature Extraction

We employ *apktool* to decompile apps. Apktool can restore an app's resources and translate its `dex` code into `smali` format simultaneously. However, we found that apktool may crash when processing some apps. For such apps, we use *aapt* and *baksmali* to obtain their resources and smali code separately. To construct ATG, we use A3E [10] to perform static analysis on apps. It took around 22 hours to generate ATGs for all apps in our data set. Then we run PageRank to pick 5 major packages, which took another 3 hours. In total, the extraction of statistical features consumed 25 hours. Such a long period is not unexpected because both A3E and the PageRank algorithm are time-consuming. It is acceptable as we only need to do this computation once.

When extracting layout features of apps, we observed that 93.7% apps define their layouts in XML under the `res/layout` directory. Therefore, we can obtain their layout structures by traversing the XML files. Since some apps generate their layouts dynamically, we use GATOR (v1.0) [4, 40] to process them, which can construct an app's layouts from the codes by conducting static reference analysis for GUI objects. We further enhanced GATOR from two aspects. First, it does not handle certain GUI components (i.e., fragments and dialogs). As fragment is widely used to realize components reuse in different layouts, we added functions to handle it. Second, as GATOR only takes in source codes, we empowered it to accept dex files.

Similar to activities, layouts of fragments can be defined in static XML files or dynamically created at runtime. Therefore, we first identify fragment objects in activities, and then examine the fragment and check whether its layout is defined in resource file. If so, we directly parse the XML file to obtain its structure. Otherwise, we construct its layout structure by re-using the code logic designed for processing dynamically-generated activity layouts. The extraction of structural features took 49.7 hours.

4.3 Clustering and NNS

In coarse-grained clustering, we use a parallel implementation of spectral clustering [14], which can effectively handle large-scale data. Since the last step of spectral clustering is actually running *k-means*, the number of clusters should be specified. We use *x-means* [36] to estimate a proper number, instead of choosing it arbitrarily.

5. EXPERIMENTS

Our data set contains 169,352 apps crawled from 10 Android markets, including the offical market Google Play and 9 other third-party markets. Our experiments were conducted on a PC running Ubuntu Linux 12.04LTS with an 8-core Intel i7 3.50GHz CPU and 32GB memory.

5.1 Ground Truth

We use 200 pairs (400 apps) of repackaged and original apps as ground truth to evaluate ResDroid. All the repackaged apps are real malware or adware (121 from SandDroid[3] and other 79 from ContagioMobile[4]). The original apps were downloaded from Google Play. There are chances that different versions of the same app present dissimilar GUIs. Therefore, to make the ground truth more reliable, we only select original apps that have the nearest (or the same) version codes with the repackaged ones.

5.2 Clustering-based Approach

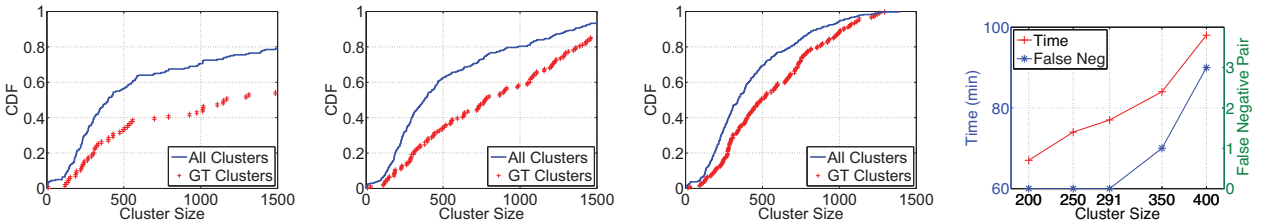
During the coarse-grained processing, the x-means algorithm [36] was used to estimate the number of clusters (denoted as C). The recommended value was 291. We also tried other values close to 291 (i.e., 200, 250, 350, and 400) when performing the spectral clustering. Figure 4 shows the CDFs of the sizes of clusters. With the increment of C, the average cluster size decreases. Figure 4(b) shows that when $C = 291$ nearly 10% clusters (around 20% groud truth clusters) contain more than 1,500 apps. As shown in Figure 4(c), when $C = 400$, all clusters contain less than 1,500 apps.

It is obvious that the smaller a cluster is, the more quickly (and hopefully more accurate) the hierarchy clustering will be finished. Furthermore, although we need to perform more clustering, it is easy to parallelize the tasks as they are independent. So a large C may be expected. However, as shown in Figure 4(d), more false negatives appear along with the increment of C. The false negative is 0 when $C = 291$, but it increases to 3 when $C = 400$. Moreover, a larger C requires longer time to perform the spectral clustering. Considering both cluster sizes and false negative rate, we finally chose 291 as the number of clusters in coarse-grained clustering.

We apply hierarchical clustering to each cluster produced by coarse-grained clustering to generate PR-Groups. In hierarchical clustering, we set the cutoff value to 0.1. That

[3] `http://sanddroid.xjtu.edu.cn`

[4] `http://contagiominidump.blogspot.com`

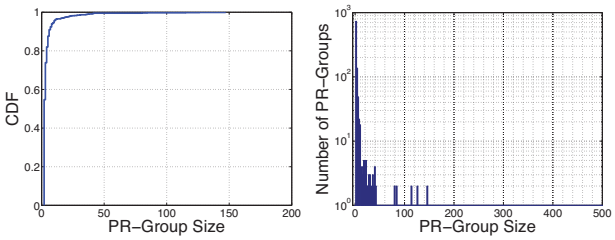


(a) $C = 200$ (b) $C = 291$ (c) $C = 400$ (d) Consumed time and false negatives with different C

Figure 4: CDFs of sizes of clusters produced by spectral (coarse-grained) clustering with different numbers of cluters

means, if two clusters have a distance less than or equal to 0.1, they will be merged into the same PR-Group.

1,605 PR-Groups are obtained and they include 6,906 apps in total. Figure 5 shows the distribution of the sizes of PR-Groups. Over 98% of PR-Groups have sizes smaller than 50. The average size of PR-Groups is 4.03, which is small enough for manually checking if need. Among all of our ground truth pairs, the repackaged app and the original app fell into the same PR-Groups, meaning that the false negative rate is 0.



(a) CDF (b) Histogram

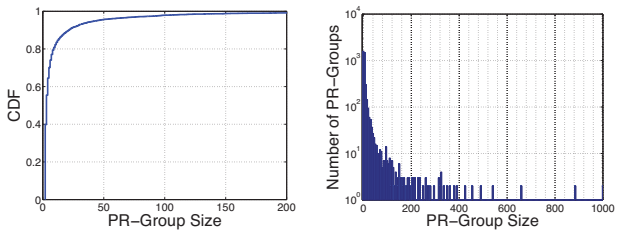
Figure 5: Distribution of the sizes of PR-Groups that are produced by hierarchical clustering with cutoff value 0.1.

5.3 NNS-based Approach

We first build a k-d tree according to statistical features and query nearest neighbors of each tree node for selecting candidate app pairs. More precisely, we set the number of neighbors (n) to 10. Then, we compare their structural features and calculate their distance following Eqn.(5). Note that w_l and w_e can be adjusted to support different criteria. For example, if $w_l > w_e$, more emphasis is paid to the layout features. Otherwise, the event handler features may be regarded as more important. In our experiments, $w_l = w_e = 0.5$. The distance threshold θ was set to 0.15, meaning that if the distance between two apps is smaller than 0.15 they will be classified into the same PR-Group.

The total number of PR-Groups produced by NNS-based approach is 2,070, including 20,867 apps. The average size of PR-Groups is 9.9. Figure 6 illustrates the distribution of the sizes of PR-Groups. Similar to the clustering-based approach, most of PR-Groups (over 90%) are smaller than 50. However, there are several PR-Groups whose sizes are larger than 500 and the largest size is 997. Only two pairs of repackaged and original apps are not grouped into the same PR-Groups, and therefore the false negative rate of NNS-based approach is 1%.

5.4 Accuracy



(a) CDF (b) Histogram

Figure 6: Distribution of the sizes of PR-Groups when NNS-based approach is employed, with distance threshold 0.15.

Section 5.2 and Section 5.3 show that both clustering-based approach and NNS-based approach can effectively detect repackaged apps with very low false negative. To evaluate the false postive rate of our approaches, we randomly selected 100 groups from the PR-Groups produced by clustering-based approach, and manually checked their resources and `smali` codes. We also executed these apps in Android emulator to check whether they have similar GUI and functionality. We did the same thing to the PR-Groups produced by NNS-based approach.

In the results of clustering-based approach, we found three PR-Groups that contain false positives. So the false positive rate this approach is 3%. But not surprisingly, the sizes of these three PR-Groups (i.e., 80, 84 and 127) are larger than the average value. Our analysis reveals two major reasons for these false positives. First, those apps have only a few activities and their simple functionalities lead to similar statistical features and structural features. Second, apps created by some online tools (e.g., App Makr[5]) may use the same template, and therefore these apps have similar appearance and event handlers. We discuss how to handle them in Section 6.

In the results of NNS-based approach, we found 5 PR-Groups containing false positives, and hence the false positive rate is 5%. Similar to the results of clustering-based approach, these false positives were all found from groups with large size (i.e., 520, 548, 641, 885 and 991). It shows that NNS-based approach is not as accurate as clustering-based approach. The reason may be that when querying nearest neighbors, k-d tree (or other NNS algorithms) only considers most similar ones "locally" whereas in clustering-based approach each sample will be compared with others (i.e., construct a global view). It demonstrates a tradeoff be-

[5] `http://www.appmakr.com`

tween efficiency and effectiveness. That is, clustering-based approach suffers from high computational complexity but is more accurate. In contrast, NNS-based approach is much more efficient but less accurate.

5.5 Complexity

Since the time complexity of both spectral clustering and hierarchical clustering are $O(N^3)$, the overall complexity of clutering-based approach is $O(N^3)$. Although the computational cost of clustering-based approach is high, we may leverage parallel computing to speed up the process. For NNS-based approach, we sort feature vectors in lexicographical order before building k-d tree. This step can be accomplished within $O(NlogN)$. Moreover, the complexity of building k-d tree and that of querying nearest neighbors are also $O(NlogN)$. Therefore, the overall time complexity of NNS-based solution is $O(NlogN)$.

5.6 Code Obfuscation/App Hardening

We implement DexDumper to extract the original `classes.dex` from apps protected by typical hardening systems. Therefore, attackers could not employ such hardening techniques to evade ResDroid.

We also tested ResDroid's ability to confront obfuscation techniques. We first employed Proguard to obfuscate apps from source codes. The results showed that both statistical features and structural features were not affected. Then we used SandMark[6] to generate obfuscated code from bytecode. However, since SandMark does not support Android's dex format, we converted dex files into jar files through dex2jar and then fed jar files to SandMark. Unfortunately, although SandMark provides 39 kinds of obfuscation methods, only three of them (static method bodies, method merger and class encrypter) could successfully process these jar files. After converting the obfuscated jar files back to dex through `dx`, we found that *static method boides* and *method merger* had no influence on ResDroid. However, ResDroid failed to extract structural features when the jar was encrypted by the class encrypter, and we propose possible solutions in Section 6.

5.7 Observations

Apart from the ground truth, ResDroid identified 64 repackaged apps. We examined some of them manually and report the observations.

5.7.1 App Plagiarism

We found that some plagiarizers repackaged apps and republished these apps as their own. For example, an app (package name: com.bluedog1893.android.translate) has identical structural features with another app (package name: com.dollars.translate). However, they have completely different icons. After manual examination, we found that the former one contains most of the resources and the whole codes from the latter one, but the string resources have been converted from English to Chinese. Moreover, while the original app does not have ads, the plagiarizer added ad libraries to make profits.

5.7.2 Massively-produced Apps

We found a set of apps from the same developer having the same structural features. Figure 7 shows three of them.

[6] `http://sandmark.cs.arizona.edu`

(a) App 1 (b) App 2 (c) App 3

Figure 7: Three apps have exactly the same layout

To create these apps, the developer just wrote the codes once and then applied the same codes to different resources for quickly producing "new" apps.

Another example comes from a PR-Group where the package names of all 83 apps follow the pattern: com.lvping.mobile.cityguide.*. These apps provide guidance for travellers, and the last piece of their package names represents the city's name. For example, the app with package name com.lvping.mobile.cityguide.sydney236 offers a travelling guide for Sydeny. However, these 83 apps were signed with two different certificates. Specifically, 4 of them are signed by the certificate whose MD5 fingerprint is '246DA3F3F52830A9E3FD04111BA4C1D4', while other 79 apps are signed by the certificate whose MD5 is '76439FA93B09D3FA51874769C74486AB'. We examined the owners and issuers of the two certificates. The first one is "Android Debug" used for signing debugging version of apps. The second one is the company's domain (i.e., `lvping.com`). We are not sure whether the ones signed by the debugging certificate are repackaged apps created by attackers or it is just because the developer forgot to export release version and sign them using official certificates.

5.7.3 Misusing Certificates

We found a pair of repackaged apps that have the same package name (i.e., `a5game.leidian2`) but different certificate fingerprints. One was signed by 'EE1C7585428F65BAC2D156B0792D2358', and the other app was signed by '86544D775DCBA00275CD304C5C37BCC7'. We investigated the two certificates and found the former is owned by "5agame.com", but the latter's owner is in Chinese characters, which is the name of the website `http://www.5agame.com`. After careful examination of their codes, we did not find anything abnormal. It is likely that both the two apps were actually published by the same developer "5agame.com", but they chose different certificates for the same app for some unknown reasons. This may result in failures of updates, because the Android system does not allow the newer app and the older one have different certificates. Therefore, it is an example of misusing app certificates.

6. DISCUSSION

6.1 Attack Analysis

Zhang et al. described the behaviors of three kinds of repackaging attacks [45], including (1) lazy attacks that use automatic code obfuscation tools to repackage apps without modifying the functionality of original apps; (2) malware that embeds malicious payload into original apps without

changing them; (3) amateur attacks that may make some changes to original apps besides employing the automatic code obfuscation tools. Since none of them will modify the resources and the related codes, ResDroid can detect them.

Advanced attackers who know ResDroid may change features to evade detection. For example, they can insert many junk resources into the repackaged app for affecting the statistical features. However, we consider the relations between resources and codes (e.g., references from XML files, loading from code) and identify core resources, from which statistical features are extracted. Therefore, junk resources that are not carefully crafted will be filtered out. Dedicated attackers may change statistical features by inserting resources and altering the dex file simultaneously at a cost of increased app size and degraded performance. We may apply dead-code detection techniques [12] to identify such junk functions and remove them along with the corresponding resources when computing the features.

As explained in Section 3.5, it is difficult for attackers to re-implement a new layout while keeping the same looks and feels. Moreover, to retain the normal functionality and QoE of the repackaged app, attackers will not remove event handlers. Although attackers can add new event handlers, they cannot modified the result of normalized LCS as explained in Section 3.5.3. Therefore, the structural features are robust and can raise the bar for attackers to evade detection.

6.2 Limitations and Future Work

We notice that apps created by automatic building tools will cause false positives, because these tools provide developers a set of templates to create apps. Apps using the same template will shared similar appearance and the event handlers, therefore, their statistical features and structural feature are alike. ResDroid cannot differentiate them. In future work, we will use components' attributes to differentiate them. Another possible approach is to use code-level detection systems such as DNADroid [19] to handle them.

If code obfuscation/app hardening systems employ various dynamic loading techniques and encryption methods to prevent static analysis, ResDroid may not be able to handle such apps through static analysis. However, since these techniques will eventually load codes and resources into memory, we will design a kernel-based dynamic approach like [42], which keeps monitoring the behavior of a hardened app and dumps selected codes and resources after they are loaded. Moreover, we will examine how to fingerprint apps protected by different code obfuscation/app hardening systems.

7. RELATED WORK

Code clone detection. Considerable research has been conducted on code clone detection [11, 39, 41]. Existing approaches can be roughly classified into four groups [41]: (1) textual analysis that extracts fingerprints from code directly; (2) lexical analysis that converts codes into lexical tokens and then detects duplicated token sequences (e.g., CP-Miner [32], etc.); (3) syntactic analysis that first turns codes into abstract syntax trees (AST) and then applies tree matching or structural metrics to detect clones (e.g., Deckard [30], etc.); (4) semantic analysis that employs static program analysis to extract more precise information about the code, such as program dependency graph, for detection (e.g., GPLAG [33], etc.). Some of these methods have been used to detect repackaged apps by analyzing dex files, or the converted Java class files, or the disassembled smali codes.

Repackaged apps detection. Assuming apps from the official market are original, DroidMOSS applies fuzzy hashing to each app's opcodes and then compares it to original apps' fingerprint for detecting repackaged apps [47]. Similarly, Androguard [1] uses several standard similarity metrics to hash methods and basic blocks for comparison. Juxtapp characterizes apps through k-grams of opcodes and feature hashing and then clusters the corresponding bitvectors to identify app clones [27]. PiggyApp was designed to detect piggybacked apps, a special kind of repackaged apps, which contain injected code [46]. It first decouples modules according to their dependency relationship and then construct fingerprint for the primary module by collecting various features, such as requested permissions, Android API calls used, etc. [46]. These methods are vulnerable to simple obfuscation techniques because they consider few semantic information about codes [29, 43, 44].

Dresnos used normalized compression distance (NCD) [15] to compare the similarity of apps according to their method signatures, including external API used, exceptions, and control flow graph (CFG) [22]. Potharaju et al. proposed an approach to detect plagiarized apps according to symbol tables and method-level AST fingerprints. This approach can handle two kinds of obfuscation techniques that mangles symbol table or inserts random methods with no functionality [37]. DroidSim utilizes component-based control flow graph (CB-CFG) to quantify the similarity between apps [43]. DNADroid constructs a program dependency graph (PDG) for each method and then performs subgraph isomorphism comparison on PDGs after filtering out unnecessary methods [19]. To speed up DNADroid, AnDarwin splits PDGs into connected components (i.e., semantic blocks), each of which will be represented by a semantic vector containing the number of specific types. After that, it employs locality sensitive hashing (LSH) to identify code clones that have similar semantic vectors [20]. Chen et al. proposed a novel approach that uses the centroid of control dependency graph to measure the similarity between methods for detecting cross-market app clones [13]. Although these methods are better than the previous ones, they could still be easily evaded by obfuscation techniques (e.g., inserting dummy codes or adding data related variables) [43, 45].

Recently, Hao et al. showed that it is possible to detect app clones using UI state transition graphs [28]. In a simultaneous research, Zhang et al. proposed ViewDroid that first constructs feature view graph and then applies subgraph isomorphism algorithm to measure the similarity between two apps [45]. Although both ViewDroid and ResDroid exploit UI for detecting repackaged apps, there are three major difference between them. First, ViewDroid only uses the relationship among activities while ResDroid employs both the layout of activities and the relationship among activities. Moreover, we take into account Android's fragment component that provides similar functionality as an activity. However, ViewDroid does not handle such component. Second, by only examining core resources, ResDroid would be more efficient and more robust to third-party libraries than ViewDroid that considers all views. Third, whereas ViewDroid targets on comparison between a pair of apps, ResDroid is built on top of a two-stage methodology and equipped with two kinds of algorithms.

8. CONCLUSION

We propose a novel approach that leverages new features extracted from core resources and codes to detect repackaged apps. These features do not require processing all opcodes and are resilient to code obfuscation/app hardening techniques. To speed up the detection, we adopt the divide-and-conquer strategy to reduce the comparison and support parallel processing. Our solution have been realized in ResDroid and the extensive evaluation using real repackaged apps has demonstrated its effectiveness and efficiency.

9. ACKNOWLEDGMENT

We thank reviewers for their comments, and thank Wenjun Hu and Kai Chen for providing us samples. This work is supported in part by the Hong Kong GRF (No. PolyU 5389/13E), the National Natural Science Foundation of China (No. 61202396), the PolyU Research Grant(G-UA3X), and the Open Fund of Key Lab of Digital Signal and Image Processing of Guangdong Province(2013GDDSIPL-04).

10. REFERENCES

[1] Androguard. `https://code.google.com/p/androguard/`, 2011.

[2] 1.2% of apps on google play are repackaged to deliver ads, collect info. http://www.net-security.org/secworld.php?id=15976, 11 2013.

[3] App hardening emerges as a key component of mobile security strategies. http://betanews.com/2013/12/03/app-hardening-emerges-as-a-key-component-of-mobile-security-strategies, 2014.

[4] Gator: Program analysis toolkit for android. `http://dacongy.github.io/gator/`, 2014.

[5] Managing the activity lifecycle. `http://developer.android.com/training/basics/activity-lifecycle/index.html`, 2014.

[6] Resources overview. `http://developer.android.com/guide/topics/resources/overview.html`, 2014.

[7] C. Aggarwal and C. Reddy. *Data Clustering: Algorithms and Applications*. Chapman and Hall/CRC, 2013.

[8] A. Andoni and P. Indyk. Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. *Commun. ACM*, 51(1), 2008.

[9] Arxan Technologies Inc. State of security in the app economy. http://www.arxan.com/resources/state-of-security-in-the-app-economy/.

[10] T. Azim and I. Neamtiu. Targeted and depth-first exploration for systematic testing of android apps. In *Proc. ACM SIGPLAN*, 2013.

[11] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo. Comparison and evaluation of clone detection tools. *IEEE Trans. Software Eng.*, 33(9), 2007.

[12] D. Beyer, A. Noack, and C. Lewerentz. Efficient relational calculation for software analysis. *IEEE Trans. Software Eng.*, 31(2), 2005.

[13] K. Chen, P. Liu, and Y. Zhang. Achieving accuracy and scalability simultaneously in detecting application clones on android markets. In *Proc. ACM ICSE*, 2014.

[14] W.-Y. Chen, Y. Song, H. Bai, C.-J. Lin, and E. Y. Chang. Parallel spectral clustering in distributed systems. *IEEE Trans. Pattern Anal. Mach. Intell.*, 33(3), 2011.

[15] R. Cilibrasi and P. Vitanyi. Clustering by compression. *IEEE Transactions on Information Theory*, 51(4), 2005.

[16] Cisco Systems Inc. Annual security report, 2014.

[17] C. Collberg and J. Nagra. *Surreptitious Software: Obfuscation, Watermarking, and Tamperproofing for Software Protection*. Addison-Wesley, 2009.

[18] S. Corporation. Internet security threat report, 2014.

[19] J. Crussell, C. Gibler, and H. Chen. Attack of the clones: Detecting cloned applications on android markets. In *Proc. ESORICS*, 2012.

[20] J. Crussell, C. Gibler, and H. Chen. Scalable semantics-based detection of similar android applications. In *Proc. ESORICS*, 2013.

[21] J. Davies, D. M. German, M. W. Godfrey, and A. Hindle. Software bertillonage. *Empirical Software Engineering*, 18(6), 2013.

[22] A. Dresnos. Android: Static analysis using similarity distance. In *Proc. HICSS*, 2012.

[23] W. Enck, P. Gilbert, B. gon Chun, L. P. Cox, J. Jung, P. McDaniel, and A. Sheth. Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *Proc. OSDI*, 2010.

[24] C. Gibler, R. Stevens, J. Crussell, H. Chen, H. Zang, and H. Choi. Adrob: examining the landscape and impact of android application plagiarism. In *Proc. ACM MobiSys*, 2013.

[25] M. Grace, Y. Zhou, Q. Zhang, S. Zou, and X. Jiang. Riskranker: scalable and accurate zero-day android malware detection. In *Proc. ACM MobiSys*, 2012.

[26] M. Halkidi, Y. Batistakis, and M. Vazirgiannis. On clustering validation techniques. *Journal of Intelligent Information Systems*, 17(2-3), 2001.

[27] S. Hanna, L. Huang, E. Wu, S. Li, C. Chen, and D. Song. Juxtapp: a scalable system for detecting code reuse among android applications. In *Proc. DIMVA*, 2012.

[28] S. Hao, B. Liu, S. Nath, W. Halfond, and R. Govindan. PUMA: Programmable ui-automation for large scale dynamic analysis of mobile apps. In *Proc. ACM MobiSys*, 2014.

[29] H. Huang, S. Zhu, P. Liu, and D. Wu. A framework for evaluating mobile app repackaging detection algorithms. In *Proc. TRUST*, 2013.

[30] L. Jiang, G. Misherghi, Z. Su, and S. Glondu. Deckard: Scalable and accurate tree-based detection of code clones. In *Proc. IEEE ICSE*, 2007.

[31] J.-H. Jung, J. Y. Kim, H.-C. Lee, and J. H. Yi. Repackaging attack on android banking applications and its countermeasures. *Wireless Personal Comm.*, 73(4), 2013.

[32] Z. Li, S. Lu, S. Myagmar, and Y. Zhou. Cp-miner: Finding copy-paste and related bugs in large-scale software code. *IEEE Transactions on Software Engineering*, 32(3), 2006.

[33] C. Liu, C. Chen, J. Han, and P. Yu. Gplag: Detection of software plagiarism by program dependence graph analysis. In *Proc. ACM KDD*, 2006.

[34] M. Newman. *Networks: An Introduction*. Oxford University Press, 2010.

[35] A. Papadopoulos. *Nearest Neighbor Search: A Database Perspective*. Springer, 2004.

[36] D. Pelleg, A. W. Moore, et al. X-means: Extending k-means with efficient estimation of the number of clusters. In *Proc. ICML*, 2000.

[37] R. Potharaju, A. Newell, C. Nita-Rotaru, and X. Zhang. Plagiarizing smartphone applications: Attack strategies and defense techniques. In *Proc. ESSoS*, 2012.

[38] C. Qian, X. Luo, Y. Shao, and A. Chan. On tracking information flows through JNI in android apps. In *Proc. IEEE/IFIP DSN*, 2014.

[39] D. Rattan, R. Bhatia, and M. Singh. Software clone detection: A systematic review. *Information and Software Technology*, 55(7), 2013.

[40] A. Rountev and D. Yan. Static reference analysis for gui objects in android software. In *Proc. IEEE/ACM CGO*, 2014.

[41] C. Roy, J. Cordy, and R. Koschke. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Sci. Comput. Program.*, 74(7), 2009.

[42] Y. Shao, X. Luo, and C. Qian. Rootguard: Protecting rooted android phones. *IEEE Computer*, June 2014.

[43] X. Sun, Y. Zhongyang, Z. Xin, B. Mao, and L. Xie. Detecting code reuse in android applications using component-based control flow graph. In *Proc. IFIP SEC*, 2014.

[44] M. Vasquez, A. Holtzhauer, C. Bernal-Cardenas, and D. Poshyvanyk. Revisiting android reuse studies in the context of code obfuscation and library usages. In *Proc. IEEE MSR*, 2014.

[45] F. Zhang, H. Huang, S. Zhu, D. Wu, and P. Liu. Viewdroid: Towards obfuscation-resilient mobile application repackaging detection. In *Proc. ACM WiSec*, 2014.

[46] W. Zhou, Y. Zhou, M. Grace, X. Jiang, and S. Zou. Fast, scalable detection of piggybacked mobile applications. In *Proc. ACM CODASPY*, 2013.

[47] W. Zhou, Y. Zhou, X. Jiang, and P. Ning. Detecting repackaged smartphone applications in third-party android marketplaces. In *Proc. ACM CODASPY*, 2012.

[48] Y. Zhou and X. Jiang. Dissecting android malware: Characterization and evolution. In *Proc. IEEE Symp. Security and Privacy*, 2012.

TROGUARD: Context-Aware Protection Against Web-Based Socially Engineered Trojans

Rui Han*, Alejandro Mesa*, Mihai Christodorescu†, Saman Zonouz‡
*†Electrical and Computer Engineering
*University of Miami, †QualComm Research, ‡Rutgers University
{r.han, a.mesa11}@umiami.edu, mihai@qti.qualcomm.com, saman.zonouz@rutgers.edu

ABSTRACT

Despite the increasing number of social engineering attacks through web browser applications, detection of socially engineered trojan downloads by enticed victim users remains a challenging endeavor. In this paper, we present TROGUARD, a semi-automated web-based trojan detection solution, that notifies the user if the application she downloaded behaves differently than what she expected at download time. TROGUARD builds on the hypothesis that in spite of millions of currently downloadable executables on the Internet, almost all of them provide *functionalities* from a limited set. Additionally, because each functionality, e.g., text editor, requires particular system resources, it exhibits a unique system-level activity pattern. During an offline process, TROGUARD creates a profile dictionary of various functionalities. This profile dictionary is then used to warn the user if she downloads an executable whose observed activity does not match its advertised functionality (extracted through automated analysis of the download website). Our experimental results prove the above mentioned premise empirically and show that TROGUARD can identify real-world socially engineered trojan download attacks effectively.

1 Introduction

Social engineering attacks rely on the user being a point of weakness in any secure system and create one of the most challenging security problems, where users themselves perform or facilitate attack steps. For example, in the case of "fake anti-virus" malware, the user is convinced to download the legitimate-looking executable to her computer and then willingly execute a piece of software that is core to the attack. Protecting a computer system and its users against such socially engineered download attacks appears to be impossible, as any such protection could interfere with the user's freedom to install software on their own computer. We describe in this paper our efforts to alleviate this threat without impinging on the user's freedom, by providing a system that automatically compares the stated goal of a software program (as perceived by the user) with its actual goal (as inferred from the program's execution).

Web-based socially engineered download attacks that result in a trojan software installation on the victim's computer are becoming more widespread. This is mainly due to new security measures (e.g., memory-page protection and address randomization to prevent buffer overflows) that significantly reduce the success rate of other popular, automated attack vectors. Trojan downloads range from rogue security software (also known as "scareware" or fake anti-virus), to fake games, fake video codecs, and to fully functional pirated softwares that are infected with malware. As a case in point, `TrojanClicker.VB.395` (detected by TrojanHunter [1]) purports to be an Adobe Flash updater; however, once downloaded and installed by unsuspecting users, the trojan launches a spyware that monitored and uploaded all Google searches to a remote server `www.msjupdat e.com`. To make the users aware of such attacks, Adobe announced that malicious hackers were starting to use fake Flash Player downloads as social engineering lures for malware and issued a call-to-arms for users to validate installers before downloading software updates. The company's notice came on the heels of malware attacks on Facebook, MySpace and Twitter that attempted to trick Windows users into installing a Flash Player update [23]. Furthermore in the last few years the threats of trojan downloads have expanded quickly from desktop to mobile platforms, e.g., the Opfake browser [11], where the app-store concept seems to drive the user-made download decisions away from provenance and towards functionality.

There are unfortunately few techniques to help users protect themselves against trojan downloads. One technique is based on dynamically updated blacklists, where web browsers check against an online blacklist service (e.g., Google SafeBrowsing [14]) that the current web page does not contain malicious software. Similarly, anti-virus software (e.g., McAfee [21]) also uses a blacklist to determine whether the downloaded program is malicious. Both of these security techniques rely on precise and timely maintenance of the blacklists—a time-consuming challenging endeavor in practice. Protection techniques that monitor for "drive-by downloads" (e.g., Blade [18]) are inapplicable here because trojan downloads are performed through user interaction, not via an exploit. User education to protect against trojans is ineffective at best since the vast majority of users lack the necessarily complex technical skills and tools to determine whether a downloaded program is malicious. Consequently, almost all of the existing techniques fall short in providing an effective protection mechanism against trojan download attacks that could ideally manage the whole download-install-execute process for the user and provide security in that context.

Our approach bridges the gap between what the user perceives regarding the functionality of the downloaded software and the functionality observed during its actual execution. We observe that, throughout the download–install–execute process, the user maintains a consistent expectation about the functionality of the downloaded program. Following the Principle of Least Surprise [9] in user-interface design, i.e., "design should match the user's experience, expectations, and mental models" [27], we wish to build on the user's expectations of a downloaded application and her experience with past applications with similar functionalities. To this end, our solution detects trojan downloads by comparing the user's expectation of functionality (as primed by the software download web page) with the actual functionality exhibited at runtime, and imposes constraints on the downloaded application's local execution using the contextual information on the download web page.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.
ACSAC'14, December 08 - 12 2014, New Orleans, LA, USA.
ACM 978-1-4503-3005-3/14/12...$15.00
http://dx.doi.org/10.1145/2664243.2664270.

Table 1: 90% of the top MacOS Malware are Trojans [19].

Rank	Name	Percentage
1	Trojan.OSX.FakeCo.a	52%
2	Trojan-Downloader.OSX.Jahlav.d	8%
3	Trojan-Downloader.OSX.Flashfake.ai	7%
4	Trojan-Downloader.OSX.FavDonw.c	5%
5	Trojan-Downloader.OSX.FavDonw.a	2%
6	Trojan-Downloader.OSX.Flashfake.ab	2%
7	Trojan-FakeAV.OSX.Defma.gen	2%
8	Trojan-FakeAV.OSX.Defma.f	1%
9	Exploit.OSX.Smid.b	1%
10	Trojan-Downloader.OSX.Flashfake.af	1%

Comparing an application's expected and actual behaviors, while intuitive, poses significant challenges. It is unclear whether an actual system-level behavior can be summarized concisely that is high-level enough to compare with a label that a typical human user could place on the downloaded application. This is mainly because of the huge gap between the human description of the application (in terms of abstract concepts and functionalities such as web browsing) and its system-level behavior (in terms of concrete operations and concrete runtime metrics such as syscalls). Additionally even obtaining a concise label from the human description is not straightforward, as web pages for application downloads often contain extraneous marketing information. We tackle these challenges to create an end-to-end system for detecting trojan downloads.

Our system, called TROGUARD, analyzes the information visible to the user on the software download web page to infer the perceived software category. TROGUARD also monitors the execution of the downloaded program to infer its actual software category in real-time, and alerts the user if the perceived and actual categories do not match. We chose to abstract as *software category* both the web-page information around the program download and the execution of the program, because a software category matches closely to how users commonly understand software. Finding this middle ground between human descriptions of software functionality and low-level system operations invoked by the software is non-trivial, because those software categories need to be not only meaningful to users (e.g., "office application" vs. "text editing tool") but also extractable from the low-level system traces. Furthermore, applications within the same category should expose similar system-level profiles to be classified as the same category by TROGUARD later.

We make the following contributions in this paper:

- A new approach to detecting trojan downloads is introduced, that at its core relies on comparing the functionality expected by the user (efficiently determined through in-depth and automated analysis of the download web site) to the functionality exhibited during the downloaded application's execution;
- An end-to-end system for detecting trojan downloads is designed and implemented to identify mismatches between user-perceived and actual software categories. TROGUARD consists of a browser extension and plugin as well as a host monitor that communicate with each other to bridge the gap between user experience and system activity; and
- A comprehensive evaluation over a large data set shows high-accuracy detection (up to 98.3%), even in the presence of programs that naturally combine functionality from multiple (traditional) software categories.

The paper motivates our approach (Sections 2 and 3), describes our design (Sections 4.1 and 4.2), and evaluates our TROGUARD system (section 5). We discuss the past work (section 6) proposed solution's limitations in section 7 and finally conclude in section 8.

2 Motivation and Overview

The attacks via web-based trojan downloads are hard to protect against because they involve a social engineering step where the user (as sociotechnical root of trust) gets tricked (i.e., compromised) into downloading and running the trojan program themselves. This favors attackers such that nine out of the top-10 MacOS threats in 2012 were trojan downloads (Table 1) causing 30% increase in number of signatures created by Kaspersky Labs [19].

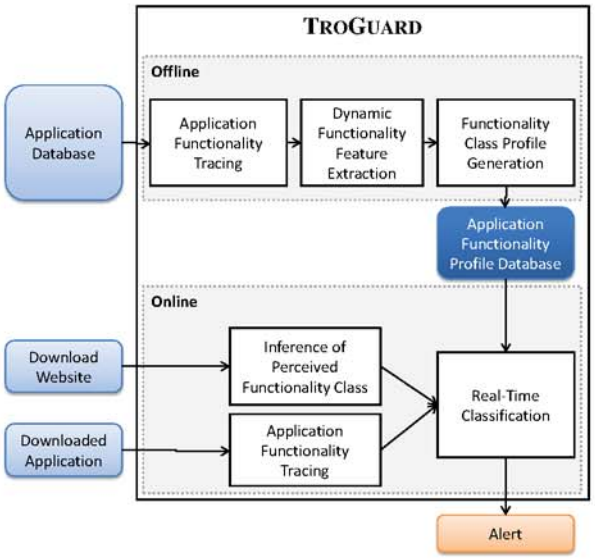


Figure 1: TROGUARD's High-Level Architecture

We tackle this problem by equipping the user with a tool to answer the question "Is this downloaded program doing what I expect it to do?". To this end, we combine an analysis of the information about the program that is available to the user on the download web page with an analysis of the runtime information collected by the user's computer from the program execution. Both of these analyses produce summaries of the program functionality, one of perceived functionality (as the user sees it) and one of actual functionality (as the host system sees it). The perceived and actual functionality profiles are then compared against each other and the user is notified in the case of mismatch.

We present an overview of the proposed solution and discuss how it protects the client systems against web-based socially engineered trojan threats. Figure 1 illustrates individual components in TROGUARD and how they are logically interconnected. The overall solution consists of two major steps of 1) offline application tracing and data analysis, and 2) online user query and dynamic malware detection.

More specifically, during the offline phase, TROGUARD captures and traces several behavioral aspects of various applications with different *functionalities* dynamically. TROGUARD analyzes the program from a collection of labeled applications where each application, e.g., Firefox, is marked with the functionality it provides, e.g., web browser. The abovementioned collection is created such that most of widely-used functionalities, e.g., web browser or office suite, are provided by multiple applications. The objective of the offline phase is to create a behavioral profile of each functionality in terms of its system-level activities, such as filesystem reads/writes, network data transfer, and user interactivity. The premise, that we prove by our experimental evaluation, is that behavioral aspects of applications with similar functionality are similar to each other and different from those of applications with different functionalities.

Once the application activities are captured, the recorded data sets with the functionality labels are fed into a supervised machine-learning algorithm to extract discriminative features about each functionality. For instance, game applications often exhibit intense user interactivity and time-varying system CPU usage while music players present minimal user interactivity and stable CPU usage over time, with low filesystem and network interaction. Similarly, word-processor users will press navigation and alphanumeric keys significantly more than other key categories, while for game players either use a limited times of alphanumeric keys such as A, S, D, and W. The discriminative features for each functionality form a

profile feature vector and are stored for the online detection phase. It is noteworthy that the offline phase is a one-time effort and the functionality profiles, once created, can be reused across multiple systems.

The second phase in TROGUARD is that of on-demand website and application analysis, leading to the automated real-time malware detection. When the user intends to download a legitimate-looking (potentially malicious trojan) program, e.g., a game, from a website, TROGUARD monitors the download and installation process on the local system closely using its browser extension and plugin components, respectively. Once the browser extension component in TROGUARD notices a download initiated by the user, it starts the website analysis process to determine the type of application (functionality) being downloaded, by parsing the text from the web site and from its images (which are first processed using OCR). TROGUARD then calculates an list of most-likely functionalities and presents them to the user before the download starts. The user confirms or corrects the expected functionality for TROGUARD to use during its dynamic program analysis. Once the program is downloaded, installed, and executed in a sandboxed environment (all steps automatically monitored), TROGUARD's runtime tracing component starts behavioral monitoring, data collection, and real-time classification of its behavior to verify that the application does not violate the corresponding functionality profile obtained from the offline phase.

Revisiting our example of `TrojanClicker.VB.395` from the introduction, TROGUARD would infer that the program to be downloaded claims to be a software updater and would retrieve the appropriate functionality profile from its database. When the trojan starts executing, TROGUARD would monitor its activity, looking for updater-like behaviors (e.g., many filesystem and registry operations for existing entries, little-to-no network traffic). As the trojan installs itself into the system (into a new location and as an extension to any web browsers present) and as it communicates over the network (sending collected logs of Google searches), TROGUARD observes the discrepancy and notifies the user consequently.

3 Threat Model

TROGUARD protects the users against a specific class of trojan download attacks where, through a web-based social engineering step, the user is lured into downloading and installing a malicious legitimate-looking executable on her system. The goal of the web-based trojan exploit is to take effective control of the client machine in order to complete subsequent malicious activities such as bot deployment.

The attacker wishes to trick the user into downloading the trojan program, and can use any social engineering technique for this purpose. Additionally, the attacker has full control over the web page that proposes the download to the user and can use any web technology on that page, with the exception of exploits. In other words, we place the user's web browser and underlying operating system into the trusted computing base (TCB) and assume that they have no vulnerabilities. Protecting the TCB from attacks is orthogonal to our work and has been widely researched by others, with a variety of solutions available.

We assume that the attack does not involve the use of exploits or other automated techniques, but only the willing cooperation of the user to install and launch the trojan software. It is important to highlight that our threat model is subtly different from drive-by downloads (e.g., as addressed by Blade [18]), where the adversary performs a *surreptitious* download via the user's browser through a shell-code injection step, with no social engineering needed. Furthermore, unlike in our threat model, the user does not notice the download and installation of the malware. This type of drive-by download attack requires a vulnerable browser to be feasible and is outside the scope of the present work.

4 Design and Implementation

TROGUARD analyses the source website of an application, while that application is being downloaded, to determine the appropriate functionality profile for that application. It then compares the execution of the downloaded application against this profile and alerts the user if it discovers any discrepancies. In this section, we discuss what a functionality profile is and how we build a representative database of them.

4.1 Offline Construction of Profile DB

4.1.1 Functionality Classes

Functionality class is a key concept in our system, as it represents both the user's understanding of a software category and the system's observation of a software's execution behavior. Defining an ideal structure to capture the concept of functionality class is close to impossible, as an unlimited number of structures of varying degrees of abstraction could serve the purpose.

In TROGUARD we use a highly summarized view of the program execution to define a functionality class. In particular, the profile associated with a functionality class is a vector of key behavior features, as explained in the following subsections. Each behavior feature is high-level enough to be describable to the user (e.g., "Program X was classified as a game because in normal use it generates a lot of user interaction", where *level of user interaction* is one feature in the profile vector). Further more, each behavior feature should be efficiently computable in real time from the program execution observations.

The remaining challenge is how to select the most representative functionality classes. By "most representative" we mean both classes that are meaningful to users and with member programs that share behaviors. As an example of a poor choice of functionality class, "System Utilities" fails our criteria because its meaning is too generic to a user and its member programs have many, distinct behaviors with no common functionality. We turned to ten software-download websites such as Softpedia, Tucows, and File-Guru, and surveyed their top-level software categories. As these website are popular Internet destinations for software downloads, their categories likely reflect users' understanding and serve as a good starting point for TROGUARD. We summarized the categories and combined them into a more concise set, as shown in Figure 2.

It is noteworthy that the correct selection of functionality classes results in effective detection of behavioral mimicry attacks, where the carefully crafted malware attempts to accomplish a malicious objective while pretending a legitimate functionality delivery.

4.1.2 Bridging the Semantic Gap

The goal of our system is to distinguish an application's genuine type from its advertised type based on functional profiling of the application against a trained application functionality class database. The system should first be trained using a supervised training algorithm with a large set of sample applications. To generate discriminative features, comprehensive functional profiles are required. Unlike previous work on behavior based malware clustering [4] and [7], besides system call traces, application behavioral in our work is expanded to include user space properties: CPU and memory usage, network protocols, port numbers, the number of IP addresses (or domain names) the application connects to, and the user machine interactivity, such as keyboard strokes and mouse clicks. The presence or absence of those attributes in each application's profile not only represent the core functionalities of that application, but can also be used to separate a particular type of application from the rest. Practically, such behavioral profiles are generated by collecting and analyzing the system wide activities both in kernel space and user space. In this section we will discuss the feature list generation and it properties.

4.1.3 Intermediate-Level Application Behaviors

The main challenge in designing such functionality-based detection solutions is the large semantic gap that exists between the high-level user-perceivable functionality classes (subsubsection 4.1.1)

TROGUARD Functionality Class	Softpedia.com	download.cnet.com	Tucows.com
Graphics Editor Game	Artistic SW Games	Graphic Design SW Games	Design tools Games
Browser Instant Messenger (IM)	Internet Communications	Browsers Communications	} Internet
Media Player Audio Editor Video Editor	} Multimedia	} MP3/Audio SW Video SW	} Audio / Video
Office IDE Calculator	Office Programming Utilities	Productivity SW Developer Tools Utilities/OS	Business Dev/Web Home/Education

Figure 2: Functionality classes in TROGUARD and their corresponding categories for three popular software-download web sites.

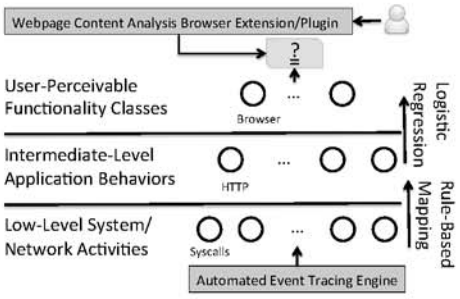


Figure 3: TROGUARD Components.

and low-level system activities, e.g., syscalls. To fill the semantic gap without sacrificing detection accuracy much, TROGUARD uses an intermediate level of features that not only are semantically closer to the functionality classes but also can be inferred from the low-level system and network event logs (Figure 3). In particular, the system-wide tracing instrumentations notifies TROGUARD about low-level activities. TROGUARD accomplishes intermediate-level feature inference from the low-level activities using predefined rules using regular expressions. The inference rules are designed as a one-time effort (took two person-days in our experiments) and, once developed, could be reused across machines. The inference rules are either direct or indirect. TROGUARD directly infers an intermediate-level feature when it explicitly matches particular points in the system-level activity logs such as a file type access attempt or a specific network protocol. Indirect inference requires more computation since the information should be extracted from different points such as syscall arguments or their sequences, and processed to check whether they satisfy a given regular expression rule. For instance, the intermediate attributes `protocol::imap` and `works-with-format::pdf` belong to the directory mapped attributes from Debtags [32] and can be obtained directly from system call argument. However, the feature `interface::x11` is inferred indirectly from low level system call activities such as opening and frequently reading Xorg library files.

4.1.4 Low-Level System Activities

TROGUARD extracts low-level system features using its dynamic kernel and user space tracing engines. For low-level kernel activity tracing, TROGUARD intercepts syscalls; however, it does not conduct executable-specific syscall analysis, e.g., n-gram profiling, since those profiles do not hold across different applications of even the same functionality class. For instance, a file download application may read a 512-byte packet using either a single read syscall or two 256-byte reads. Therefore, TROGUARD analyzes syscalls to obtain more semantic information such as used protocols and system libraries that the application is accessing. Furthermore, we focus on the input and output arguments of the syscalls as well as the syscall statistics such as average amount of network data transfer. Additionally, TROGUARD gathers syscall statistics such as syscall frequency in different operation categories, i.e., file system I/O statistics, and network I/O statistics, filesystem accesses, and inter-process communication. Though syscall analysis, TROGUARD also extracts the information flow dependency among the operating system assets, e.g., processes and directories. For instance, when a process sends an IPC message to another running process, the second process becomes directly (information flow-) dependent on the first process. Transitive information flow among a sequence of assets cause indirect dependencies. TROGUARD adds such data dependencies to the system-level features of the target application.

To further enrich the application's feature vector, TROGUARD captures CPU and memory usage, user-machine interactivity, port numbers, and the number of unique remote IP addresses (or domain names) the application connects to. In particular, TROGUARD considers CPU and Memory usage logs, because, as shown in our experiments, those statistics provide fairly discriminative features for applications of different functionality classes. As a case in point, the web browser resource consumptions start with a sharp jump up at the start up, and then settle down to a moderate level linearly increasing for each tab, whereas the text editors consume a stable and fixed amount of system resources. Moreover, resource usage variations for some applications denote the degree of user interactivity at each time point. Furthermore, TROGUARD captures the human interactivity features more explicitly by logging temporal statistics of keyboard strokes and mouse clicks. To extract the IP/port and protocol information, TROGUARD makes use of the syscall arguments and network tracing tools. Furthermore, for remote IP addresses with known domain name, TROGUARD uses the manually labeled DMOZ open directory [10] to get the more semantic information about the web service category that the application connects to.

4.1.5 Completeness of Functionality Profiles

The functionality profiles we construct in TROGUARD are based on the application behaviors observed during a training phase, in which selected, well-known applications are run under monitoring. This necessarily means that the functionality profiles are incomplete, which could lead to false alarms (e.g., alerting that a downloaded copy of the Banshee music player is not really a music player).

In our current implementation we considered two options for the training phase. The first option relies on manual exploration, where users exercise most of the core functionality of each application in the training set. While this does not technically ensure 100% coverage of code paths, it sufficiently captures the code paths commonly executed by normal users. The second option relies on symbolic execution [22] to improve the code coverage and thus the functionality coverage. Due to space limitations, we cover the symbolic execution and its results in our upcoming journal submsision.

4.2 Online Trojan Detection

We detail here the design of the two key components powering TROGUARD, the website analysis to infer functionality classes from download pages, and the host runtime monitoring to infer functionality classes from program executions. It is noteworthy that TROGUARD can be configured to perfom all the detection steps locally on the target system, and therefore causes no privacy disclosure threats to end-users.

4.2.1 Website Analysis

TROGUARD fires up when the user interacts with the web browser to download a software application. When the user clicks a file URL to download an application in a TROGUARD protected computer, the browser extension invokes a popup window with dropdown options displaying a set of potential application types extracted from the web page. The displayed application types are

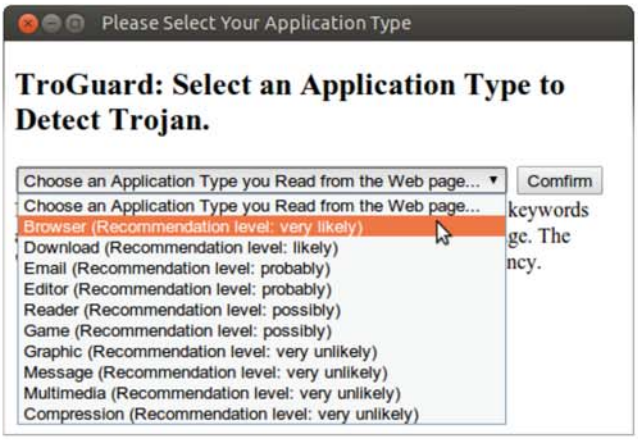


Figure 4: Proof of concept example popup window

a set of frequently occurring keywords such as web browser, email client, and text editor, etc. When the user confirms the application type by either confirming the default type or selecting another type she/he believes the program belongs to, this keyword is submitted to the extension and used later for the detection step.

TROGUARD uses the Chrome Extension technology [6], to bring up a popup window (see Figure 4) which provides the user with a list of potential application type he/she is currently downloading. TROGUARD performs website content analysis in the background prior to showing the list to the user. The content analysis component of TROGUARD makes use of the text in introducing and marketing the product, as well as related pictures on the current web page. This component automatically search all the DOM nodes in the current web page, captures screen-shots of the visible surface of the browsing window, and employs an OCR engine to extract the text in the screen-shots. We use Tesseract [28] as the OCR engine. Specifically, we study the websites front end content to obtain software vendor's favorite word choices in describing their applications. Some application names are described in a single keywords, such as browser and game, and those keywords are descriptive of the application type on their own. If the content analysis component extracts any of these descriptive keywords from the download page, the corresponding application type is added to the list of potential applications shown to the user. For other less descriptive application names, TROGUARD requires co-occurrence of multiple keywords on the page to characterize its application type in website analysis. For instance, we count the appearance of both "message" and "chat" in the web page to add Instance Message application type to the potential application list.

4.2.2 Application Monitoring

The runtime monitor is started as soon as the downloaded executable is launched. The monitor operates by monitoring low-level system activities, mapping them to intermediate-level application behaviors, and then checking these intermediate-level behaviors against the functionality profile named by the functionality class inferred from the download web page. If the intermediate-level behaviors do not match the functionality profile, an alert is raised and the user is given the option to terminate the application.

The captured low-level system features consist of intercepted system calls, CPU utilization, memory usage, keyboard activity, mouse clicks, and network activity, e.g., socket numbers and IP addresses.

TROGUARD further monitors the child and peer processes, and assigns their OS-object accesses to the top application. Peer processes are processes that were influenced by the application or one of its peer processes via inter-process communication (IPC), shared memory, remote-process calls (RPC), or other cross-process mechanisms. Consequently, TROGUARD adds the assets that interact with the application directly or indirectly to the target application's feature vector. This aggressive approach to assigning all direct and

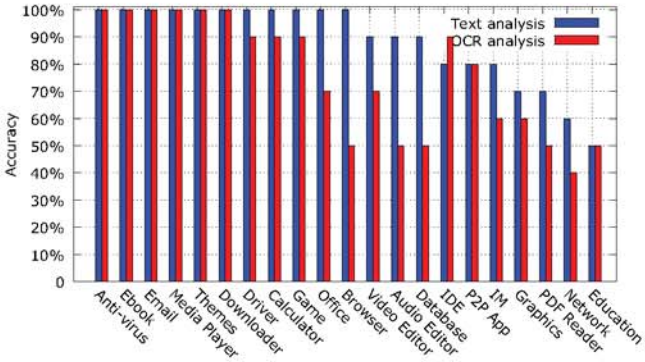


Figure 5: Website Analysis Accuracy

indirect resource accesses to the top application ensures that all malicious actions are visible to TROGUARD.

Additionally, TROGUARD also takes into account the frequency and amount of transferred data among the assets as well as between the local host and remote network end-points. For the application's network activity, TROGUARD also records the average number of unique remote end-points that the application communicates with every time unit. To capture the user-machine's interactivity, we calculate statistical measures to encode the degree and type of interactivity from observed input events. In particular, we divide keyboard keys into several types: alphanumeric, modifiers, navigation and typing modes, system commands and miscellaneous [30]. For mouse clicks, TROGUARD captures left/right/middle clicks and scroll wheel rotation.

TROGUARD is primarily a monitoring, classification, and detection framework and as such has to be supplemented with mechanisms for protecting the host environment from the untrusted application's execution. Sandboxing is the best choice of mechanism in our scenario, as it cleanly separates the application's persistent changes to the OS from the rest of the host system. TROGUARD makes use of SE-Linux platform to generate and enforce policies to keep the downloaded applications within a contained sandbox until their benign behavior is confirmed by TROGUARD. During the training phase, TROGUARD logs the executed syscalls for the applications in a functionality class and then uses the union of those logs to generate SE-Linux policies automatically. TROGUARD creates a policy ruleset by parsing the log files for each functionality class. Once the user downloads an application, TROGUARD employs the created sandbox (i.e., loads the corresponding generated policy's kernel modules) for the claimed functionality class. Consequently, the loaded sandbox policies will not allow the downloaded applications to expose behaviors not already exposed by any of the application instances within the same functionality class during the training.

5 Evaluation

We deployed TROGUARD in a testbed environment and evaluated various aspects of its operation. In particular, we designed a set of experiments to empirically determine TROGUARD's efficiency, accuracy, and discriminative power between functionality classes.

We installed TROGUARD in an Ubuntu 12.10 computer system with Intel® Core™ i7 3.6 GHz Processor and 8 GB of memory was used for the experiments. We extended and used LTTng tool [29] to trace system-level activities including the semantic information such as filenames and network protocols. The Weka framework [12] was used for functionality classification of running applications.

SAMAN

5.1 Website Content Classification Accuracy

We compare the accuracy of pure HTML document analysis and pure image OCR analysis by testing them in 100 popular download websites such as `Download.com`, `Softpedia.com`, `Tucows.com`, etc. We visited each download web page trying to download a

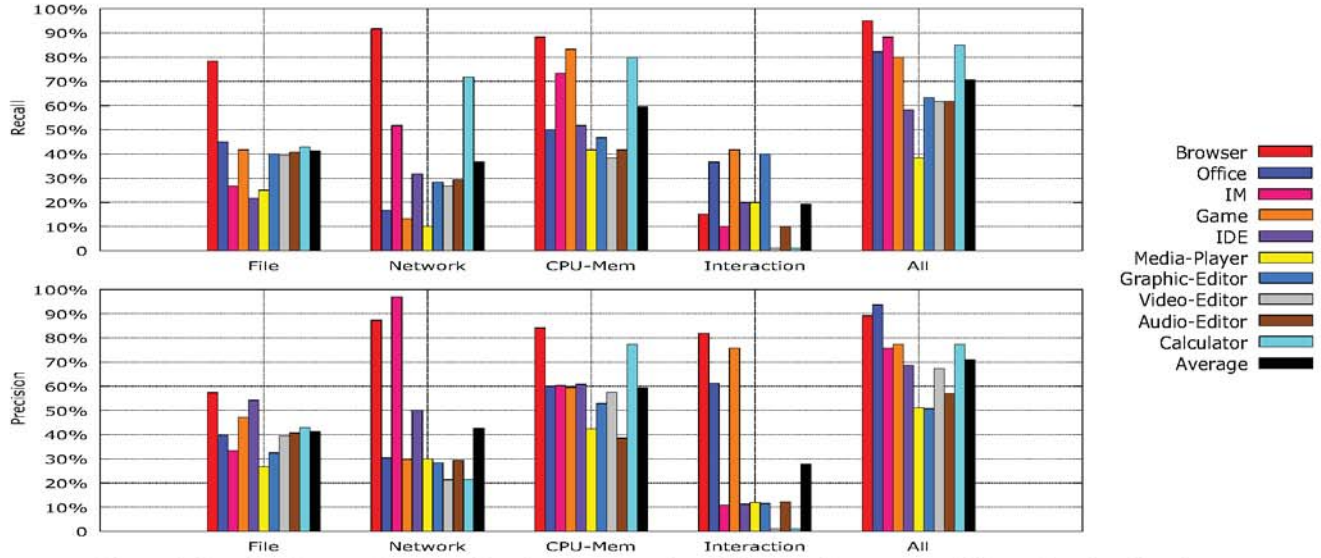
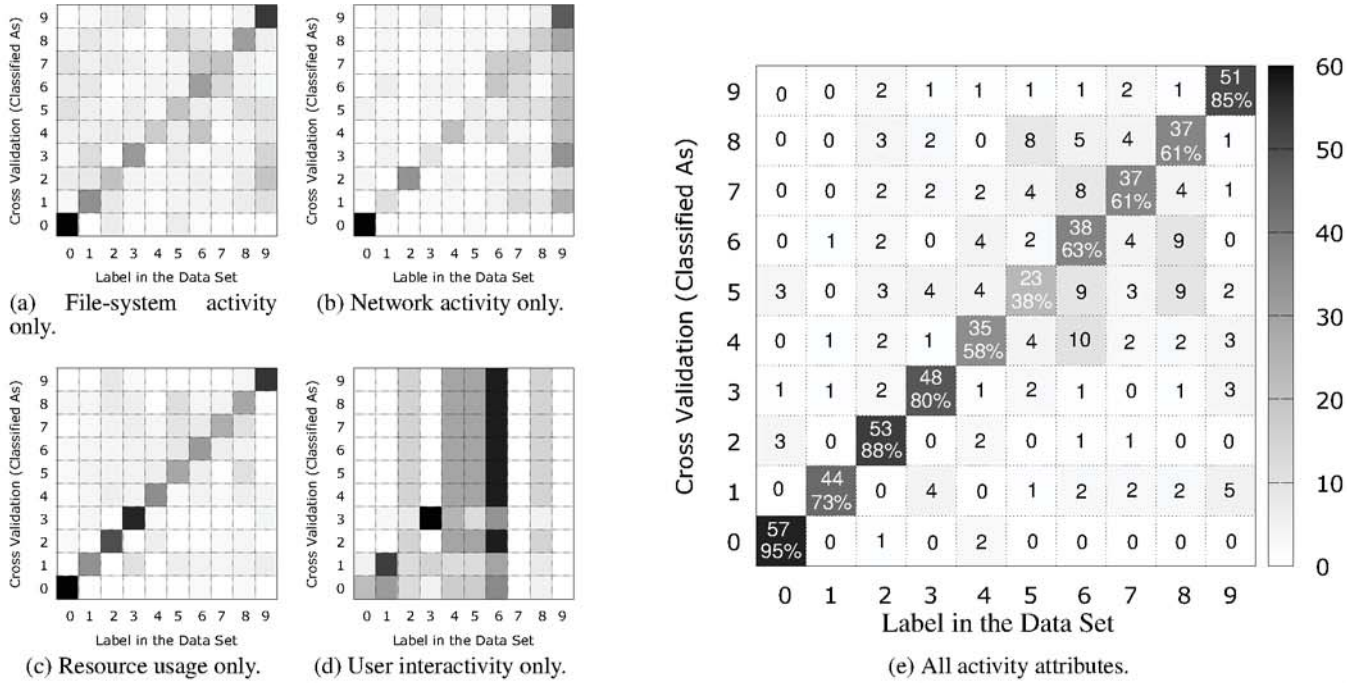


Figure 6: Classification recall and precision for detectors using different attribute sets on different functionality classes.



(a) File-system activity only.

(b) Network activity only.

(c) Resource usage only.

(d) User interactivity only.

(e) All activity attributes.

Figure 7: Confusion matrices for a variety of detectors. Figures (a)–(d) describe class-specific detectors, each using a particular class of activity attributes. Figure (e) describes a detector using all attributes. A darker shade of grey in a cell indicates that the detector had higher confidence in classifying a program from the given row as the part of the functionality class from the given column. A perfect detector produces a completely black diagonal confusion matrix.

random application. Figure 5 presents the accuracy results for the website content analysis in TROGUARD. The vertical axis presents the accuracy value based on the portion of cases where a functionality class was determined correctly using the automated website analysis engine, i.e., the automatically determined class matched the functionality class that the user believed was downloading. We studied the accuracy of text-based and image-based functionality class inference separately. For image-based analysis, TROGUARD uses the open source Tesseract OCR engine [28]. In particular, the engine took screen shots of the download website before performing an in-depth OCR analysis to search for the target key terms, e.g., the functionality class names and related keywords. As shown in the figure, TROGUARD's pure text-based analysis engine outperforms the image-based website analyzer in almost all of the cases except the IDE functionality class.

5.2 Application Classification Accuracy

To evaluate the application classification in TROGUARD, we installed 100 applications[1] chosen from ten different functionality classes (10 application per functionality class) and collected data corresponding to individual attributes. During the off-line training stage, each application ran for 60 seconds while a user exercised the application's main functionalities. It is noteworthy that, after several trials, we found the 60-second interval sufficient to capture

[1]The applications were chosen from Ubuntu Software Center and www.Softpedia.com.

the main functionalities of each application according to its functional category. Each application's data log was divided into six time periods (10 seconds each), to be used later for testing phase. Consequently, our collected database composed of 600 labeled data points.

We evaluated the performance of application type learning and classification component in TROGUARD using 10-fold cross validation [3]. We considered two statistical measures, namely 1) recall, the fraction of relevant instances that are retrieved during the classification; and 2) precision, the fraction of retrieved instances (i.e., marked as a particular functionality class by TROGUARD) that are correct (i.e., actually of the marked class).

Figure 6 shows the recall values for each functionality class as a result of classification using different subsets of attributes. The employed attributes fall into four categories, file system-related (File), network-related (Network), system resource-related (CPU–Mem), and user interactivity-related (Interaction). We also present results of the classification using all of the attributes. The results for each subset of attributes are a clustered set of bars, each representing the recall value for an individual functionality class. As the results show, the browser functionality class gives the highest recall percentage except when the classification is performed using the Interaction attributes only. This is because various web pages require different kind of user interactions. As a case in point, the way users interact with an online game website, i.e., intense keyboard and mouse interaction, is extremely different from that of a text or video news channel, i.e., minimal interaction while the video is being played. Therefore, many browser samples have been wrongly marked as other functionality classes such as graphic editor that require similar mouse activity. Another interesting functionality class is the game class whose most distinguishing characteristic are its CPU–Mem attributes because games usually require high CPU and memory resources in comparison to other functionality classes. The network attributes are the weakest for distinguishing game samples as they do not make regular connections to network very often.

Figure 6 also shows the precision results for the application type learning and classification component of TROGUARD. As can be seen, using all of the attributes results in application marking with a higher accuracy. Comparing the recall and precision results using Interaction attributes for the browser class, the classifier has been selective in marking application samples as the browser functionality class resulting in high precision. Additionally, many of the graphic editor, media player and IDE samples are marked falsely as browsers resulting in a poor precision rates. Based on both recall and precision results, the user interactivity attribute contributes the least to the overall accuracy. In addition to the use of different GUI schemes in applications, this may be attributed to several applications executing in the background while they are not the active window on screen, i.e., zero user interactivity.

Figure 7 shows the accuracy results from a different perspective, providing more details on how applications were classified correctly or confused by mistake by the TROGUARD framework. In particular, we present one $10x10$ gray-scale confusion matrix for each set of attributes, e.g., File. To further clarify, each (i, j) element in a confusion matrix represents how many of application data points of the i-th functionality class was marked (classified) as the j-th functionality class[2]. Therefore, the diagonal elements denote the samples that were classified correctly. Clearly, out of the four sets of attributes, CPU–Mem attributes are the most discriminative, while Network attributes do relatively well except for the data points labeled as the calculator functionality class. Regarding the least contributing attribute set (Interaction), as the Media Player applications do not require high user interaction, most of their corresponding data points (column 6) have been classified wrongly.

[2]The functionality class numbering is as follows: Browser (0), Office (1), Game (2), IDE (3), IM (4), Graphic Editor (5), Media Player (6), Video Editor (7), Audio Editor (8), and Calculator (9).

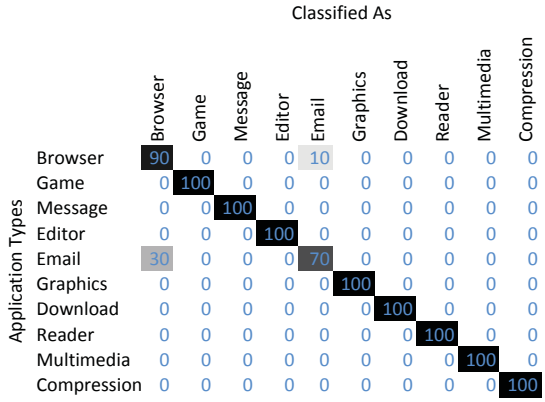


Figure 8: Classification Using Intermediate Features

Table 2: Training times (seconds)

	File	Network	CPU-Mem	Inter.	All
#Attr.	44	20	8	9	81
Time	0.49	0.14	0.19	0.4	0.82

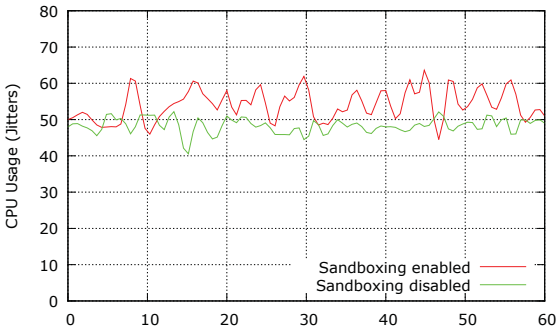


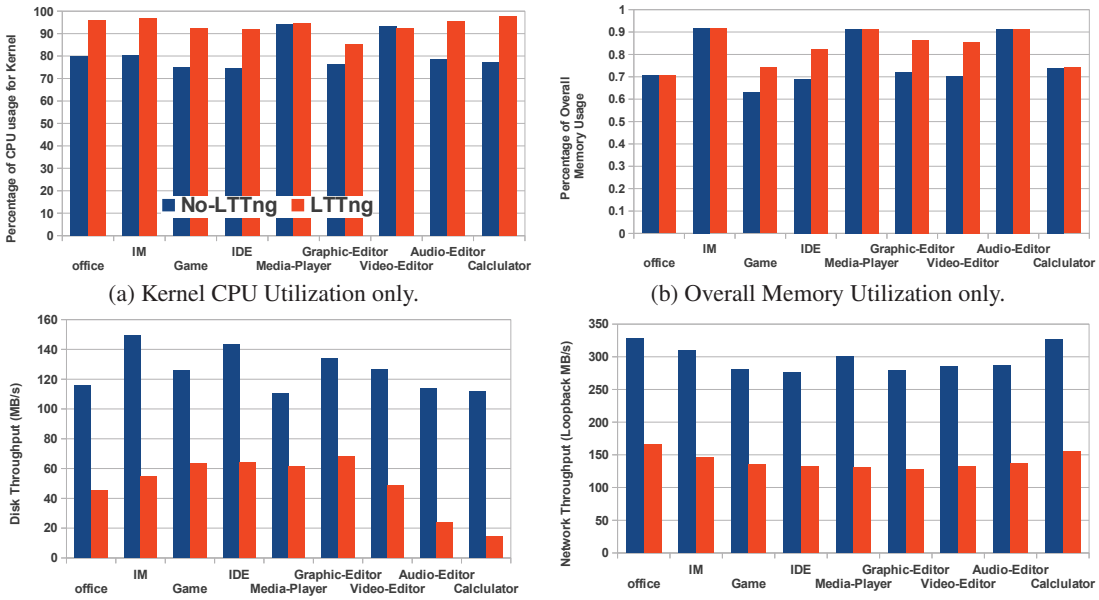
Figure 9: Sandboxing CPU Usage Overhead

On the other hand, we can see accurate classification results for IDE applications with high user interaction.

Finally, we measured how much the usage of intermediate-level features improved the classification accuracy (see Figure 8). For intermediate-level feature-based functionality classification, TROGUARD uses Debtags [32] that is a large set of Debian packages, approximately $39K$ packages, each labeled manually with many *facet* features by developers to facilitate application package search by high level application functionality tags. We grouped the packages into 10 functionality classes automatically using logical rules on combination of facet features, and then verified the automatically categorized packages one by one manually for correctness. As a case in point, we used the following rule for the browser category `browser:|use::browse|format::html|protocol::http`, i.e., the package is a browser type if its facet feature set include any of the three listed features. We later used the abovementioned functionality class labels for supervised learning using the intermediate-level facet features of each Debtags data point (application). The computed learning rules were later used for trojan detection and classification of legitimate Debtags and non-Debtags applications. The final average accuracy was 96% compared to 70% (see Figure 7) that empirically proves the benefit of intermediate-level features as an approach to fill the semantic gap between the high-level functionality classes and low-level system traces.

5.3 Sandboxing Performance

We measured how much overhead TROGUARD's generated SELinux policies put on the system once the user downloads the application. Figure 9 shows the results. On average TROGUARD's sandbox puts 5.4% overhead on the system throughput. It is noteworthy that the overhead is temporal and will go away once TROGUARD



(a) Kernel CPU Utilization only.

(b) Overall Memory Utilization only.

(c) Disk Throughput (Loopback) only.

(d) Network Throughput only.

Figure 10: TROGUARD Overhead on System Resources

confirms that the application is benign and could run outside of the SE-Linux sandbox.

5.4 Application Classification Performance

During the application type learning phase, TROGUARD saved the collected data for the 100 applications in human readable plain text files (totaling 40.5 GB). TROGUARD built 5 models using the meta classifier Ensembles of Nested Dichotomies (END) in the Weka machine learning suite [12], based on the C4.5 decision tree classification algorithm [12]. The algorithm can handle both continuous (numeric) and discrete (nominal) attributes. TROGUARD builds the decision tree with 2 as the minimum number of instances per leaf. Additionally, TROGUARD uses 1 fold for reduced-error pruning, and 2 extra folds for growing the tree. Table 2 shows the performance results for the application classification engine. As shown the total 81 attributes were distributed among the four attribute sets. The second row shows how long it takes for TROGUARD to complete the supervised learning and training phases using all of the application data logs. The first four columns show the time requirement for classification using a subset of attributes, and the last column reports the overall time required for creating trained models using all of the attributes. As shown, the training phase using the all the attributes takes less than 1 second to complete that is an acceptable duration for real-world deployment of the TROGUARD framework.

5.5 Performance Overhead

We studied the performance overhead of data collection tool LTTng kernel tracer. The LTTng kernel tracer is a highly efficient tracing tool that typically used by system developer to track down performance issues and debug problems in multi-process system. The reported impact of LTTng on kernel operations in terms of percentage of CPU cycles against vanilla kernel is less then 5% in [2].

We measured the impact of TROGUARD syscall interception engine on various aspects of the system performance, namely kernel CPU utilization, memory utilization, disk throughput, and network throughput. In addition, we benchmarked the network throughput over loopback. Figure 10 shows the results. In our experiments, the maximum CPU overhead TROGUARD's kernel tracer introduced is 20% for the calculator. The maximum memory overhead is 15% for running video editor. The measured performance overhead for disk throughput ranges from 45% to 87%. The fairly high disk operation overhead of TROGUARD is due to its logging activities to the system's disk in parallel. The network performance degradation ranges from 55% to 60%. It is noteworthy that these overheads are not permanent and go away once TROGUARD comes to a conclusion on whether the running application is a trojan.

5.6 Behavioral Mimicry Attacks

One of the main challenges that behavioral-based detection solutions face generally is their vulnerability against behavioral mimicry attacks. We evaluated how TROGUARD performs in terminating such attacks that try to pretend delivering a legitimate functionality while conducting a malicious activity in the background. First, we describe a concrete case study to show step by step how TROGUARD detects a sample trojan `Freesweep` game application that is downloaded from a legitimate-looking malicious website. Next, we will present our results on 50 other mimicry attack samples.

We obtained the `Freesweep` game application trojan payload using the Metasploit suite [20] as a Ubuntu deb game package. The trojan package included the original game, the exploit payload, and a post-installation script that executed the exploitation script. Upon execution, the malicious payload opened a socket connection secretly and created a reverse shell to connect back to an adversarial remote site.

We masqueraded a `Softpedia` web page as shown in Figure 11 to hold the malicious trojan game package. Consequently, the victim user was convinced to download, install and run the game on her desktop computer. Once the victim clicked to download the package, TROGUARD's extension component noticed the download action request in the browser. TROGUARD analyzed the download website and correctly estimated the functionality class, i.e., game. The determined functionality class was shown to the user through a browser popup (similar to Figure 4) with the default option set to game. After the user's confirmation, TROGUARD switched to the monitoring mode to trace and capture the application's execution footprint. Analysis of the logged data sets using the 100-application trained model resulted in classification of most of the downloaded application's data points as calculator. Figure 12 shows the calculated probability values for each functionality class, i.e., the probability that the downloaded application is of a particular functionality class. The mismatch between the confirmed func-

Figure 11: Game Trojan's Download Webpage (subsection 5.6)

Functionality-Class Mismatch:	
Expected:	Game
Observed:	10% Game, 50% Calculator

Figure 12: TROGUARD Alert

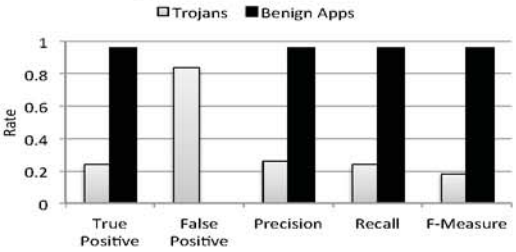


Figure 13: TROGUARD Against Mimicry Exploits

tionality class and the derived class caused TROGUARD to raise a warning notifying the user of the mismatch and a potential malicious download.

We evaluated TROGUARD on 50 mimicry trojan samples created by reverse engineering and malicious code patching of the benign pdf readers. Figure 13 shows the classification results of how accurately the mimicry trojans are classified compared to their corresponding benign applications. The ratings are averaged over all trojan samples for each classification parameter that show lower classification accuracy when TROGUARD classifies a trojan. For instance, TROGUARD classified 90% of benign pdf-reader applications correctly, in contrary, only 10% of mimicry trojans that pretended to be a pdf-reader application was classified as a pdf-reader by TROGUARD. All others were classified as Because of such low classification accuracies (below the predefined acceptance threshold 0.8), TROGUARD was able to detect and terminate all 50 trojan samples.

6 Related Work

Drive-by Download attack prevention. Lu et al. [18] presented a drive-by download defense system built on the principle of preventing the execution of binary content to which the user did not consent. They correlate user download authorizations to identify unconsented downloaded content and then redirect disk I/O to prevent execution of that content. The obvious limitation for their techniques is that it does not prevent social engineering attacks where the user consents to the download and installation of malware disguised as benign applications. Thus our approach complements defenses against drive-by downloads.

Behavior-based malware analysis and classification. Bayer et al. [4] and Lanzi et al. [15] perform dynamic analysis to generate program profiles from execution traces. Bayer executes binaries in an emulated environment to produce a behavior profile in terms of OS objects, OS operations that are carried on those objects, dependences and comparisons between OS objects, which are summarized from the collected system call traces. To add more discriminative features, Bayer also includes network analysis which extracts high-level semantic operations from low-level socket system calls such as names of download files, name of IRC channels, or mail subjects. Although the malware clustering accuracy can reach 97% using this feature set, the feature set cannot be applied to our software application clustering engine. In the whole computer software category, malware behavior is a subset of all the program behaviors. We have to compose a feature set that discriminative with respect to all category of software. Because our mechanism is not solely to detect malicious application, we are distinguishing the advertised application type from its genuine type. Therefore our features are higher level summaries that better reflect program activity.

Lanzi [15] propose a system centric detection model, in which it capture how a broad set of diverse applications interact with OS resources such as file system and registry entries. A single policy derived from file system and registry entries access pattern can be used to distinguish between malicious and benign applications. However, this unsupervised method for constructing access pattern is insufficient for our purposes as it cannot be mapped to the functionality perceived by the user from the web page description. These and other works in a similar vein [5,8,13,16,25,26] all share the standard malware-detection approach of ignoring the user and detecting malware based on predefined behaviors. In contrast, our system focuses on detecting inconsistencies between user expectations and program executions.

General program behavior profiling. Lo et al. [17] design a system for finding program bugs or abnormal code runs caused by intrusions. In their terminology, a software behavior is a series of events including execution of a statement, a method call, or a basic block in a control flow graph. The technique is not scalable enough for real time use, in contrast to TROGUARD. Okazaki et al. [24] introduced early their work on process profiling from system call traces to deploy an anomaly-based intrusion detection system. The profile is simply a record of system call types and their observed frequencies. Although efficient, their approach is not resilient to circumvention by attackers, who can easily design their trojan to perform any required number of system calls to match a desired profile. Zhang [31] proposed a system to improve the resource allocation in grid computing based on whether the application belongs to CPU intensive, I/O and paging intensive, network intensive, or idle. Although they demonstrate their work improved the throughput in schedule tasks in clusters, it is unclear that the same characteristics are useful in a security setting.

7 Discussion

We discuss various aspects and the potential deployment limitations of the TROGUARD framework.

How representative are the functionality classes in TROGUARD*?* Coming up with a comprehensive, meaningful and universal set of functionality classes is challenging, and at the same time, critical to the TROGUARD's correct operation. To this end, we have identified and used a common set of functionality classes from three popular software-download web sites and the Debian package tags that host a large comprehensive set of software applications and are viewed by millions of users every day (subsubsection 4.1.1).

How usable can TROGUARD *be for non-technical users?* To provide the users with a usable security solution, one of our main objectives in designing every component in TROGUARD has been to separate the low-level system details that are used for various analyses from the high-level information that is communicated to the user. As a case in point, while the kernel-level modules are tracing different system-level activities of the suspicious application, the user's interaction with TROGUARD is through the browser

extension using terms that the (potentially non-technical) user usually sees on popular download websites and is familiar with. In the simplest form, the user only has to confirm that the application class diplayed by the browser extension matches the class he expected based on what he saw and read on the download page. We believe that such design points has made TROGUARD usable by a vast range of users. Of course, this belief has to be validated through an actual usability study, and we intend to pursue such a study in our future work, pending IRB approval.

Is TROGUARD *useful even if it is not absolutely accurate?* Semi-automated detection of social engineering attacks using mostly system-level information has remained to be a very challenging problem that is also indicated by their recent increasing popularity among attackers. TROGUARD introduces a new solution against web-based socially engineered trojan attacks through categorization of applications' functionality classes rather individual applications (developed by possibly unknown third-parties). Additionally, as our implementation is not yet optimized, both the website analysis and application classification accuracies can be improved through parameter tuning as well as usage of more complete techniques such as static executable analysis techniques to estimate the application's functionality class.

8 Conclusions

We presented TROGUARD, an effective system solution to protect the client systems against web-based trojan attacks that make use of social engineering techniques to convince victim users to willingly download and execute a legitimate-looking malicious software. Our results over a large set of applications show that TROGUARD can effectively bridge the gap between the high-level user perceptions and low system-level execution traces in order to block trojan intrusions efficiently under real-world adversarial situations.

Acknowledgements

We appreciate the anonymous reviewers and thank our sponsor, the Office of Naval Research (Grant N000141210462).

9 References

[1] Trojanhunter; available at www.trojanhunter.com, 2013.

[2] R. G. Anjoy and S. K. Chakraborty. Efficiency of lttng as a kernel and userspace tracer on multicore environment. Technical report, 2010.

[3] S. Arlot and M. Lerasle. V-fold cross-validation and v-fold penalization in least-squares density estimation. 2012.

[4] U. Bayer, P. M. Comparetti, C. Hlauschek, C. Krügel, and E. Kirda. Scalable, behavior-based malware clustering. In *NDSS*. The Internet Society, 2009.

[5] U. Bayer, I. Habibi, D. Balzarotti, E. Kirda, and C. Kruegel. A view on current malware behaviors. In *Proceedings of the 2nd USENIX conference on Large-scale exploits and emergent threats: botnets, spyware, worms, and more*, LEET'09, pages 8–8, Berkeley, CA, USA, 2009. USENIX Association.

[6] N. Carlini, A. P. Felt, and D. Wagner. An evaluation of the google chrome extension security architecture. In *Proceedings of the 21st USENIX conference on Security symposium*, Security'12, pages 7–7, Berkeley, CA, USA, 2012. USENIX Association.

[7] M. Christodorescu, S. Jha, and C. Kruegel. Mining specifications of malicious behavior. In *6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 5–14, 2007.

[8] S.-Y. Dai, Y. Fyodor, M.-W. Wu, Y. Huang, and S.-Y. Kuo. Holography: a behavior-based profiler for malware analysis. *Software: Practice and Experience*, 42(9):1107–1136, 2012.

[9] P. J. Denning and R. D. Riehle. The profession of it is software engineering engineering? *Communications of the ACM*, 52(3):24–26, 2009.

[10] M. Desnoyers and M. Dagenais. The lttng tracer: A low impact performance and behavior monitor for gnu/linux. In *Proceedings of the 27th Annual ACM Symposium on Applied Computing*, pages 354–359. ACM, 2012.

[11] L. Garber. Security, privacy, and policy roundup. *IEEE Security & Privacy*, pages 15–17, 2012.

[12] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten. The weka data mining software: an update. *ACM SIGKDD Explorations Newsletter*, 11(1):10–18, 2009.

[13] E. Kirda, C. Kruegel, G. Banks, G. Vigna, and R. A. Kemmerer. Behavior-based spyware detection. In *Proceedings of the 15th conference on USENIX Security Symposium - Volume 15*, USENIX-SS'06, Berkeley, CA, USA, 2006. USENIX Association.

[14] C. Kuo, F. Schneider, C. Jackson, D. Mountain, and T. Winograd. Google safe browsing. project at google. *Inc., June–August*, 2005.

[15] A. Lanzi, D. Balzarotti, C. Kruegel, M. Christodorescu, and E. Kirda. Accessminer: using system-centric models for malware protection. In *Proceedings of the 17th ACM conference on Computer and communications security*, CCS '10, pages 399–412, New York, NY, USA, 2010. ACM.

[16] T. LEE, J. J. MODY, Y. L. LIN, A. M. MARINESCU, and A. A. POLYAKOV. Application behavioral classification, 06 2007.

[17] D. Lo, H. Cheng, J. Han, S.-C. Khoo, and C. Sun. Classification of software behaviors for failure detection: a discriminative pattern mining approach. In *Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining*, KDD '09, pages 557–566, New York, NY, USA, 2009. ACM.

[18] L. Lu, V. Yegneswaran, P. Porras, and W. Lee. Blade: an attack-agnostic approach for preventing drive-by malware infections. In *Proceedings of the 17th ACM conference on Computer and communications security*, CCS '10, pages 440–450, New York, NY, USA, 2010. ACM.

[19] T. . malicious programs for Mac OS X. Mcafee antivirus solution; available at http://www.securelist.com, 2012.

[20] D. Maynor. *Metasploit Toolkit for Penetration Testing, Exploit Development, and Vulnerability Research*. Syngress, 2007.

[21] McAfee. Mcafee antivirus solution; available at http://www.mcafee.com, 2013.

[22] A. Moser, C. Kruegel, and E. Kirda. Exploring multiple execution paths for malware analysis. In *Proceedings of the 2007 IEEE Symposium on Security and Privacy*, SP '07, pages 231–245, Washington, DC, USA, 2007. IEEE Computer Society.

[23] R. Naraine. Adobe: Beware of fake flash downloads; available at http://www.zdnet.com, 2008.

[24] Y. Okazaki, I. Sato, and S. Goto. A new intrusion detection method based on process profiling. In *Applications and the Internet, 2002. (SAINT 2002). Proceedings. 2002 Symposium on*, pages 82 –90, 2002.

[25] K. Rieck, T. Holz, C. Willems, P. Düssel, and P. Laskov. Learning and classification of malware behavior. In *Proceedings of the 5th international conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, DIMVA '08, pages 108–125, Berlin, Heidelberg, 2008. Springer-Verlag.

[26] K. Rieck, P. Trinius, C. Willems, and T. Holz. Automatic analysis of malware behavior using machine learning. *J. Comput. Secur.*, 19(4):639–668, dec 2011.

[27] J. H. Saltzer and F. Kaashoek. *Principles of computer system design: an introduction*. Morgan Kaufmann Pub, 2009.

[28] R. Smith. An overview of the tesseract ocr engine. In *Proc. Ninth Int. Conference on Document Analysis and Recognition (ICDAR)*, pages 629–633, 2007.

[29] D. Toupin. Using tracing to diagnose or monitor systems. *Software, IEEE*, 28(1):87–91, 2011.

[30] Wikipedia. Computer keyboard — Wikipedia, the free encyclopedia, 2013.

[31] J. Zhang and R. J. Figueiredo. Application classification through monitoring and learning of resource consumption patterns. In *Proceedings of the 20th international conference on Parallel and distributed processing*, IPDPS'06, pages 144–144, Washington, DC, USA, 2006. IEEE Computer Society.

[32] E. Zini. A cute introduction to debtags. In *Proceedings of the 5th annual Debian Conference*, pages 59–74, 2005.

Spam Ain't As Diverse As It Seems: Throttling OSN Spam With Templates Underneath

Hongyu Gao*, Yi Yang*, Kai Bu*, Yan Chen*, Doug Downey*, Kathy Lee*, Alok Choudhary*
*Department of Electrical Engineering and Computer Science, Northwestern University
*College of Computer Science and Technology, Zhejiang University

ABSTRACT

In online social networks (OSNs), spam originating from friends and acquaintances not only reduces the joy of Internet surfing but also causes damage to less security-savvy users. Prior countermeasures combat OSN spam from different angles. Due to the diversity of spam, there is hardly any existing method that can independently detect the majority or most of OSN spam. In this paper, we empirically analyze the textual pattern of a large collection of OSN spam. An inspiring finding is that the majority (63.0%) of the collected spam is generated with underlying templates. We therefore propose extracting templates of spam detected by existing methods and then matching messages against the templates toward accurate and fast spam detection. We implement this insight through *Tangram*, an OSN spam filtering system that performs online inspection on the stream of user-generated messages. Tangram automatically divides OSN spam into segments and uses the segments to construct templates to filter future spam. Experimental results show that Tangram is highly accurate and can rapidly generate templates to throttle newly emerged campaigns. Specifically, Tangram detects the most prevalent template-based spam with 95.7% true positive rate, whereas the existing template generation approach detects only 32.3%. The integration of Tangram and its auxiliary spam filter achieves an overall accuracy of 85.4% true positive rate and 0.33% false positive rate.

Categories and Subject Descriptors

J.4 [**Computer Applications**]: Social and behavioral sciences

General Terms

Human Factors, Measurement, Security

Keywords

Online social networks, spam, spam campaigns

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.
ACSAC '14, December 08 - 12 2014, New Orleans, LA, USA
Copyright 2014 ACM 978-1-4503-3005-3/14/12 ...$15.00
http://dx.doi.org/10.1145/2664243.2664251.

1. INTRODUCTION

Spam has been plaguing the Internet community for more than a decade. With the tremendous popularity of online social networks (OSNs) in recent years, spammers quickly start to exploit this new media channel. Despite the development of countermeasures, spammers find their way to adapt and stick. Research reveals that on Twitter, one of the most popular OSNs nowadays, more than 4% of collected tweets are spam [27], which has slipped through all the deployed defense mechanisms. Researchers have proposed to combat OSN spam from multiple angles, including mining the textual content [6], studying the redirection chains of embedded URLs [15] as well as classifying the URL landing pages [26]. Despite the effort to build OSN spam mitigation systems, it is not yet clear what techniques the spammers are using to construct OSN spam, and how the techniques evolve. This missing piece of information plays a crucial role to inspire effective designs to throttle OSN spam.

In this paper, our first contribution is to reveal the OSN spam generation techniques according to spam textual patterns (**Section 2**). We conduct a measurement study, identifying 115 campaigns from more than 500 thousand spam messages. We find that the majority of spam (63.0%) is generated with underlying templates, which is consistent with prior email spam research [12, 13, 22]. Templates are valuable for spammers, because they let spammers control and customize the semantic meaning of generated messages to boost the conversion rate. Meanwhile, they generate diversified messages that are more difficult to detect. OSN spammers have evolved to use more sophisticated templates that break the assumptions in prior email spam template generation research, making them ineffective for OSN spam. In addition, the spam without underlying templates calls for effective countermeasures as well.

In particular, we identify three major challenges that render existing spam template generation techniques ineffective.

1. **Absence of invariant substring in template.** Prior spam template generation research [22,33] made a crucial assumption that an invariant substring is hard-coded in a template, so that every instantiation of the template contains such string. Unfortunately, an OSN spam template does not always contain any invariant substring.
2. **Prevalence of noise.** Spammers extensively add semantically unrelated noise words into spam messages. The presence of noise diversifies spam messages, and increases the difficulty to identify semantically meaningful text segments.

3. **Spam heterogeneity.** Spam instantiating different templates mixes with spam without any underlying templates. It is hard to obtain a training set with a single template in an online detection scenario.

Our second contribution is to propose Tangram, a system that performs effective template generation to combat OSN spam (**Section 3**). Many existing methods detect spammers instead of spam. These methods are based on account activity and need long observation periods for the account features to accumulate [3, 25, 29, 31]. Many other detection approaches are based on URL analysis, which inherently cannot detect spam without URLs [15, 26, 30]. Researchers have revealed significant amount of such spam [4]. The few existing methods that detect spam with or without URLs in real-time suffer from high false positive rates [4, 6, 24]. In contrast, Tangram is the first accurate online OSN spam detection system that detects spam with or without URLs. It extracts templates of spam detected by existing methods and then matches messages against the templates toward accurate and fast spam detection. In practice, it can be deployed in combination with heterogeneous detection methods, like account and URL analysis, to optimize detection accuracy. Specifically, Tangram is highly accurate because of the following four innovations.

1. **Embrace the absence of invariant substring.** We identify frequently appearing segments within messages and then locate equivalent segments among messages. Such segments are later assembled into spam templates, which are used to match future spam.
2. **Mitigate the prevalence of noise.** We cast a sequence-labeling task to label each word in a given message as either "noise" or "non-noise". Only "non-noise" words are used to generate templates.
3. **Break spam heterogeneity.** We pre-cluster spam and perform template generation within individual partitions. The template generation procedure also discards outlier messages in the partition.
4. **Build a double defense.** We mitigate spam without underlying templates using a supplementary module that detects spam with excessive semantically unrelated noise words. In an adversarial setting, spammers may deliberately copy or adapt benign content copied from other sources in attempt to avoid detection by Tangram. Since spammers need to use popular content to attract the audience and they generate a large number of spam, the OSN administrator will observe the popular content with replaced URLs in large volume, which defeats the spammers' purpose to evade detection. In addition, we can equip Tangram with multiple heterogeneous detection modules in practice. It is hard for spammers to evade detection from multiple angles simultaneously.

We evaluate Tangram's ability to detect real-time spam on a large Twitter dataset of 17 million tweets (**Section 4**). Equiped with one necessary auxiliary spam filter, Tangram detects 95.7% of the most prevalent template-based spam. The complete system detects Twitter spam with 85.4% true positive rate and 0.33% false positive rate. The true positive rate is similar with existing systems [4, 6, 24], while the false positive rate is at least 2 times better than that in [6] and 10 times better than [4, 24]. Tangram reacts to newly emerged spam campaigns quickly. Finally, we investigate Tangram's robustness (**Section 5**) and related work (**Section 6**).

2. MOTIVATION: REVEALING TWITTER SPAM TEMPLATE

In this section, we empirically analyze the textual patterns of Twitter spam as a first attempt to quantitatively reveal popular techniques that generate current OSN spam. We find that the majority of spam is generated by underlying templates. This finding motivates Tangram.

2.1 Data Collection

We first collected a large dataset from Twitter that contained about 17 million public tweets generated by 4.2 million users. The tweets were generated between June 1, 2011 and July 21, 2011. In the data collection process, we continuously downloaded popular Twitter Hashtags from the website *What the Trend* [1]. We then downloaded all public tweets that contained the Hashtags. Our data collection method was inevitably biased towards tweets containing popular Hashtags. Consequently, the spam tweets in our dataset were also biased towards spammers using hashtags. We, however, would like to emphasize that the numerical accountIDs in our dataset followed a uniform distribution, suggesting the dataset was not overly biased towards specific account groups.

We revisited Twitter in March 2012 to label the collected tweets. For each account that posted tweets in the dataset, we crafted a special URL, using which we could access the account's personal profile on Twitter. We only checked if we were redirected to a page indicating that the account had been suspended. We found that 120,386 accounts were suspended. They posted 558,706 tweets, all of which were labeled as spam. There were 532,676 unique spam tweets, showing that our collected spam had few duplicates. The other tweets were labeled as legitimate. Using the same method, we collected another relatively small tweet dataset generated during January, 2012, containing 46,844 spam tweets. The labeled spam tweets did not represent the spam that Twitter could already detect. Rather, their presence in our collected data meant that Twitter failed to detect them when they were generated[1].

Big Name A	an eye-catching action -	*URL*
Celebrity B	an eye-catching action -	*URL*
Big Name A	offensive content , look at this video	*URL*
Celebrity B	offensive content , look at this video	*URL*
RIP Celeb C	offensive content , look at this video	*URL*

Table 1: Retrofitted sample spam from a template-based campaign. (*Notes: We intentionally substituted likely offensive contents such as celebrity identities and sexuality in the example spam tweets. Interested readers are welcome to contact the authors for original datasets.*)

2.2 Template-based Spam Dominates

Through mining the textual pattern of spam, we find that the majority (63.0%) of the 2011 spam dataset shares under-

[1]The spam labeling approach assumed that suspended accounts generated no legitimate message. To validate this assumption, we randomly selected 100 suspended accounts and manually checked all their posted tweets, which were indeed spam. Although spamming was not the only reason for account suspension in Twitter, our manual checking showed that it was the overwhelmingly dominant reason.

lying templates. Table 1 shows five (intentionally retrofitted) sample spam tweets from a much larger, but typical template-based campaign. We substitute the embedded URLs with the symbol URL for brevity. The tweets in this campaign comprise three components: a celebrity name, an eye-catching action, and a URL. Each component has one or more choices of textual content. The number of unique spam messages that this template can potentially generate, therefore, increases quickly with the number of components.

Spam Template Model. We formally model the *true* spam template as a macro sequence $(m_1, m_2, ..., m_k)$. We define two types of macros: *dictionary* macros and *noise* macros. At the time of spam generation, a *dictionary* macro picks the textual content from a pre-defined list of choices (e.g., celebrity name in Table 1). It is possible for a dictionary macro to have only one choice. In this case, the macro reduces to an invariant substring that all generated messages will contain. *Dictionary* macros carry semantic meanings so that spammers can lure the recipients to visit the embedded URLs. In comparison, we abstract any macro that does not convey any semantic meaning, but purely increases the message diversity or increases the chance of exposing the spam to more users, as a *noise* macro. A noise macro can be implemented by, for example, randomly generating a character string, mentioning Twitter users, etc.; it is not our interest to differentiate among them. The concatenation of the instantiation of macros constitutes a spam message.

We assume that a template shall contain at least one *dictionary* macro, while it may or may not contain any *noise* macro. However, we do *not* assume the existence of any invariant substring. Written in human language, a spam message is not restricted to any particular expression to present a semantic meaning. We have also observed spam template without invariant substring in our data. This relaxed assumption is one of our major differences from the existing template generation work [22, 33] that relies on invariant substrings.

2.3 Template-based Spam Continues Dominating

Spam Categorization. We first split spam into two categories: "semantically similar" and "semantically dissimilar". Semantically similar spam forms big clusters that share the same semantic meaning, whereas semantically dissimilar spam does not. We further divide the "semantically similar" spam into "template-based" and "paraphrase" spam. In contrast with template-based category, *Paraphrase Category* consists of spam tweets that share the same semantic meaning but cannot be uniformly divided into semantically equivalent segments. Meanwhile, the tweets do not share regular wording, e.g., "browsing statistics click Celebrity John URL" and "interesting site on Celeb J URL". In addition, semantically dissimilar spam consists of *No-content Spam* that contains very little semantically meaningful textual content and *Other Spam* that we have not systematically categorized.

Template-based spam continues dominating other categories of spam. Table 2 provides the popularity of four spam categories in June/July, 2011 and January, 2012. Template-based spam remains to be the most popular category in 2012, with its percentage increasing to 68%. The no-content category almost vanishes. Its percentage dramatically drops to 0.3%. It is possible that the no-content category exhibits strong patterns and can be easily blocked. The increasingly popular template-based spam indicates that our detection method with focus on spam template generation is effective to combat modern OSN spam.

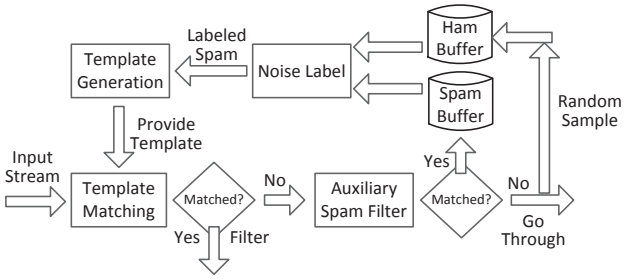


Figure 1: Tangram framework: The template generation and matching overview.

Spam Category		2011	2012
Similar Semantic	Template-based	63.0%	68.3%
	Paraphrase	14.7%	12.9%
Dissimilar Semantic	No-content	8.4%	0.3%
	Others	13.9%	18.5%

Table 2: The popularity of four spam categories in June/July, 2011 and January, 2012, respectively.

3. TANGRAM: TEMPLATE-BASED SPAM DETECTION SYSTEM

In this section, we present Tangram, a template-based spam detection system with high accuracy and speed. We first formulate the notions of template, template matching and template generation. Next, we present the details of the online Tangram system. Tangram does not need the knowledge of spam categorization (Section 2.3) as input. It only needs the stream of raw messages, plus a small number of labeled tweets for noise labeling as described in Section 3.4.

3.1 System Design Overview

Tangram builds template-based spam detection on top of existing detection methods toward higher accuracy and speed. It generates the underlying templates of spam detected by various existing methods. It then uses the templates to accurately, quickly match and detect spam. Figure 1 depicts the Tangram workflow. It takes a stream of raw messages as input, and classifies them as either spam or legitimate online. After the classification, spam is filtered, while legitimate messages pass through. Two components can classify messages: the template matching module and the auxiliary spam filter. The template matching module, along with the template generation technique, is our major contribution. The auxiliary spam filter, on the other hand, supplies training spam messages. It can be any deployed spam filter, e.g., a blacklist spam filter. We further discuss the dependence on the auxiliary spam filter in Section 5.

Template Matching and Template Generation. We define a *template* to be a sequence of macros of two types, dictionary and noise (Section 2.2). We represent a dictionary macro as a set of values separated by "|" and a noise macro as ".*". Thus, templates produced by Tangram are naturally encoded as regular expressions, specifically concatenations of "|" clauses and ".*"s. *Template matching* matches a

given message against the corresponding regular expression. A successful template match implies the tested message instantiates the template, and should be flagged as spam. We define *template generation* as the task of inferring the template's regular expression representation from a set of observed spam instances.

Initially the template matching module is not equipped with any template, so all messages will pass through. However, if a message is blocked by the auxiliary spam filter, it is treated as an instantiation of an unforeseen template, and is saved in the spam buffer. Once the number of messages in the spam buffer exceeds a predefined window size threshold t, the system invokes the template generation procedure, and deploys the newly generated templates in the template matching module. The system input is a mixture of spam instantiating different templates and spam *without* underlying templates. This differentiates our system from previous spam template generation work [22].

The template generation first identifies the subset of spam messages believed to share the same template (Section 3.3). These messages are tokenized into sequences of words. After executing noise detection (Section 3.4), we are left with spam content generated by dictionary macros. We divide every message into the same number of segments. Each segment, containing zero or more tokens, corresponds to one macro in the template. We then construct the macro by combining the segment's unique strings across messages with "|". The concatenation of the macros for all segments constitutes the complete template.

Inferring the number of segments and which tokens belong to which segment are key challenges in template generation[2]. We use the heuristic of preferring more compact templates (i.e., shorter regular expressions) that match all of the input spam messages without using wild cards. This heuristic follows the traditional approach of preferring simpler descriptions to more complex ones; our experiments validate its effectiveness. Furthermore, finding the shortest template for a set of messages is an NP-hard problem (Appendix A). We develop approximate techniques that work well in practice, as described below.

3.2 Single Campaign Template Generation

For ease of presentation, we first introduce the approach to generate a single template given spam instantiating the *same* underlying template. It is the basis of Tangram. We use the template generation process of the campaign in Table 1 as a running example to elaborate our approach. We expand the approach to generate templates on a mixture of spam instantiating multiple templates in Section 3.3.

A strength of our approach is generating templates without any invariant substring. However, we do expect that some non-trivial *subset* of a campaign will share a common substring, because the dictionary macro may instantiate to the same textual content when multiple spam messages are generated. This property helps us infer the correspondence of segments between messages. For example, if we observe the first two messages in Table 1 from a campaign, it strongly indicates that "Big Name A" and "Celebrity B" are two instantiations of the same macro. This indication holds for the campaign even if some messages do not have the substring "an eye-catching action -".

We systematically exploit such substrings shared by subsets of a campaign in three steps, *common supersequence computation*, *column concatenation*, and *regular expression representation*.

Common Supersequence Computation. The first step is to compute the messages' common supersequence. Shortest common supersequence is an NP-hard problem (Appendix A). We use an approximation algorithm named Majority-Merge [10] because of its simplicity. It takes n sequences as input, and initializes the supersequence, s, as an empty string. Next, it iteratively chooses the majority of the leftmost tokens of the input sequences, denoted as a, and appends a to s. Meanwhile, the leftmost a are deleted from the input sequences. It repeats this step until all sequences are empty, and outputs s. Each token in the output supersequence is trivially a substring shared by some subset of the campaign. Desirable substrings are i) shared by large subsets and ii) long. We achieve goal i) by producing a shorter supersequence.

We build a matrix during the execution of Majority-Merge algorithm. Table 3 shows an excerpt (due to page width limit) of such a table for the example campaign. The header row is the supersequence output by the Majority-Merge algorithm. The rest of the rows represent the input sequences, one row for each sequence. We label the j_{th} column using the token that is chosen in the j_{th} step of the Majority-Merge algorithm. The cell at row i, column j will be assigned the column label if the i_{th} sequence is picked in step j. Otherwise it will be an empty string, ε. Naturally, the concatenation of each row is exactly the corresponding input sequence. We denote this property as the *supersequence property*.

To produce a shorter supersequence, we need to merge columns that shares the same label, while maintaining the supersequence property. After merging, the cell will be assigned the column label if either cell before merging has been assigned so. Without loss of generality, we state the three sufficient conditions that determine whether column k can be merged into column j without affecting the supersequence property (Appendix B). Note that the merging is directional, after which column j is kept while column k is deleted. *Condition i*) column j and column k have identical label; *Condition ii*) in any row at least one column is ε; and *Condition iii*) if the cell at row i, column k is not ε, all cells in row i, between column j and column k must be ε. Table 4 excerpts the column merging result. Noticeably, the repeated columns of "offensive content, look at this video" is gone after the merging, yielding a more compact matrix representation.

Column Concatenation. To achieve goal *ii)* for obtaining *long* substrings shared by subsets of campaigns, we further concatenate the matrix columns obtained from the previous step. Column concatenation also operates on a column pair, after which each cell becomes the concatenation of the two corresponding cells. Different from column merging, column concatenation does not require the target columns to share identical label. It only requires that the value of the corresponding, non-ε cells in the two columns has 1-1 mapping. For example, the first two columns in Table 4 are

[2]Common techniques for segmenting text into chunks from Natural Language Processing (NLP) cannot easily apply to our segmentation task because 1) it is difficult to use standard NLP tools on OSN text [23] and 2) our segments often do not correspond to typical NLP segments such as noun phrases.

Big	Name	A	Celebrity	B	an	eye-catching	action	offensive	content	,	look	at	this	video	URL	...
Big	Name	A	ε	ε	an	eye-catching	action	ε	ε	ε	ε	ε	ε	ε	ε	...
ε	ε	ε	Celebrity	B	an	eye-catching	action	ε	ε	ε	ε	ε	ε	ε	ε	...
Big	Name	A	ε	ε	ε	ε	ε	offensive	content	,	look	at	this	video	URL	...
ε	ε	ε	Celebrity	B	ε	ε	ε	offensive	content	,	look	at	this	video	URL	...
ε	ε	ε	ε	ε	ε	ε	ε	ε	ε	ε	ε	ε	ε	ε	ε	...

Table 3: Excerpt of the initial matrix built by the common supersequence computation process. The header row is the computed supersequence. Each remaining row corresponds to one input sentence.

Big	Name	A	Celebrity	B	an	eye-catching	action	-	RIP	Celeb	C	...
Big	Name	A	ε	ε	an	eye-catching	action	-	ε	ε	ε	...
ε	ε	ε	Celebrity	B	an	eye-catching	action	-	ε	ε	ε	...
Big	Name	A	ε	ε	ε	ε	ε	ε	ε	ε	ε	...
ε	ε	ε	Celebrity	B	ε	ε	ε	ε	ε	ε	ε	...
ε	ε	ε	ε	ε	ε	ε	ε	ε	RIP	Celeb	C	...

Table 4: Excerpt of the intermediate matrix after the matrix column reduction step. The header row is a shorter supersequence.

concatenated because "Big" always maps to "Name", but the 5th and the 6th columns in Table 4 cannot be concatenated because "B" maps to two values, "an" and ε.

The effect of column concatenation is two-fold. First, it moves multiple tokens into one cell, which helps reveal the true template by assembling tokens (words) into word phrases. For example, the three separate columns "Big", "Name", and "A" in Table 4 become one celebrity name. Second, the cells on the same column after column concatenation may have different contents. This maps to the dictionary macro case, where different cell contents are different instantiation of the dictionary macro.

Regular Expression Representation converts the matrix into a regular expression to represent the generated template. We initialize the regular expression representation to be an empty string, s. Then we iterate through each column. If all the cells in the column share an identical value, we append the value to s. Otherwise, we make a "|" clause by concatenating all the unique values with "|", and append the clause to s. The generated regular expression representation from Table 4 is ˆ(Big Name A | Celebrity B | RIP Celeb C) (offensive content , look at this video | an eye-catching content -) URL$, where "ˆ" and "$" respectively mark the beginning and the ending of a message. The output regular expression matches all input sequences that build this matrix.

3.3 Multi-campaign Template Generation

We now expand single campaign template generation to multi-campaign template generation over spam instantiating *different* templates or even without underlying templates. An intuitive solution is to first separate the spam into distinct campaigns automatically, then invoke the single campaign template generation procedure on each one of them. We use a clustering and refining approach to this task.

We first use single-linkage clustering to group messages that share at least k consecutive identical tokens, k being a system parameter. The goal is to put semantically similar messages in the same cluster, while separating semantically different messages into different clusters. The transitive closure of these links forms our initial clustering. This clustering does *not* require every message pair in the cluster to share an invariant substring. We use a small training set of collected spam tweets to choose the value of k experimentally, before applying the system on the much larger dataset. The value of k is not sensitive to the training set size. For example, we test with size 10,000, 5,000 and 2,000 and obtain consistent results. With a training set of size 10,000, a loose threshold (e.g., $k = 3$) results in a big cluster containing 42% of the spam, while spam messages in this cluster have different semantic meanings like Lady Gaga, Apple product and so on. A tight threshold (e.g., $k >= 5$) results in a large number of small clusters, where multiple clusters share the same semantic meaning. For example, 9 out of the 20 largest clusters in the experiment should be merged. In comparison, $k = 4$ produces the best result in our experiments.

We then refine the clusters using the single campaign template generation algorithm. Intuitively, spams from different campaigns will result in non-compact templates, a fact we utilize to identify which spam should be removed from a given cluster. Specifically, during template generation, we repeatedly remove the row with the largest number of ε, because it likely results from a clustering error. A column will be removed if all its non-ε cells are removed. We repeat this process until the matrix contains sufficiently low number of ε.

3.4 Noise Labeling

Spam tweets often mention other users, popular terms and hashtags that are unrelated to the semantics of the rest of the tweet. Earlier research has confirmed this phenomenon [8]. Such content helps to expose spam to a larger audience, because users may search or browse tweets by topic. It also diversifies spam and increases the difficulty to detect spam. We refer to this type of content as *noise*. Popular forms of noise include celebrity names, TV shows, trending hashtags and many others. We next elaborate how noise affects template generation and design a model to automatically label noise given a small amount of easily, manually labeled noise as trained data. Once trained well, the model can accurately label noise tokens in real-time stream of spam tweets during Tangram execution.

Noise creates extra difficulties for template generation. If the generated template contains a segment of noise, the template will be too "specific", in the sense that it cannot match the spam with a different sequence of noise terms. In addition, spam instantiating different templates may coincidentally share an identical sequence of noise terms. It increases the chance to mislead the template generation module so that it attempts to extract a single template for them. Thus,

we first perform a pre-processing step to identify noise tokens in the tweet, and then effectively ignore them when generating the template (i.e., we replace them with .*, a wildcard that matches anything).

We treat noise detection as a sequence labeling task, in which the goal is to automatically label each token in the tweet as noise or non-noise. We employ a standard sequence-labeling approach, Conditional Random Fields (CRFs) [14]. The CRF is a model, learned from training data, that infers a label for each token in a given tweet. The model exploits regularities in the features of noise and non-noise tokens (detailed below), as well as regularities in label sequences.

The CRF requires identifying a set of features for each token that are relevant to the task. In our case, we found a set of features that appear to be highly indicative of noise. The key observation is that noise terms are popular, yet unrelated to each other and to other elements of the tweet. As a result, we would expect regions of noise to contain individual tokens that are common on Twitter, but transitions between tokens that are relatively uncommon. We capture these intuitions with three numeric features. Let $freq(s)$ represent the frequency of a string s, which we measure of a large set of unlabeled tweets. For each token t_i in a tweet, we create the following three features in the CRF: $freq(t_i)$, $freq(t_i t_{i+1})^2/(freq(t_i)freq(t_{i+1}))$, and $freq(t_{i-1}t_i)^2/(freq(t_{i-1})\ freq(t_i))$. The first feature captures the popularity of the token t_i, whereas the second and third estimate how likely t_i is to occur given the surrounding tokens. We processed these features into five discrete quantiles for incorporation into the CRF.

We further add four orthographic features to capture common elements of noise terms. They indicate whether t_i is capitalized, is numeric, is a hashtag, or is a user mention (i.e. using @).

To train our CRF, we hand-labeled 1,000 tweets as training data, manually identifying each token as noise or non-noise. We then employed this learned model on each tweet before template generation. In a separate experiment on the labeled tweets, we found that our trained CRF correctly labeled an average of 92% of test-set tokens as noise or non-noise.

Detection based on noise labeling. Besides pre-processing tweets to facilitate template generation, noise labeling can also directly detect spam from another angle. Intuitively, we expect legitimate messages to have very few semantically unrelated noise terms, whereas spam contains much larger number of noise terms. We design a straightforward idea to use the percentage of noise terms in the message to distinguish spam. Our system classifies a tweet as spam if its percentage of noise terms is larger than a threshold t. In the experiments we set t to be 75%. This threshold is relatively high and conservative, because we want to minimize the false alarm on legitimate tweets.

4. EXPERIMENTS

We evaluate Tangram using the labeled dataset in Section 2.1 as ground truth. The two major metrics that we use to evaluate the system are accuracy and speed. We conduct a strict accuracy evaluation. We only count spam caught by template matching or noise detection as true positives. We count spam missed by these two but caught by the auxiliary spam filter as false negatives. In this way, we are only evaluating the detection accuracy of the modules proposed in this paper, not the accuracy of the auxiliary spam filter. For speed, we evaluate the template generation and matching latency. We feed the system with the collected tweets obeying their timestamp order to reflect the performance in real-world scenario. We conduct all experiments on a server with an eight-core Xeon E5520 2.2GHz CPU and 16GB memory.

Tangram needs an auxiliary spam filtering module to provide the initial set of spam messages to construct the underlying template. We leverage an existing online OSN spam filtering tool [6] for this task to conduct a realistic evaluation. We reuse the same parameters reported in the paper. The auxiliary spam filter is *not* an oracle. It may mistakenly report legitimate messages as spam, or miss to report spam messages.

4.1 Detection Accuracy

We test Tangram with spam window size $t = 1000$, which means when the number of spam messages that slip through the template matching module but are blocked by the auxiliary spam filter reaches 1000, the system will invoke the template generation module to infer the underlying templates of the messages. The value of parameter k is 4.

The results show that the TP rate for the most prevalent template-based spam achieves 95.7%. The system can also detect some spam messages that are not template-based, because the system treats all messages as if they were template-based, and makes best-effort detection. As expected, the TP rate of such messages is lower than the TP rate of template-based messages. The overall TP and FP rate are 76.2% and 0.12%, respectively.

True Positive Analysis. Table 5 reports a detailed breakdown of true positive rate into different spam categories. Tangram has two detection modules. Both modules perform well on the specific spam category that they are designed for. The template generation/matching module can detect template-based spam with 95.7% TP rate. The noise detection module can detect no-content spam with 73.8% TP rate. Unfortunately, the true positive rate of the other two spam categories is lower. About 80% of the false negatives (spam misclassified as legitimate) belong to the other two categories.

False Positive Analysis. Since the labeling approach we use to build the ground truth may miss to label true spam tweets (Section 2.1), We further compare the true positives against the detected tweets that are not labeled. We observe that spammers frequently attach *Retweet* marks (RT @*username*) and *Mentions* (@*username*) at the beginning of tweets, as well as noise words after the embedded URL. Hence, we remove all the noise and acquire the stem of spam tweets. Any tweet that shares the same stem with spam tweets is also regarded as spam. The comparison reveals that 15,271 (0.12%) tweets reported by Tangram are neither labeled as spam, nor sharing the same stem with spam tweets. They represent the false positives that our system incurs. The comparison approach exploits characteristics that may only apply to our specific dataset. Hence, we only use it as a post-processing step in the evaluation, rather than adopting it in the system design.

Among the false positive tweets, 42.0% of them are caused by overly general spam templates. Another 21.7% of them are popular tweets like birthday wishes for Nelson Mandela. These popular tweets are mistakenly reported as spam by

System	Template Based		Syntactical Clustering	Tangram + Syntactical
	Tangram	Judo		
Spam Category				
Template-based	95.7%	32.3%	70.1%	98.4%
Paraphrase	51.0%	52.2%	51.4%	70.1%
No-content	73.8%	41.9%	67.0%	83.1%
Other	18.4%	30.4%	43.2%	44.7%
Overall TP	76.2%	35.9%	63.3%	85.4%
FP	0.12%	5.0%	0.27%	0.33%

Table 5: The detection accuracy of Tangram and two existing systems compared in Section 4.2. The last column shows the accuracy if we combine Tangram and the syntactical approach.

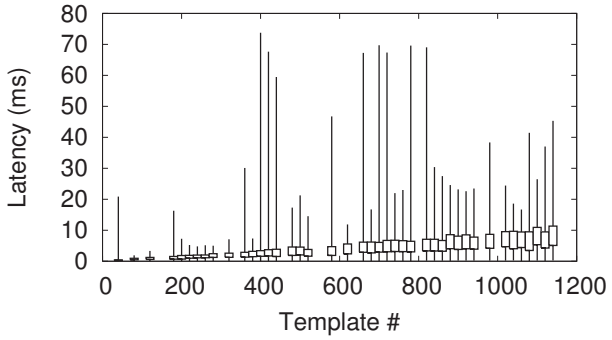


Figure 2: The box plot of template matching latency as a function of the number of generated templates.

the auxiliary filter, so templates are generated to match them.

4.2 Detection Accuracy Comparison with Existing Work

We limit the direct experimental comparison to only the approaches that examine the message content to detect spam.

Syntactical clustering + machine learning. We first compare with a recent spam detection work that adopts syntactical message clustering and supervised machine learning [6] (denoted as the syntactical clustering approach hereafter) in detail. The two systems share similar design goals. In addition, the existing approach is used as the auxiliary spam filter in our experiment. Hence, it is crucial to quantify the detection accuracy gains over directly using the existing system. We run the system using the same dataset on which we test Tangram.

The syntactical clustering approach achieves an overall detection accuracy of 63.3% TP rate and 0.27% FP rate. The true positive rate obtained by the syntactical clustering approach on our data is lower than the reported number in [6]. The reason is that our spam labeling approach labels more spam tweets as ground truth, which the syntactical clustering approach does not detect. In contrast, Tangram achieves a substantial improvement on both the TP rate (to 76.2%) and the FP rate (to 0.12%). Table 5 lists the detailed accuracy comparison. The difference between the spam detected by these two systems indicates that they can potentially complement each other. Both systems work in a similar way and it is straightforward to integrate them. For example, a message can be labeled as spam if either system blocks it. This simple integration suffers from the increased FP rate of 0.33%, but can boost the TP rate to 85.4%.

Judo. To validate that our template generation technique is more tailored to OSN spam detection, we also compare our work with a recent email spam detection system called *Judo* [22]. *Judo* detects email spam based on template generation. We have already presented the difference between the two systems analytically by elaborating the difference in the critical system assumptions, i.e., invariant substring in template and quality of training samples. We further demonstrate their difference in experimental results, shown in Table 5, column "Judo". Different from our system, *Judo* requires training set that contains *pure spam* generated by the *same* underlying template. As a result, in real-world deployment the system relies on the assumption that only a small number of templates are actively used at a given time, so that the training set is pure at least within a given small time window. We implement the template generation mechanism of *Judo* as described in [22], and test the detection accuracy using the same dataset. Even with small window size (10 spam messages), the generated templates can only achieve 35.9% TP rate. The TP rate further drops to 10.6% if the window size is increased to 20. On the other hand, the FP rate is high (5.0%). It shows that real-world OSN trace breaks the crucial assumptions of *Judo*. As a result, *Judo* achieves extremely high accuracy in email spam detection, but does not perform well for OSN spam detection. In comparison, our system achieves much higher accuracy on the same corpus.

4.3 Template Generation/Matching Speed

Template matching. The template matching latency incurred by Tangram is minimal and is not noticeable to users. Figure 2 plots the minimum, 25% quantile, 75% quantile and maximum of the template matching time as a function of the number of generated templates during our online experiment. We observe a large variance of template matching latency, because the time consumed for regular expression matching highly depends on the text being matched. Nevertheless, the largest latency in the entire dataset is less than 80ms. The overall trend is that the template matching latency, shown by the boxes representing the 25% quantile and the 75% quantile, grows slowly with the number of templates. Even with more than one thousand templates, the median template matching latency is only 8ms.

Template generation. It is crucial to throttle spam campaigns at their early stage. Hence, we measure how fast templates can be generated. The time to generate template depends on the number of buffered spam messages. In our experiment, the mean template generation time is only 2.3 seconds. Although the time needed for template generation is larger than the time for template matching, the template generation time is *not* the bottleneck of Tangram, since template generation is performed in parallel with template matching.

4.4 Sensitivity for New Campaigns

We take the five largest campaigns, one of which matches the template instantiated by spam in Table 1, and evaluate how fast Tangram reacts to newly emerged spam. We randomly select a small percentage of messages from each campaign, and use them as training samples to generate the template. We vary the percentage of training samples from 0.05% to 0.5%. The remaining messages serve as the testing set. We measure the true positive rate as the percentage

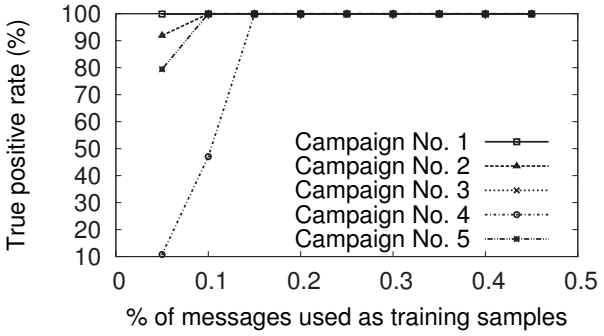


Figure 3: The TP rate when template generation is performed separately for each campaign, varying the size of training set. Most observation points reach 100% TP rate.

of the testing set that the generated template can match. Figure 3 shows the results. We observe that all campaigns achieve almost 100% coverage even with only 0.15% of messages as training samples. Three campaigns have lower coverage when only 0.05% of messages are used to generate the template, because the system has not observed all possible values of dictionary macros due to insufficient training samples. Nonetheless, the coverage quickly climbs up to almost 100% when the percentage of training samples increases. The result indicates that when new spam campaigns emerge, the system can react quickly and generate effective templates to throttle them.

5. DISCUSSIONS

How to increase the attack resilience of the auxiliary spam filter? In practice, Tangram needs an auxiliary filter with a low false positive rate. This is a reasonable requirement, since we can tune the auxiliary filter to be conservative in reporting spam. Spammers can try to evade Tangram by evading the auxiliary spam filter, so that no training samples are available for template generation. We can address this by introducing heterogeneity into the auxiliary filter. Using a combination of multiple inherently different existing spam detection systems as the auxiliary filter makes it very hard for spammers to evade *all* of them.

How to mitigate training sample poisoning? Powerful adversaries can manipulate the training samples to mislead the training of the detection system. One possible way is to inject popular legitimate content into spam, hoping that the generated template matches a large number of legitimate messages. However, our experimental results show that Tangram will not generate template for spam with seemingly legitimate content. As a second precaution, we can set a threshold and only deploy the templates that incur a false positive rate lower than the threshold.

Another popular attack is to send spam with different patterns from the training samples after the detection system finishing training. Tangram is inherently immune to such attack, because it does not have separate training and testing phase. If spammers produce spam from a non-stationary distribution, Tangram can also detect it as long as sufficient large amount of spam is produced from each distribution.

How to mitigate paraphrase spam? The major difficulty for Tangram to detect paraphrase spam is the lack of ordering among semantically meaningful segments, which is assumed by the template model. Nonetheless, semantic analysis techniques that do not consider the word ordering may be used to mitigate such spam, e.g., clustering based on cosine similarity using bag-of-words model. In addition, paraphrase generation is still an active area of research in Natural Language Processing [20]. So is paraphrase detection [5]. The detection of paraphrase spam can leverage existing paraphrase detection approaches, and is one of our future research directions.

How to mitigate spam that re-uses legitimate content? Since spammers need to use popular content to attract the audience and generate a large number of spam, the OSN administrator will observe the popular content with replaced URLs in large volume, which defeats the spammers' purpose to evade detection. Also, in practice, we can equip Tangram with multiple heterogeneous detection modules that do not rely on spam content.

6. RELATED WORK

Spam Detection. *Judo* [22] also infers the underlying template used to generate spam. However, *Judo* (as well as its adaptation to web spam [33]) assume the presence of invariant substring in the template, and requires a clean spam trace instantiating the same template as input. These requirements are hard to satisfy in the OSN environment. We also show via experiments that the *Judo* design does not yield satisfactory detection accuracy on a real-world OSN trace. In contrast, our system is designed specifically for OSN and is not limited by the above two requirements.

Researchers have proposed other approaches that fight spam using the textual content, including using the syntactical textual similarity within the same campaign [6,34] and extracting signature of embedded URLs [30]. Meanwhile, other features of spam/spammers are used to fight spam as well. Egele et al. model account profiles and use anomaly detection to identify compromised accounts in OSNs [4]. Song et al. propose to use the social graph property to detect spam tweets [24]. Yang et al. use sender profile features among others [31]. Other proposed techniques include focusing on embedded URL information like redirection chains, DNS and WHOIS information and so on [15,18,19], classifying URLs' landing pages [2,26] and using sender's reputation [3,9,25,29]. Building sender profile features takes time and it is difficult to adopt for real-time detection. Compared with our work, although some of the above approaches report higher spam coverage, they incur significantly higher false positive rate, showing that the features they use are less precise than spam templates. More importantly, very few existing works can do real-time detection and filter spam without URLs simultaneously, where as Tangram by design can achieve both.

Spam Measurement. Thomas et al. examine a large corpus of suspended Twitter accounts in [27], which provides rich knowledge on Twitter spammers that inspires our work from multiple aspects. In addition, Grier et al. and Gao et al. discovered the popularity of compromised spamming accounts in Twitter and Facebook, respectively [7,8]. Due to the different data collection method, most spamming accounts in our dataset are created by spammers. Yang et al. analyze the social network formed by spamming ac-

counts and reveal different categories of legitimate accounts that follow spamming accounts [32]. Levchendo et al. and Kanich et al. study the monetization of spam campaigns [11, 16]. In comparison, our work focuses on detecting spam, whereas the above works focus on revealing spammers' characteristics using known spam.

Signature Generation. The problem of spam template generation bears similarity with polymorphic worm signature generation [17, 21]. The worm signature generation is based on the assumption that polymorphic worm content contains invariant substrings, which is reasonable because some invariant bytes are crucial for successfully exploiting the vulnerability. However, this assumption is not solid in the context of spam detection, where spammers can express the same message using different expressions in human language. Our Twitter spam analysis supports this argument. Venkataraman et al. formalize the limits on the pattern-extraction algorithms for signature generation in the presence of powerful adversaries [28]. Due to the difference in the underlying assumptions, i.e., with and without invariant substrings, it is hard to directly apply their conclusions to our spam template generation problem.

7. CONCLUSION

We have proposed and evaluated Tangram, a template-based system for accurate and fast OSN spam detection. Our measurement study reveals that 63% of Twitter spam is likely to instantiate underlying templates. Based on the empirical findings, Tangram mainly employs template generation/matching to mitigate OSN spam. Tangram distinguishes from existing template generation work in that it can construct template in the absence of invariant substrings. Tangram detects OSN spam in real-time without a separate training phase. Experimental results show that Tangram can detect 95.7% of the most prevalent template-based spam in the collected Twitter dataset. Equipped with one necessary auxiliary spam filter, the combined system achieves an overall true positive rate of 85.4% and a false positive rate of 0.33%.

Acknowledgement

This work is supported in part by the following grants: NSF awards CCF-1029166, ACI-1144061, IIS-1343639, and CCF-1409601; DOE award DESC0007456; DARPA contract D11AP00268; Fundamental Research Funds for the Central Universities under Grant No. 2014QNA5012. We would also like to sincerely thank ACSAC 2014 chairs and reviewers for their helpful feedback.

8. REFERENCES

[1] What the trend. `http://www.whatthetrend.com/`.

[2] D. S. Anderson, C. Fleizach, S. Savage, and G. M. Voelker. Spamscatter: characterizing internet scam hosting infrastructure. In *USENIX Security*, 2007.

[3] F. Benevenuto, G. Magno, T. Rodrigues, and V. Almeida. Detecting spammers on Twitter. In *CEAS*, 2010.

[4] M. Egele, G. Stringhini, C. Kruegel, and G. Vigna. COMPA: Detecting Compromised Accounts on Social Networks. In *NDSS*, 2013.

[5] S. Fernando and M. Stevenson. A semantic similarity approach to paraphrase detection. In *CLUK*, 2008.

[6] H. Gao, Y. Chen, K. Lee, D. Palsetia, and A. Choudhary. Towards Online Spam Filtering in Social Networks. In *NDSS*, 2012.

[7] H. Gao, J. Hu, C. Wilson, Z. Li, Y. Chen, and B. Y. Zhao. Detecting and characterizing social spam campaigns. In *IMC*, 2010.

[8] C. Grier, K. Thomas, V. Paxson, and M. Zhang. @spam: the underground on 140 characters or less. In *CCS*, 2010.

[9] S. Hao, N. A. Syed, N. Feamster, A. G. Gray, and S. Krasser. Detecting spammers with snare: spatio-temporal network-level automatic reputation engine. In *USENIX Security*, 2009.

[10] T. Jiang and M. Li. On the approximation of shortest common supersequences and longest common subsequences. In *ICALP*, 1994.

[11] C. Kanich, C. Kreibich, K. Levchenko, B. Enright, G. M. Voelker, V. Paxson, and S. Savage. Spamalytics: An empirical analysis of spam marketing conversion. In *CCS*, 2008.

[12] C. Kreibich, C. Kanich, K. Levchenko, B. Enright, G. M. Voelker, V. Paxson, and S. Savage. On the spam campaign trail. In *LEET*, 2008.

[13] C. Kreibich, C. Kanich, K. Levchenko, B. Enright, G. M. Voelker, V. Paxson, and S. Savage. Spamcraft: An inside look at spam campaign orchestration. In *LEET*, 2009.

[14] J. D. Lafferty, A. McCallum, and F. C. N. Pereira. Conditional random fields: Probabilistic models for segmenting and labeling sequence data. In *ICML*, 2001.

[15] S. Lee and J. Kim. WarningBird: Detecting suspicious URLs in Twitter stream. In *NDSS*, 2012.

[16] K. Levchenko, N. Chachra, B. Enright, M. Felegyhazi, C. Grier, T. Halvorson, C. Kanich, C. Kreibich, H. Liu, D. McCoy, A. Pitsillidis, N. Weaver, V. Paxson, G. M. Voelker, and S. Savage. Click Trajectories: End-to-End Analysis of the Spam Value Chain. In *S&P*, 2011.

[17] Z. Li, M. Sanghi, Y. Chen, M.-Y. Kao, and B. Chavez. Hamsa: Fast signature generation for zero-day polymorphicworms with provable attack resilience. In *S&P*, 2006.

[18] J. Ma, L. K. Saul, S. Savage, and G. M. Voelker. Beyond blacklists: learning to detect malicious web sites from suspicious urls. In *KDD*, 2009.

[19] J. Ma, L. K. Saul, S. Savage, and G. M. Voelker. Identifying suspicious urls: an application of large-scale online learning. In *ICML*, 2009.

[20] N. Madnani and B. Dorr. Generating phrasal and sentential paraphrases: A survey of data-driven methods. *Computational Linguistics*, 36(3):341–387, 2010.

[21] J. Newsome, B. Karp, and D. Song. Polygraph: Automatically generating signatures for polymorphic worms. In *S&P*, 2005.

[22] A. Pitsillidis, K. Levchenko, C. Kreibich, C. Kanich, G. Voelker, V. Paxson, N. Weaver, and S. Savage. Botnet Judo: Fighting Spam with Itself . In *NDSS*, 2010.

[23] A. Ritter, S. Clark, O. Etzioni, et al. Named entity

recognition in tweets: an experimental study. In *EMNLP*, 2011.

[24] J. Song, S. Lee, and J. Kim. Spam filtering in twitter using sender-receiver relationship. In *RAID*, 2011.

[25] G. Stringhini, C. Kruegel, and G. Vigna. Detecting spammers on social networks. In *ACSAC*, 2010.

[26] K. Thomas, C. Grier, J. Ma, V. Paxson, and D. Song. Design and Evaluation of a Real-Time URL Spam Filtering Service. In *S&P*, 2011.

[27] K. Thomas, C. Grier, V. Paxson, and D. Song. Suspended Accounts In Retrospect: An Analysis of Twitter Spam. In *IMC*, 2011.

[28] S. Venkataraman, A. Blum, and D. Song. Limits of learning-based signature generation with adversaries. In *NDSS*, 2008.

[29] A. H. Wang. Don't follow me - spam detection in twitter. In S. K. Katsikas and P. Samarati, editors, *SECRYPT*, pages 142–151. SciTePress, 2010.

[30] Y. Xie, F. Yu, K. Achan, R. Panigrahy, G. Hulten, and I. Osipkov. Spamming botnets: signatures and characteristics. In *Proc. of SIGCOMM*, 2008.

[31] C. Yang, R. Harkreader, and G. Gu. Die free or live hard? empirical evaluation and new design for fighting evolving twitter spammers. In *RAID*, 2011.

[32] C. Yang, R. Harkreader, J. Zhang, S. Shin, and G. Gu. Analyzing spammers' social networks for fun and profit: a case study of cyber criminal ecosystem on twitter. In *WWW*, 2012.

[33] Q. Zhang, D. Y. Wang, and G. M. Voelker. Dspin: Detecting automatically spun content on the web. In *NDSS*, 2014.

[34] L. Zhuang, J. Dunagan, D. R. Simon, H. J. Wang, and J. D. Tygar. Characterizing botnets from email spam records. In *LEET*, 2008.

APPENDIX

A. TEMPLATE GENERATION COMPUTATIONAL CHALLENGE

We prove that the template generation problem is NP-hard. We use the following notions for ease of presentation:

1. $len(S)$**:** When S is a token sequence, this operation computes the number of tokens in S. When S is a set or a list, this operation recursively computes the sum of $len()$ for all the elements in S. $len(\varepsilon)$ is zero.
2. $cat(S)$**:** This operation concatenates the elements of a set S in arbitrary order and returns the concatenation.
3. $[s_1, s_2, ..., s_k]$**:** An ordered list containing k elements.

Single Campaign Template Generation (SCTG): We start with the problem to reconstruct the underlying template given a set of spam messages instantiating the *same* template.
Input: n token sequences, instantiating the *same* template.
Output: An ordered list $\mathbf{L} = [S_1, S_2, ..., S_k]$, where each S_i is a set of token sequences. S_i may contain ε (representing nothing). Every input token sequence can be represented as the concatenation of one element in each S_i. Meanwhile, $len(\mathbf{L})$ is minimized.
Hardness: SCTG is NP-hard. We prove this by reducing Shortest Common Supersequence Problem, a well-known NP-hard problem, into SCTG (Table 6).

Input: A set of n token sequences $M = \{m_1, m_2, ..., m_n\}$
Reduction: Find $\mathbf{L}$, the SCTG solution of M. $S :=$"" **Foreach** S_i in $\mathbf{L}$: $S := S + cat(S_i)$ **End Foreach** S is the shortest common supersequence of M.
Proof: S is the shortest common supersequence of M
It is trivially true that S is a common supersequence of M. Assume $\exists S'$, such that S' is a common supersequence of M and that $len(S') < len(S)$. We can construct an ordered list $\mathbf{L}'$ of $len(S')$ elements, where the i_{th} element is a set $\{$the i_{th} token in S' , $\varepsilon\}$. Apparently, every $m_i \in M$ can be represented as the concatenation of one element of each set in $\mathbf{L}'$. Because $\mathbf{L}$ is the output of the SCTG solver, we must have $len(\mathbf{L}) <= len(\mathbf{L}')$. Because $len(\mathbf{L}) = len(S)$, $len(\mathbf{L}') = len(S')$ and $len(S') < len(S)$, we have $len(\mathbf{L}) > len(\mathbf{L}')$. We thus reach a contradiction. Hence we reject the assumption and prove that S is the shortest common supersequence of M.

Table 6: The reduction from Shortest Common Supersequence Problem to SCTG.

Multiple Campaign Template Generation (MCTG): In real-world deployment scenarios, the system is expected to receive a mixture of spam instantiating multiple templates, and it is non-trivial to separate them. Accordingly, the system should be able to generate multiple templates from the input spam. Intuitively, the MCTG problem is at least as hard as the SCTG problem, since a MCTG solver is able to solve the SCTG problem as well.

B. CORRECTNESS OF MATRIX COLUMN REDUCTION

In Section 3.2, we state three conditions to merge matrix columns. *Condition i*) column j and column k have identical label; *Condition ii*) in any row at least one column is ε; and *Condition iii*) if the cell at row i, column k is not ε, all cells in row i, between column j and column k must be ε. Under the three conditions, we prove that the *supersequence property* holds after merging by contradiction. Let column j and column k satisfy the above three conditions, and we assume that after merging column k into column j, the resulting concatenation of column labels are not a supersequence of some input sequence i (row i). There is at least one token in sequence i that is not covered by the resulting concatenation of column labels. Let it be at the m_{th} column in the original matrix. m must be k because other columns still perserve after merging. Hence, row i column k must not be ε. We denote it as t. Due to *condition ii*), row i column j must be ε. Due to *condition iii*), the subsequence of row i between column j and column k is merely one token, t. Since column j and column k have identical token (*condition i*)), after merging the label of columns between j and k can cover t. Hence, the resulting column labels are still a supersequence of input sequence i. We hereby reach a contradiction, and the proof is complete.

A Taste of Tweets: Reverse Engineering Twitter Spammers

Chao Yang
SUCCESS Lab
Texas A&M University
yangchao@cse.tamu.edu

Jialong Zhang
SUCCESS Lab
Texas A&M University
jialong@cse.tamu.edu

Guofei Gu
SUCCESS Lab
Texas A&M University
guofei@cse.tamu.edu

ABSTRACT

In this paper, through reverse engineering Twitter spammers' tastes (their preferred targets to spam), we aim at providing guidelines for building more effective social honeypots, and generating new insights to defend against social spammers. Specifically, we first perform a measurement study by deploying "benchmark" social honeypots on Twitter with diverse and fine-grained social behavior patterns to trap spammers. After five months' data collection, we make a deep analysis on how Twitter spammers find their targets. Based on the analysis, we evaluate our new guidelines for building effective social honeypots by implementing "advanced" honeypots. Particularly, within the same time period, using those advanced honeypots can trap spammers around 26 times faster than using "traditional" honeypots.

In the second part of our study, we investigate new *active* collection approaches to complement the fundamentally *passive* procedure of using honeypots to slowly attract spammers. Our goal is that, given limited resources/time, instead of blindly crawling all possible (or randomly sampling) Twitter accounts at the first place (for later spammer analysis), we need a lightweight strategy to prioritize the *active* crawling/sampling of *more likely* spam accounts from the huge Twittersphere. Applying what we have learned about the tastes of spammers, we design two new, *active and guided* sampling approaches for collecting most likely spammer accounts during the crawling. According to our evaluation, our strategies could efficiently crawl/sample over 17,000 spam accounts within a short time with a considerably high "Hit Ratio", i.e., collecting 6 correct spam accounts in every 10 sampled accounts.

Keywords

Online Social Network Websites, Spam, Twitter

1. INTRODUCTION

Online Social Networks (OSNs) such as Twitter and Facebook have been utilized by social spammers to garner victims. Particularly, social spammers could actively garner their victims through creating spam accounts to initialize unsolicited social relationships or send unsolicited messages, rather than merely passively waiting for victims' visits as in the traditional email spam or web spam cases.

In fact, restricted by OSNs' anti-spam measures, many OSN spammers have evolved to launch *Targeted Social-Media Spamming* (i.e., spammers selectively choose their spamming targets by analyzing those targets' behaviors [34]). Many Twitter users have undergone the following experience: once they write some big brand names such as "Ipad" or "Best Buy" in their tweets, they may receive a slew of @mentions[1] offering "free" products or gift cards related to the brands [6, 5]. Note that before 2012, @mentions will be directly shown on the users' public timeline; since then, users will receive senders' @mentions in the users' "Notifications[2]", no matter the users follow the senders or not. Such observations indeed imply an obvious interaction between users' social behaviors and spammers' actions (as illustrated in Figure 1).

Figure 1: Illustration of interactions between users' social behaviors and spammers' actions.

The benefit for spammers to use this strategy to find targets is straightforward. Through selectively choosing targets to initialize unsolicited friend requests or send unsolicited messages, spammers could significantly decrease the risks of being detected under current OSNs' policies. (According to our observation, a Twitter account that constantly follows more than 50 accounts per day will highly possibly be suspended by Twitter within a week.) Furthermore, after knowing targets' tastes or social friend-circles, spammers could significantly increase their chances of successful spamming, either by actively pushing spam messages related to targets' tastes (e.g., on Twitter) or pretending to be in the same social friend-circle (e.g., on Facebook). In this way, social spammers could garner victims more effectively by launching customized actions based on their targets' social behavior characteristics. Thus, this is different from the scenario for traditional email spam or web spam, in which attackers usually know nothing about their targets and can merely blindly send spam.

[1] @mentions are tweets that are sent to specific users by using the tag of "@".

[2] A tab in the users' homepage shows users' social interactions with others.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.
ACSAC '14,December 08 - 12 2014, New Orleans, LA, USA
Copyright 2014 ACM 978-1-4503-3005-3/14/12 ...$15.00
http://dx.doi.org/10.1145/2664243.2664258.

However, so far we know little about basic insights of the interactions between users' behaviors and spammers' actions. Such insights can facilitate us to understand common questions such as "Why do I get spam friends? [13]", "Why do I receive spam messages? [2]" and "How do spammers find their targets?". The desire of addressing such questions, and thus *obtaining insights for defending against social spammers*, form one important motivation of this work.

In addition, among many existing research efforts in fighting against spam/spammers [23, 41, 28, 15, 38, 36, 33], social honeypot techniques are quite promising, and have been widely deployed to collect spammers [28, 29, 36]. A social honeypot is essentially a specially created fake account with the intent to capture spammers' social interactions. However, current social honeypots are designed to be either *too static* (few behaviors performed by honeypots) or *too uniform* (few variations among honeypots' behaviors). As a result, those honeypots are not used in an optimal or effective way to trap as many spammers as they can. The fundamental reason is that we lack basic insights of the strategies utilized by spammers to select spam targets. Thus, a good understanding of spammers' tastes is pressing and we seriously need *systematic guidelines for building more effective (attractive) social honeypots*.

Furthermore, although many existing studies rely on social honeypots (or even manual identification) to collect likely spam accounts (aiming at further analyzing them to generate defense insights), such strategies are still not very efficient in terms of collecting a large number of spam accounts from the huge Twittersphere. In particular, the technique of social honeypots is relatively passive and typically requires a long time to attract many spam accounts. The strategy of manually labeling spam accounts is tedious, time-consuming, and very difficult to scale. Thus, given limited resources/time, a light-weight strategy to selectively sample *more likely* spam accounts (instead of blindly collecting all or randomly sampling accounts) during the crawling from the huge Twittersphere is strongly desired.

In this paper, through reverse engineering spammers' tastes (their preferred targets to spam), we aim at providing guidelines for designing more effective social honeypots, and designing lightweight and guided strategies to actively sample more likely social spam accounts. To achieve this goal, we use Twitter as a case study due to its great popularity and publicity. Specifically, to reveal which behaviors tend to incur spammers' contact, we implement 96 "benchmark" Twitter social honeypots with 24 diverse fine-grained social behavior patterns to trap spam accounts. After launching our social honeypots for five months, we successfully garner around 600 spam accounts. Using these data, we analyze spammers' tastes (how spammers find their targets), through comparing the effectiveness of social honeypots with different behavior patterns. Based on the analysis, we design and implement 10 more effective ("advanced") honeypots to trap Twitter spammers. Within the same time period, using those advanced honeypots can trap spammers around 26 times faster than using "traditional" honeypots. To further understand spammers' tastes, we also design an algorithm to extract semantic topic terms, which may highly attract spammers' attentions. In addition, with the concern of limited time/resources, through reverse engineering spammers' strategies of selecting targets, we gain the insights to design two guided approaches to prioritize the *active* sampling of more likely spam accounts from Twittersphere, which is an effective complement to existing *passive* social honeypots.

In summary, the main contributions of our study are:

- We present a deep analysis of spammers' tastes: spammers tend to contact with accounts that tweet messages and follow accounts related to specific topics.
- We deploy "advanced" (more effective) honeypots based on our provided guidelines, which can trap spammers around 26 times faster than using "traditional" honeypots.
- We design two lightweight, guided approaches to prioritize the sampling of more likely Twitter spam accounts in the huge Twittersphere. According to our evaluation, our designed two samplers can efficiently collect over 17,000 Twitter spam accounts in a short time with a considerably high "Hit Rate" (correctly collect around 6 spam accounts in every sampled 10 accounts).

2. RELATED WORK

Detection of spam accounts. The task of detection is to answer the question: given an OSN account, how can we tell whether it is a spam account or not? Most existing OSN spam account detection solutions can be mainly classified into two categories. The first category of work, such as [41, 28, 15, 38, 36, 33, 19], utilizes machine-learning techniques to classify spam accounts according to their collected training data and selections of classification features (e.g., the ratio of an account's following number to its follower number). While successful, some recent research has reported that spammers have begun to utilize multiple tricks to evade existing detection features by pretending to be legitimate ones [41, 33]. Thus, this type of approach may fail to detect those evasive spam accounts. The second category of research detects spam accounts by examining whether the URLs or domains of the URLs in the tweets exist in the public URL blacklists or domain blacklists [23]. One limitation of such approaches is that, current URL (and domain) blacklists usually have a big lag to identify malicious links in the spam tweets [23]. Our work is not a detection solution. However, our proposed sampling strategies (Section 5) can provide a guided approach to prioritize the sampling of more likely spam accounts (instead of blindly/randomly crawling) in the huge Twittersphere, thus providing a good first-layer filter for existing detection approaches.

Utilization of honeypots. A honeypot is a decoy (e.g., a computer, data, or a network site) mainly set up to attract attackers. Traditionally, the honeypot techniques have been widely used for capturing malware and related malicious activities. Server-side honeypots are mainly implemented by emulating vulnerable services or software to trap attacks, aiming at collecting malware and malicious requests [44], understanding network and web attacks [26], building network intrusion detection systems [27], or preventing the spread of spam email [20]. Client-side honeypots are mainly used to detect compromised (web) servers [32, 1, 39, 31]. In [14], Antonatos *et al.* proposed an approach to detect instant messaging (IM) threats using IM honeypots.

In the context of OSN, social honeypots are defined as OSN accounts that appear to belong to real users, but are actually fake accounts used for attracting spammers. Due to its simplicity and low false positives, social honeypots are a great way to collect spammers for further study, e.g., understanding their characteristics and then further building effective machine-learning features to detect them. Many existing studies [36, 28, 28] use this social honeypot technique. However, an important missing component in this line of research is that, we still know little about the interactions between users' behaviors and spammers' actions, e.g., why this social honeypot can attract few/many spammers. Essentially, we need *a systematic analysis on how to build more effective social honeypots*, which is an important goal of this work. Thus, this paper bridges the gap in existing research using social honeypots.

Measurement of spam campaigns and networks. In [43], Yardi *et al.* analyzed Twitter spam accounts' social behaviors and network structures by investigating a specific spam campaign [43]. In [21], Gao *et al.* conducted a study on detecting and characterizing social spam campaigns on Facebook, based on the observation that spam accounts in the same spam campaign tend to send similar spam messages simultaneously. In [37], Thomas *et al.* analyzed tools, techniques, and support infrastructure utilized by spam accounts through retrospecting suspended accounts. In [42], Yang *et al.* presented a deep empirical analysis on spammers' social network. All these studies deepen our understanding of spam campaigns and networks, thus providing valuable insights for defenders. In this paper, we perform a deep social honeypot measurement study to understand spammers' tastes, thus help to design new guidelines for building better social honeypots and guided strategies to prioritize the sampling of more likely spam accounts. Thus, our work is a new supplement to existing work.

Recently, Ghosh *et al.* investigated link farming activities in Twitter and and found that a small number of legitimate Twitter users account for the majority of link farming activities [22]. Irani *et al.* presented a new social engineering attack to trick benign users into contacting spammers, instead of using spam accounts to initiate contact with benign accounts [25]. The results show that recommendation systems, demographics and visitor tracking could affect normal users' choices in making new friends. Our study is essentially a reverse side story: focusing on revealing *spammers'* (instead of normal users') tastes, and design more effective social honeypots and new sampling strategies to actively collect likely spam accounts.

3. PROBLEM STATEMENT

We next introduce the research scope of this work. As discussed in Section 1, our research goal is to understand the characteristics of targeted spamming in Twitter, and further to gain new defense insights against them by reverse engineering the spamming tastes of those spammers. Particularly, we use a relatively strict/conservative view (similar to existing work [19, 41] and Twitter rules [7]) to consider an account to be a spam account, if it meets one of the following criteria: (1) tend to post spam or malicious URLs in the tweets; (2) tend to post scam words in the tweets; (3) repeatedly post duplicate tweets; (4) repeatedly send "@mention" messages to other accounts with few useful content.

To achieve our research goal, we first design 96 social honeypots with diverse social behavior patterns to garner spammers (see Figure 2). Based on the functions provided by Twitter, these social behavior patterns mainly vary in terms of tweeting behaviors, following behaviors, and application usage. Particularly, the content posted by users, the famous accounts followed by users and the applications used by users may display users' tastes, incurring spammers' contact. Then, these social honeypots could trap spammers by receiving spammers' unsolicited messages and obtaining spam followers. All of social honeypots' behaviors and their trapped spammers' actions (sending unsolicited messages or building unsolicited friendships) will be saved in a local database. Next, after deeply analyzing spammers' tastes by comparing the effectiveness of honeypots with different social behavior patterns, we can provide guidelines to build effective honeypots. Finally, through reverse engineering spammers' strategies of selecting targets, we design two lightweight, guided strategies (Hashtag Sampler and Friend Sampler) to prioritize the sampling of more likely Twitter spam accounts from the huge Twittersphere. More specially, Hashtag Sampler is designed to catch spammers that target on specific accounts if they tweet specific hashtags. Friend Sampler is designed to catch spammers that target on specific famous accounts' followers.

4. REVERSE ENGINEERING SPAMMERS

In this section, we describe our methodologies of extracting and analyzing social spammers' tastes. Specifically, we design and launch multiple social honeypots with diverse fine-grained behavior patterns to garner spammers. Next, through analyzing intrinsic properties of the interactions between users' social behaviors and spammers' actions, we could better understand the following questions: Who do spammers spam? How do spammers find their victims? Through these analysis, we further provide guidelines of building more attractive social honeypots.

4.1 Collecting Spammers' Tastes

4.1.1 Design of Social Honeypots

To analyze the interactions between users' behaviors and spammers' actions, we endow social honeypots with diverse fine-grained social behavior patterns to show diverse users' tastes. Since a Twitter account mainly has three categories of social behaviors (Tweet, Follow and install applications), we design social honeypots based on the variations of these three categories: Tweet Behavior (Tweet), Follow Behavior (Follow) and Application Usage (App) (see Table 1).

Tweet Behavior. The content tweeted by users (and tweeting frequency) may directly expose users' interests. Particularly, the keywords/topics posted by users may reveal their tastes, which could be utilized by spammers to find targets. In fact, many users' real experience has shown that different tweet keywords may behave very differently in terms of incurring spammers [5]. Accordingly, we divide our social honeypots' tweet behaviors into three sub-categories: Tweet Frequency, Tweet Keywords and Tweet Topics.(To reduce possible effects to other users, our social honeypots will not post any links and "@ mentions".)

Tweet Frequency: refers to how often to post one tweet. We divide tweeting frequency into the following three patterns: 1 tweet per hour, 2 tweets per day, and 1 tweet per day. For each pattern, we use 5 honeypot accounts to post tweets according to the specific tweet frequency. Those tweets are randomly selected from the dataset containing around half million Twitter accounts and 14 million tweets, which was collected from Apr. 2010 to Aug. 2010 by using Twitter Stream APIs.

Tweet Keywords: refer to special words or terms in the tweets, which may represent specific semantic topics. We divide tweet keywords into several patterns: popular trending topics, arbitrary hashtags, current affairs, bait words, and no hashtag tweets.

- "Popular trending topics" refer to those hot Twitter trending topics [8], which are widely used by Twitter users to express their opinions or experience on specific topics or events. For each day, we collect top (the most widely used) 10 trending topics. Then, we use 5 honeypot accounts to post these 10 trending topics. Each of them will post 2 trending topics.

- "Arbitrary hashtags" refer to those tweet terms with the tag of "#". The tweets containing the same hashtag will be grouped together by Twitter and can be searched out by users from Twitter Search [11]. These hashtags are also randomly selected from the pre-collected dataset. For each day, we use 5 honeypot accounts to post 10 tweets with hashtags. Each of them posts 2 tweets, which are randomly selected from our collected dataset.

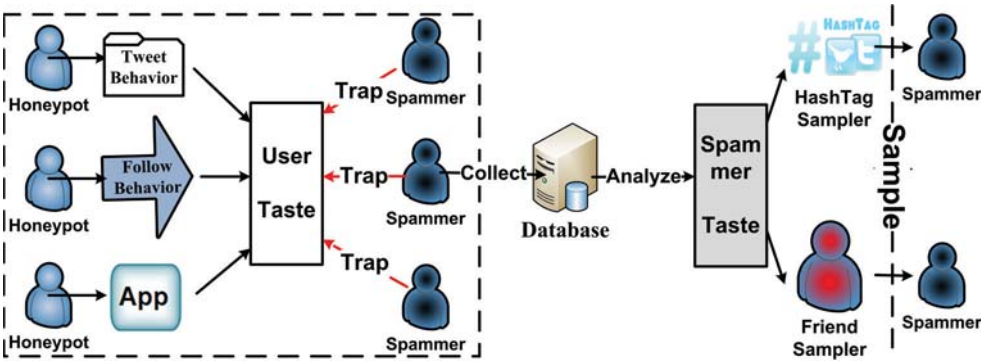


Figure 2: Illustration of the analysis flow.

Table 1: 96 "benchmark" social honeypots with 24 fine-grained social behavior patterns

Index	Category	Sub-Category	Pattern	Index	Category	Sub-Category	Pattern
1-5	Tweet	Frequency	Once per day	6-10	Tweet	Frequency	Twice per day
11-15	Tweet	Frequency	Once per hour	16-20	Tweet	Keywords	Trending Topics
21-25	Tweet	Keywords	Arbitrary Hashtags	26-30	Tweet	Keywords	Current Affairs
31-35	Tweet	Keywords	Bait Words	36-40	Tweet	Keywords	No Hashtags
41-45	Tweet	Topic (Twice per day)	Entertainment	46-50	Tweet	Topic (Twice per day)	Expertise
51-55	Tweet	Topic (Twice per day)	Sports	56-60	Tweet	Topic (Twice per day)	Economics
61-62	Tweet	Topic (Once per hour)	Entertainment	63-64	Tweet	Topic (Once per hour)	Expertise
65-66	Tweet	Topic (Once per hour)	Sports	67-68	Tweet	Topic (Once per hour)	Economics
69-70	Follow	Two accounts per day	Entertainment	71-72	Follow	Two accounts per day	Expertise
73-74	Follow	Two accounts per day	Sports	75-76	Follow	Two accounts per day	Economics
77-81	App	NA	Twitpic	82-86	App	NA	Instagr
87-91	App	NA	Twiends	92-96	Default	NA	NA

- "Current affairs" refer to important social events happened each day. To extract such events, we crawl the top 10 headlines from CNN.com, and use 5 honeypots to post those headlines (each posts two headlines).

- "Bait words" refer to those keywords that are mainly used by spammers in their scam webpages or messages to trap victims (e.g., "giftcard"). We use a list of 200 bait words, mainly obtained through feeding queries such as "scam word lists" to Google.com. Then, we also use 5 honeypots to post 2 messages containing bait words per account per day.

- "No hashtags" refer to tweets without any hashtags, which are used to compare with other social patterns. We use 5 honeypots to post 10 tweets, randomly selected from the database without any hashtags on each day.

Tweet Topics: refer to specific semantic topics in the tweets, which may explicitly reveal users' tastes. Particularly, we focus on the following four topics: *Entertainment, Expertise, Sports, and Economics*. Entertainment contains topics related to TV media, music, books and arts; Expertise contains topics related to IT technology, Science, Fashion and Household; Sports are related to golf, NBA, NCAA, NFL and NHL; Economics are related to business, finance and charity. To use our honeypots to tweet those semantic topics, we first collect tweets related with those topics by searching topic terms (e.g., "NBA") on Twitter. Then, for each topic, we use 5 honeypots to post one tweet per day. To compare, we use 2 honeypots to post 1 tweet per hour.

Follow Behavior. Besides the content posted by users, users' followings (especially those famous people or companies' official accounts) may also reveal their tastes. For example, if an account follows "Lady Gaga", the owner of the account may like music or live concert. Thus, this kind of following tastes might be utilized by spammers.

To extract spammers' such tastes, we use our honeypots to follow "verified accounts", whose tweets are related to four major topics mentioned above: Entertainment, Expertise, Sports and Economics. Specifically, for each topic, we manually collect 400 verified accounts from Twitter. These verified accounts are typically owned by famous people or organizations with high reputation, such as sports stars and official business accounts. Thus, through following those verified accounts, our honeypots explicitly show their interests to those topics. Particularly, for each topic, we use 2 social honeypots to follow 2 verified accounts per day. (To reduce possible effects, each account will follow 30 verified accounts at most.)

Application Usage. Users who install specific Twitter applications (e.g., multimedia sharing tools and online games) may also reveal their specific tastes and thus become spammers' targets. In our test, we choose three very popular Twitter social applications: Twitpic [10], Instagram [4] and Twiends [9]. (Twitpic and Instagram are popular photo and video sharing tools, and Twiends is an online Twitter friend-making tool.) For each application, we use five honeypots to install and use it.

Default. As a comparison, we also use five honeypots with default account registration configuration, which neither post any tweets nor follow any accounts.

In summary, as seen in Table 1, we design 96 honeypots with 24 diverse fine-grained behavior patterns to garner spammers. Since the aim of designing these social honeypots is to understand which specific social behaviors tend to incur spammers rather than to trap

more spammers, we refer these 96 honeypots as "benchmark" honeypots.

Figure 3: The implementation of social honeypots.

4.1.2 Implementation of Social Honeypots

To implement those "benchmark" honeypots, we develop a real-time Twitter application, named social honeypot app (SHP), which has three major operations: write, follow, and read. As illustrated in Figure 3, write operation implements tweet-behaviors by posting tweets on honeypots' timelines; follow operation implements follow-behaviors by following other accounts; read operation collects spam accounts and spam tweets trapped by our social honeypots, by reading honeypots' followers and "@mentions".

More specifically, the application obtains each honeypot's access token to automatically make the corresponding operations (write, follow and read) on the account to perform its designed social behaviors according to the protocol of OAuth 2.0. All the auxiliary data such as our collected tweet dataset, popular trending topics and bait words are loaded into the app to implement corresponding operations. Finally, the app will record each honeypot's social behaviors, and its received "@mentions" and followers into a local database everyday. In this way, we can collect the interactions between honeypots' behavior patterns and their trapped spammers' actions.

To make our honeypots more likely to be real accounts (i.e., to decrease the chance of being identified as honeypots by spammers), we register our honeypot accounts by using real names (e.g., *Tracy Thompson*) and valid email addresses. Also, we initialize the friendships among those honeypots. We admit that smart spammers might still recognize our social honeypots by deeply analyzing those honeypots' behaviors, because these honeypots are designed with a set of scheduled tasks. However, many normal accounts (e.g., official company accounts) are also customized to post particular messages/notifications in a scheduled way. Thus, it is not that trivial for spammers to distinguish honeypots from normal accounts. Also, this limitation is common for all this line of studies, which rely on deploying automated honeypots.

4.2 Analyzing Spammers' Tastes

We next show our data collection results and analysis of trapped spammers' tastes.

4.2.1 Data Collection Result

We implemented those 96 "benchmark" honeypots and ran them for five months. We collected $1,077$ unique accounts that at least follow one of our social honeypots, and 440 unique accounts that at least post one "@mention" to one honeypot. In total, there are $1,512$ unique accounts.

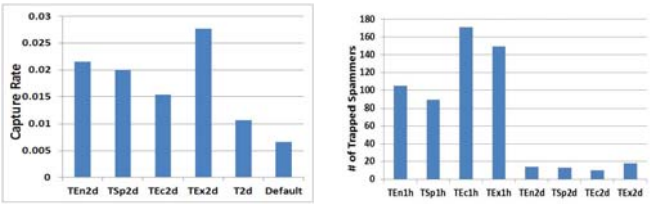
To extract spammers' tastes, we need to identify spam accounts from those $1,512$ accounts. We first found out 303 accounts that have been suspended by Twitter due to their violations to the Twitter Rule. Furthermore, following the definition of our target spam accounts' described in Section 3, we identified additional 278 spam accounts by manually examining accounts timeline and checking those accounts' posted URLs. In total, we obtain 578 spam accounts.

Note that the number of spam accounts trapped by our "benchmark" honeypots seems a little smaller than other earlier social honeypot studies (e.g., [28]). We believe this is due to the following reasons. First, those studies were conducted in early days when Twitter has relatively loose policies to identify/mitigate spammers. However, since 2009, Twitter has taken significant anti-spam efforts to actively filter/mitigate a lot of spam accounts [12]. In addition, in this work, to guarantee the correctness to analyze spammers' interests, we use a relatively strict way to consider an account to be spam. While there could be a few spam accounts missed in our data collection with this relatively strict spammer identification strategy, we believe that our major findings/conclusions in this paper will still hold.

4.2.2 Analysis of Spammers' Tastes

We next provide our analysis of spammers' tastes based on 578 trapped spam accounts. To better measure the effectiveness of social honeypots with different behavior patterns, we define a metric named *Capture Rate (CR)*, which is the average number of spam accounts trapped by a honeypot per day. Thus, a higher value of *CR* of honeypots with a specific pattern implies this pattern is more effective to trap spammers. Then, our analysis and measurement results are presented in the question-answer format.

Q1: If an account posts more tweets related with specific semantic topics, does it tend to attract more spammers' attention? Empirical Answer: Yes. One possible way to find targets for spammers is to analyze targets' tweet content, which may show targets' interests on specific topics. Then, through actively pushing spam related to those topics to those targets, attackers may achieve a better chance of success. As seen in Figure 4(a), posting messages related with specific topics (Entertainment – `TEn`, Sports – `TSp`, Economics – `TEc`, Expertise – `TEx`), will incur more spammers' contact than posting arbitrary messages even with the same tweeting frequency (twice per day). More specifically, `TEx2d`'s CR (the highest for tweeting topic twice per day) is around 3 times as that of `T2d`, and `TEc2d`'s CR (the lowest) is around 1.5 times as that of `T2d`. In addition, when we increase the frequency from twice per day to once per hour (e.g., from `TEn2d` to `TEn1h`), honeypots can trap more spammers (See Figure 4(b)). And the average values of CR for these four topics can be increased around 22.35 times (from 0.021 to 0.494). Thus, we can find that honeypots can trap more spammers through frequently tweeting messages related with certain semantic topics.

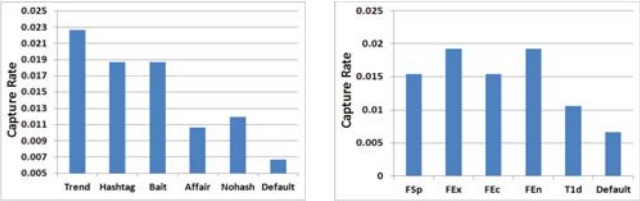


(a) Tweet Topics (b) # of Trapped Spammers

Figure 4: The effectiveness of tweet topics.

Q2: Do accounts that tweet more special terms (e.g., "Trending topics") tend to attract more spammers' contact? Empirical Answer: Yes. As seen in Figure 5(a), the values of CR for tweeting trending topics (`Trend`), arbitrary hashtags (`Hashtag`),

and bait words (Bait), are all higher than that of Nohash (arbitrary tweets without hashtags) and Default. This observation indicates that posting special key terms may also incur spammers' contact. This is because these key terms usually represent semantic topic meanings, which could be utilized by spammers to find targets (similar to tweeting topics). In addition, we can find that Trend is more effective than Hashtag. This might because trending topics are more timely and popular than arbitrary hashtags.



(a) Tweet Keywords (b) Follow Behavior

Figure 5: The effectiveness of tweet keywords and follow behavior.

Q3: Do users' following behaviors tend to expose them to spammers? Empirical Answer: Yes. As seen in Figure 5(b), similar to tweet topics, the values of CR for following verified accounts related with the topics of Entertainment (FEn), Sports (FSp), Economics (FEc) and Expertise (FEx) are all higher than that of T1d and Default. This observation implies that the behavior of following those famous ("verified") accounts could be utilized by spammers to find their targets. As a case study, we find one spam account, which mainly posts spam about TV media (the URLs in the tweets have been identified as suspicious by the URL shortening service), shares 19 followings (most of them are related the topic of art or TV media) with one honeypot.

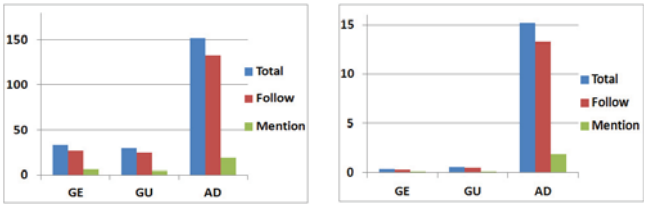
Q4: Do accounts with the usage of social apps tend to be contacted by more spammers? Empirical Answer: No. According to our data, the capture rates of honeypots with the usage of Instagram, Twitpic and Twiends are 0.008, 0.008, and 0.009, respectively. These values are lower than that of most of other social patterns and similar to *Default*. Thus, using social apps do not help much in terms of attracting spammers. This might either because spammers have not use this strategy to find targets or the selections of applications used by our honeypots are not representative.

4.2.3 Guidelines for Designing Effective Honeypots

According to the above analysis, we could summarize the following guidelines for designing more effective social honeypots to trap Twitter spammers: **(1) post tweets related with specific topics; (2) post tweets containing special keywords such as Trending topics; (3) follow famous accounts related with specific areas.**

To evaluate the effectiveness of those guidelines, we denote *96 "benchmark" honeypots* as GE, and 51 honeypots of them[3] that *meet at least one guideline* as GU. We find that GU's capture rate (0.083) is over two times as that of GE (0.040). This observation indicates that GU (that meet guidelines) is more effective to attract spammers than GE.

To further evaluate the effectiveness of our guidelines, we deploy another 10 "advanced" honeypots (AD) with more guided social behaviors for a week right after finishing the 5-month running of "benchmark" honeypots. Specifically, in each day, each of them will behave the following social patterns[4]: (1) post one topic tweet per hour related with each of those four topics mentioned in Section 4.1.1; (2) post one tweet containing one trending topic per hour; (3) post one tweet containing one arbitrary hashtag per hour; (4) post one tweet containing one bait word per hour; (5) Follow 5 experts related with each of four topics per day. Then, as seen in Figure 6, we compare the performance of AD with GE and GU by collecting data in the same week. We can find that AD is much more effective than GE and GU in trapping more spammers. Particularly, AD's capture rate (2.17) is 25.5 times as that of GU (0.085), and 45.2 times as that of GE (0.048). Although this comparison result may contain some bias due to a relatively short period time of data collection, such a huge difference could still validate the effectiveness of our guidelines for designing better social honeypots.



(a) # of Spammers (b) per Honeypot

Figure 6: The effectiveness of advanced honeypots.

4.2.4 Extracting Spammers' Interested Topic Terms.

To better understand spammers' tastes, it is meaningful to extract specific semantic terms, which tend to be used by spammers to find targets. Although this could be partially achieved by analyzing hashtags in the tweets, a more generic and automated approach (not only limited to hashtags) is still needed. Due to the page limit, we mainly introduce the intuition of our way of extracting spammers' interesting topic terms.

Figure 7: One real case study of potential victims.

As shown in Figure 7, we find that many spammers send illicit "@mentions" not only to our honeypots but also to other users (e.g., "gladynotglady" in this example), which are denoted as "potential victims" in this work. We denote each pair of potential victims and honeypots in one illicit "@mention" as a "victim relationship". Thus, we believe that there should be some common social behavior patterns between our honeypots and those potential victims in each victim relationship, which essentially incur spammers' contacts. An intuitive method is to extract the common terms used by both our honeypots and those victims, which may represent spammers' tastes. However, this approach will extract many widely-used common words, which are not representative for spammers' tastes. Although we use a big stop-word list to filter some common words, it could not help much, because many words tweeted by users are not even spelled completely or in a standard form. Instead, we first use the Latent Dirichlet Allocation (LDA)[5] [17] algorithm to

[3] 51 accounts are 16-25, 31-35, 41-76 as labeled in Table 1.

[4] To prevent spammers identifying our accounts as honeypots based on the temporal patterns, some random delays are inserted before posting each tweet.

[5] LDA is a generative probabilistic topic modeling (clustering) al-

extract topic terms, which are better to represent semantic topics, from the tweets posted by our honeypots and potential victims. Then, we output those topic terms that are highly/frequently shared by the pairs of victims and honeypots. Accordingly, such topic terms may highly attract spammers' contact.

Specifically, our honeypots receive 449 "@mention" messages from spammers and form 5,716 victim relationships with 275 unique potential victims. Then, we extract $1,500$ and 600 topic terms for each honeypot and potential victim, respectively. We finally output the top 500 as semantic topic terms. (Due to the page limit, we skip to show those topic terms.) Furthermore, through extracting semantic topic terms, we could examine whether there is tweet similarities between spammers and their targets. Particularly, we extract semantic topic terms for 278 (manually) identified spam accounts by using LDA. Then, we extract all pairs of honeypots and spam accounts, if the spam accounts either follow or "@mention" the honeypot. Then, we find 81.69% of 360 pairs of two accounts share at least one semantic term. This observation indicates a relatively strong semantic similarity between spammers and their targets.

4.2.5 Ethical Considerations

The technology of deploying social honeypots on real OSNs may raise ethical considerations: whether such social honeypots will impact the normal operation of OSNs. In our work, both "benchmark" and "advanced" honeypots are designed to neither send any @mentions nor post any URLs/spam in the tweets. Also, those accounts only follow a relatively small number (several hundreds) of "verified" accounts. Thus, we believe our designed honeypots will have very limited impacts to other normal users. In addition, the technique of social honeypots has been commonly used to capture spammers [36, 28] or to understand the security vulnerability [18] on OSNs. Furthermore, as advocated in a recent study on the ethics of security vulnerability [30], such studies served as social functions are neither unethical nor illegal.

5. PRIORITIZING THE SAMPLING OF LIKELY SPAMMERS

In this section, we design two guided approaches to actively sample more likely Twitter spam accounts from Twittersphere, based on the observation that many spammers find their targets based on targets' social behaviors.

5.1 Motivation

The collection of spam accounts is usually the first step to analyze spammers' behaviors and to further generate defense insights. However, given the limited time/resources (especially for academic researchers), it is not trivial to collect a large-scale of spam accounts in the huge Twittersphere. Existing studies mainly rely on the following three strategies to collect (likely) spam accounts: implementing social honeypots[28, 29, 36], collecting suspended accounts [37, 25], and manual identification [28, 38, 42]. However, all these three strategies have certain limitations. The honeypot approach is a *passive* one, requiring time (and luck) to wait for spammers' contacts. Collecting suspended accounts requires to develop a robust crawler and takes a considerable long time (typically several months) to crawl Twitter and to wait for collected accounts to be suspended by Twitter. Manual identification could achieve a high accuracy, which requires tedious human work and is not scalable.

Motivated by the limitations of existing strategies to collect (likely) spam accounts, we design two lightweight, guided strategies (called *Samplers* in this paper) to prioritize the active sampling of more likely spam accounts: *Hashtag Sampler and Friend Sampler*. These two samplers are designed to be able to *efficiently* collect/sample a considerable number of targeted social-media spam accounts (a specific type of spam accounts that attract victims by socially interacting with others) in a short time period with a relatively high hit rate without any training process. We clearly admit that these two samplers are not designed to detect all types of spam accounts (e.g., those spam accounts that just tweet malicious content without interacting with any other users.).

5.2 Hashtag Sampler

Basic intuition: Spammers tend to follow those accounts that post spammers' interested keywords (hashtags). According to this intuition, an account might be suspicious if it follows many accounts that share some spammers' interested hashtags in their tweets. At the high level, Hashtag sampler is designed to preferentially sample likely spam accounts through checking *common followers* of multiple accounts that share/post *similar, multiple* hashtags as spammers do.

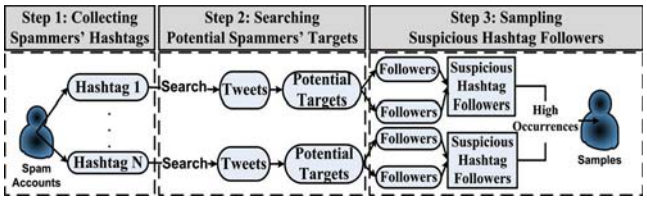


Figure 8: Illustration of Hashtag Sampler.

Detailed Strategy: As illustrated in Figure 8, Hashtag Sampler has three steps to sample likely spam accounts from Twitter: (1) collecting spammers' hashtags; (2) searching potential spammers' targets; (3) sampling suspicious hashtag followers.

Particularly, in Step 1, Hashtag Sampler collects keywords/hashtags that spammers are potentially interested in (i.e., hashtags in spam accounts' tweets) through identifying hashtags ("#") from tweets posted by our trapped spam accounts. In Step 2, for each hashtag, Hashtag Sampler searches the recent M tweets[6] that contain hashtags, through exactly querying the hashtag from Twitter Search. Then, we consider an account to be a potential spammers' *target*, if they send tweets containing that particular hashtag. Accordingly, by extracting the senders of those tweets, Hashtag Sampler searches out all potential targets. In Step 3, for each potential spammers' target, we obtain its followers by using Twitter API. After extracting all targets' followers, we denote those followers with high occurrences as *suspicious hashtag followers*. These hashtag followers essentially follow many other accounts that post that spammers' hashtag. Finally, Hashtag Sampler outputs those accounts as spam accounts, if they are sampled as suspicious hashtag followers with the usage of multiple different hashtags, i.e., they are considered as suspicious hashtag followers with a high occurrence by using different hashtags.

5.3 Friend Sampler

gorithm, which could cluster terms in the large volumes of unlabeled text into several semantic topics by identifying the latent topics words in the text.

[6]For each query term, Twitter limits to return 1,500 tweets as maximal. Thus, in our experiment, we set $M = 1,500$.

Basic Intuition: Spammers tend to select famous accounts' followers as their targets. In fact, those Twitter accounts (especially famous accounts) followed by a user could also reveal this user's taste, which could be utilized by spammers to find their potential spamming targets. According to this intuition, Friend Sampler is designed to preferentially sample likely spam accounts through checking those accounts that excessively follow multiple famous accounts' followers, i.e., examining *common followers of the followers* of some famous accounts.

Detailed Strategy: As illustrated in Figure 9, Friend Sampler first randomly selects M verified (famous) accounts from those 400 verified accounts used in Section 4. For each account, Friend Sampler collects its N followers (if available), which could be considered as spammers' potential targets. Then, we examine extracted followers of those potential targets, and save them in a dataset with their numbers of occurrences (e.g., if an account follows two potential targets, its number of occurrences is 2), denoted as suspicious account set. Finally, Friend Sampler outputs N_{fd} accounts in the suspicious account set as spammers with the top numbers of occurrences.

Figure 9: Illustration of Friends Sampler.

6. EVALUATION OF SAMPLERS

In this section, we mainly describe our evaluation methodologies and evaluation results for two samplers (Hashtag Sampler and Friend Sampler).

6.1 Ground Truth and Evaluation Metrics

Ground Truth: To evaluate the effectiveness of two samplers, we require some ground truth. However, as a common challenge for all OSN data analysis work, it is difficult to obtain perfect ground truth for a large-scale dataset. It is straightforward that an account can be considered as spam if it is suspended by Twitter. However, only considering suspend accounts as spam accounts will miss many other spam accounts, which have not been identified/suspended by Twitter. Thus, for the rest *unsuspended* accounts output by our samplers, we rely on a state-of-the-art machine-learning classifier to further examine whether they are spam accounts. This classifier is implemented based on `Random Forest` and uses the same feature set designed in [28]. Then, the classifier is trained by using 2,000 suspended accounts and 20,000 normal accounts (none of them post malicious URLs). The accuracy and false positive rate of this classifier is 99.2% and 0.97% respectively based on the training datasets through *10-fold cross validation* tests.

Note that the usage of the machine-learning technique here is to *estimate* the ground truth rather than to detect spam accounts. Also, we acknowledge that any machine learning classifier may not be absolutely accurate. However in our evaluation, we are mainly interested in getting the *estimation* of the accuracy, instead of absolute values. Furthermore, such a strategy is a common practice for similar studies on accuracy estimation of large scale unlabeled datasets [40].

Evaluation metrics: To measure the effectiveness of sampling strategies with the goal of collecting more likely spam accounts, two metrics are typically considered. The number of collected spam accounts, denoted in our work as "Hit Count (N_{hit})"; and the ratio of Hit Count to the total number of sampled accounts (N_{sample}), denoted as "Hit Ratio (H_r)". Thus, a higher value of Hit Count and Hit Ratio indicates that we can catch more spam accounts and more accurately, respectively. Motivated by the limitations of traditional ways of collecting spam accounts as described in Section 5, our two samplers are designed as lightweight, guided strategies to efficiently and effectively prioritize the sampling of more likely spam accounts instead of (otherwise) crawling/analyzing all accounts in the huge Twittersphere. Our two samplers are *not* designed to find/uncover all types of spam accounts, and they are *not* considered as spammer detectors. Accordingly, we use those two evaluation metrics (N_{hit}, H_r) instead of false positives/negatives in our evaluation. Particularly, many existing studies [45, 42, 24] similarly use these two metrics to measure the effectiveness by outputting the number of hits in a top list. With such notions, if we denote the number of suspended accounts as N_{sus} and the number of spam accounts output by the machine-learning classifier as N_{mal}, we could calculate Hit Count and Hit Ratio as follows[7]: $N_{hit} = N_{sus} + N_{mal}$; $H_r = N_{hit}/N_{sample}$.

6.2 Implementation

To implement Hashtag Sampler, we use 3,246 unique hashtags/keywords posted by 278 identified spammers. For each hashtag, Hashtag Sampler outputs $SF = 500$ (if available) suspicious hashtag followers. By using each spam account's hashtags, Hashtag Sampler samples $M = 500$ suspicious hashtag followers (if available) with the top occurrences as spammers. To implement Hashtag Sampler, we randomly select $M = 40$ verified (famous) accounts (10 accounts for each of four topics). For each verified account, we examine its $N = 5,000$ followers, which are retrieved by sending one "get-follower" request to Twitter. Then, for each follower, Friend Sampler continues to examine its followers, and samples $N_{fd} = 1,000$ top ranked accounts as spam accounts. Using these implementation parameters, we run our two samplers for four days to sample more likely spam accounts. After one month, we further examine whether those sampled accounts are suspended by Twitter.

6.3 Effectiveness of Hashtag Sampler and Friend Sampler

As seen in Table 2, Hashtag Sampler outputs 8,983 unique accounts to be likely spam accounts. Among them, 262 accounts have been suspended, and 4,665 others are output as spam accounts by the classifier. Thus, the hit count is 4,927 and the hit ratio is 0.5489, which implies that Hashtag Sampler could correctly collect one spam account by sampling less than two accounts.

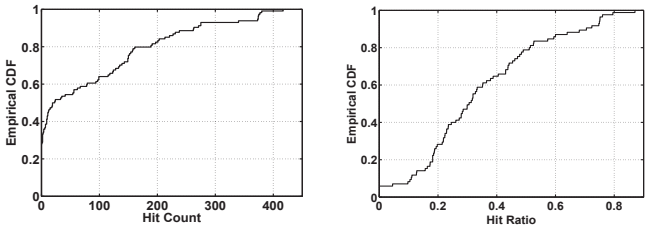
Table 2: The effectiveness of Hashtag Sampler.

Item	N_{sus}	N_{mal}	N_{hit}	N_{sample}	H_r
Value	262	4,665	4,927	8,983	0.5489

Also, we further examine hit count and hit ratio by using each spammer's hashtags. As seen in Figure 10(a), over 40% spam accounts' hashtags can be used to collect over 100 spam accounts by sampling 500 accounts. This observation shows that Hashtag Sampler can effectively collect spam accounts by focusing on spammers' tastes. Also, we find that around 30% spammers' hashtags can not be used to correctly collect spam accounts. The reason is mainly because Twitter Search does not index every tweet due

[7]Since we could not obtain ground truth for those protected and nonexistent accounts output by our samplers, we do not count such accounts in N_{sample}.

to its resource constraints [3]. According to our observation, we could crawl very few (or even no) tweets by using those spammers' hashtags.



(a) Hit Count (b) Hit Ratio

Figure 10: Collection results of Hashtag Sampler by using individual spammers' hashtags.

As seen in Figure 10(b), Hashtag Sampler could obtain reasonable hit ratios by using around 60% spammers' hashtags, which are higher than 0.3 (sampling 3 accounts will correctly collect 1 spam account).

We next show the evaluation results of the Friend Sampler. As seen in Table 3, Friend Sampler outputs 21,686 unique accounts as spammers. Among these accounts, 4,000 have been suspended, and 9,781 others are output as spam accounts by the classifier. Thus, the hit count is 13,781 and hit ratio is 0.6355.

Table 3: The effectiveness of Friend Sampler

Item	N_{sus}	N_{mal}	N_{hit}	N_{sample}	H_r
Value	4,000	9,781	13,781	21,686	0.6355

6.4 Diversity and Complementarity

Next, we analyze the diversity and complementarity of these two samplers. Essentially, we examine the number of spam accounts correctly sampled only by one algorithm, which can not be found by using the other one. If this number of each algorithm is high, it implies that these two samplers are very complementary. Thus they could be combined together to find more spam accounts. Specifically, to measure the diversity, we design a metric, named "Exclusive Ratio (E_r)", which is the ratio of the number of spam accounts that are exclusively sampled by one sampler (not sampled by the other one) to the total number of spam accounts sampled by this sampler.

As seen in Table 4, we can find that both two samplers can obtain relatively high exclusive ratios (over 77%). The ratio for Friend Sampler is even higher than 90%.

Table 4: Exclusive ratios between two samplers.

Algorithm	Hashtag Sampler	Friend Sampler
Exclusive Ratio	77.69%	90.57%

This observation shows that these two samplers are indeed complementary. And thus, they can be used together to collect more (likely) spam accounts.

As shown in Table 5, according to our collected dataset, the combination usage of these two algorithms could correctly collect 18,185 spam accounts. Among them, 4,249 accounts have been suspended by Twitter, and 13,936 other accounts are classified as spam accounts by the classifier. Thus, the hit ratio of combining these two algorithms is 0.6219. Compared with the dataset used for the purpose of building an effective machine learning classifier [16], which contains 355 manually identified spam accounts from 8,207 randomly crawled accounts (i.e., a hit ratio of only 0.04), this value of hit ratio is considerably high in terms of effectively crawling likely spam accounts in the huge Twittersphere.

Table 5: Result of combining two algorithms.

Item	N_{sus}	N_{mal}	N_{hit}	N_{sample}	H_r
Value	4,249	13,936	18,185	29,239	0.6219

7. LIMITATIONS AND FUTURE WORK

We acknowledge our manually identified spam accounts may contain some bias, and the machine-learning classifier we use to estimate the accuracy may not be absolutely accurate. However, it is challenging to obtain a perfect ground truth, and our strategies have been widely used in this line of studies [35, 19, 28, 40]. In addition, even though some values may vary according to different datasets, we believe that our major findings and insights are likely still valid independent of the datasets. It is possible that our advanced honeypots may also attract a few benign accounts' contacts. However, this highly depends on the goal of honeypots – trapping more spam accounts, or obtaining spam accounts only, for which we believe the former is more important. According to our data collection results, our advanced honeypots could trap *significantly more* spam accounts.

We note that our samplers are not designed to collect/cover all (types of) spammers in Twittersphere. In addition, we note that the number of collected spam accounts by our samplers is restricted by the number of inputs, e.g., hashtags and famous accounts. Our result is also limited by the Twitter Search API: one request could only obtain the recent 1,500 search results, and not even to mention not all tweets are indexed by Twitter Search. Thus, if our samplers are implemented by Twitter (without many restrictions), they could find more spam accounts.

8. CONCLUSION

In this paper, we perform a deep measurement study on how some Twitter spammers choose their spamming targets, through building social honeypots with diverse social behavioral patterns. Based on the analysis of spammers' tastes, we provide principled guidelines for building more effective (attractive) social honeypots. Furthermore, we design two new lightweight and effective samplers to guide the active sampling of more likely Twitter spam accounts, which we believe is a great complement to the passive social honeypot approach.

9. ACKNOWLEDGMENTS

This material is based upon work supported in part by the National Science Foundation (NSF) under Grant No. CNS-1218929. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of NSF.

References

[1] Capture HPC. `https://projects.honeynet.org/capture-hpc`.

[2] Get rid of DM spam on Twitter. `http://seanmalarkey.com/rid-dm-inbox-spam-auto-dms-mafia-invites-twitter`.

[3] I'm Missing from Search. http://support.twitter.com/groups/32-something-s-not-working/topics/118-search/articles/66018-my-tweets-or-hashtags/-are-missing-from-search-known-issue.

[4] Instagr – Photo sharing for your iphone. http://instagr.am/.

[5] Seven Things I Learned to Bait Twitter Spammers. http://www.experiencetheblog.com/2011_07_01_archive.html.

[6] So Much Twitter Spam. http://blog.sysomos.com/2011/04/07/so-much-twitter-spam/.

[7] The Twitter Rules. http://help.twitter.com/entries/18311-the-twitter-rules.

[8] Trending topics. http://support.twitter.com/entries/101125-about-trending-topics.

[9] Twiends. http://twiends.com/.

[10] Twitpic. http://twitpic.com/.

[11] Twitter Search. https://twitter.com/#!/search-home.

[12] Twitter spam reduction. http://blog.twitter.com/2010/03/state-of-twitter-spam.html.

[13] Why do I get spam followers? http://www.quora.com/Why-do-I-get-so-many-spam-followers-on-Twitter.

[14] S. Antonatos, I. Polakis, T. Petsas, and E. Markatos. A Systematic Characterization of IM Threats Using Honeypots. In *Proceedings of Proceedings of the Network and Distributed System Security Symposium (NDSS'09)*.

[15] F. Benevenuto, G. Magno, T. Rodrigues, and V. Almeida. Detecting Spammers on Twitter. In *Collaboration, Electronic messaging, Anti-Abuse and Spam Confference (CEAS)*, 2010.

[16] F. Benevenuto, T. Rodrigues, V. Almeida, J. Almeida, and M. Goncalvess. Detecting Spammers and Content Promoters in Online Video Social Networks. In *ACM SIGIR Conference (SIGIR'09)*, 2009.

[17] D. Blei, A. Ng, and M. Jordan. Latent Dirichlet Allocation., 2003.

[18] Y. Boshmaf, I. Muslukhov, K. Beznosov, and M. Ripeanu. The Socialbot Network: When Bots Socialize for Fame and Money. In *Proceedings of 2011 Annual Computer Security Applications Conference (ACSAC)*, 2011.

[19] Z. Chu, S. Gianvecchio, H. Wang, and S. Jajodia. Who is Tweeting on Twitter: Human, Bot, or Cyborg? In *Annual Computer Security Applications Conference (ACSAC'10)*, 2010.

[20] G. Dunlap, S. King, S. Cinar, M. Basrai, and P. Chen. Revirt: Enabling intrusion analysis through virtual-machine logging and replay. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI'02)*, 2002.

[21] H. Gao, J. Hu, C. Wilson, Z. Li, Y. Chen, and B. Zhao. Detecting and Characterizing Social Spam Campaigns. In *Proc. of ACM SIGCOMM IMC*, 2010.

[22] S. Ghosh, B. Viswanath, F. Kooti, N.K. Sharma, G. Korlam, F. Benevenuto, N. Ganguly, and K.P. Gummadi. Understanding and combating link farming in the twitter social network. In *Proceeding of WWW*, 2012.

[23] C. Grier, K. Thomas, V. Paxson, and M. Zhangy. @spam: The Underground on 140 Characters or Less. In *ACM Conference on Computer and Communications Security (CCS'10)*.

[24] L. Invernizzi, P. Comparetti, S. Benvenuti, C. Kruegel, M. Cova, and G. Vigna. EVILSEED: A Guided Approach to Finding Malicious Web Pages. In *IEEE Symposium on Security and Privacy (Oakland)*, 2012.

[25] D. Irani, M. Balduzzi, D. Balzarotti, E. Kirda, and C. Pu. Reverse Social Engineering Attacks in Online Social Networks. In *Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*, 2011.

[26] J. John, F. Yu, Y. Xie, A. Krishnamurthy, and M. Abadi. Heat-seeking Honeypots: Design and Experience. In *Proceedings of the 20st WWW*, 2011.

[27] C. Kreibich and J. Crowcroft. Honeycomb: Creating intrusion detection signatures using honeypots. In *Proceedings of the 2nd Workshop on Hot Topics in Networks (HotNets'03)*, 2003.

[28] K. Lee, J. Caverlee, and S. Webb. Uncovering Social Spammers: Social Honeypots + Machine Learning. In *ACM SIGIR Conference (SIGIR)*, 2010.

[29] K. Lee, B. Eoff, and J. Caverlee. Seven Months with the Devils: A Long-Term Study of Content Polluters on Twitter. In *5th International AAAI Conference on Weblogs and Social Media (ICWSM).*, 2012.

[30] A. Matwyshyn, A. Keromytis, A. Cui, and S. Stolfo. Ethics in security vulnerability research. In *IEEE Security and Privacy, Vol. 8, No. 2*, 2010.

[31] A. Moshchuk, T. Bragin, S. Gribble, and H. Levy. A crawler-based study of spyware on the web. In *Proceedings of the 13th Annual Symposium on Network and Distributed System Security (NDSS'06)*, 2006.

[32] N. Provos. A virtual honeypot framework. In *Proceedings of the 13th USENIX Security Symposium, (USENIX'04)*, 2004.

[33] J. Song, S. Lee, and J. Kim. Spam Filtering in Twitter using Sender-Receiver Relationship. In *Proceedings of the 14th RAID*, 2011.

[34] V. Sridharan, V. Shankar, and M. Gupta. Twitter Games: How Successful Spammers Pick Targets. In *Proceedings of 28th ACSAC*, 2012.

[35] M. Steyvers and T. Griffiths. Probabilistic topic models. In *Handbook of Latent Semantic Analysis*, 2007.

[36] G. Stringhini, S. Barbara, C. Kruegel, and G. Vigna. Detecting Spammers On Social Networks. In *Annual Computer Security Applications Conference (ACSAC'10)*.

[37] K. Thomas, C. Grier, V. Paxson, and D. Song. Suspended Accounts in Retrospect:An Analysis of Twitter Spam. In *Internet Measurement Conference (IMC'11)*.

[38] A. Wang. Don't follow me: spam detecting in Twitter. In *Int'l Conferene on Security and Cryptography (SECRYPT)*, 2010.

[39] Yi. Wang, D. Beck, X. Jiang, R. Roussev, C. Verbowski, S. Chen, and S. King. Automated Web Patrol with Strider HoneyMonkeys. In *Proceedings of the 13th Annual Symposium on Network and Distributed System Security (NDSS'06)*, 2006.

[40] C. Whittaker, B. Ryner, and M. Nazif. Large-Scale Automatic Classification of Phishing Pages. In *Proceeding of 17th NDSS*, 2010.

[41] C. Yang, R. Harkreader, and G. Gu. Die Free or Live Hard? Empirical Evaluation and New Design for Fighting Evolving Twitter Spammers. In *Proceedings of the 14th RAID*, 2011.

[42] C. Yang, R. Harkreader, J. Zhang, S. Shin, and G. Gu. Analyzing Spammers' Social Networks For Fun and Profit – A Case Study of Cyber Criminal Ecosystem on Twitter. In *Proceedings of the 21st International World Wide Web Conference (WWW)*, 2012.

[43] S. Yardi, D. Romero, G. Schoenebeck, and D. Boyd. Detecting spam in a Twitter network. In *First Monday, 15(1)*, 2010.

[44] V. Yegneswaran, J. Giffin, P. Barford, and S. Jha. An architecture for generating semantics-awarex signatures. In *Proceedings of the 14th USENIX Security Symposium (USENIX'05)*, 2005.

[45] J. Zhang, P. Porras, and J. Ullrich. Highly predictive blacklisting. In *The 17th USENIX Security Symposium (USENIX Security)*, 2008.

New Models of Cache Architectures Characterizing Information Leakage from Cache Side Channels

Tianwei Zhang
Princeton University
tianweiz@princeton.edu

Ruby B. Lee
Princeton University
rblee@princeton.edu

ABSTRACT

Side-channel attacks try to breach confidentiality and retrieve critical secrets through the side channels. Cache memories are a potential source of information leakage through side-channel attacks, many of which have been proposed. Meanwhile, different cache architectures have also been proposed to defend against these attacks. However, there are currently no means for comparing and evaluating the effectiveness of different defense solutions against these attacks.

In this paper, we propose a novel method to evaluate a system's vulnerability to side-channel attacks. We establish side-channel leakage models based on the *non-interference* property. Then we define how the security aspects of a cache architecture can be modeled as a finite-state machine (FSM) with state transitions that cause interference. We use *mutual information* to quantitatively reveal potential side-channel leakage of the architectures, and allow comparison of these architectures for their relative vulnerabilities to side-channel attacks. We use real attacks to validate our results.

1. INTRODUCTION

Confidentiality is a major concern in information security. Strong encryption is often used for confidentiality protection. Many attacks have been designed to break these cryptographically protected systems. Among them, side-channel attacks exploit the physical characteristics of the system to derive the crypto keys. These attacks are serious security threats for several reasons. First, side-channel attacks target the vulnerabilities of the systems instead of the cryptographic algorithms. So the same attack strategies can often be applied to different ciphers. Second, side-channel attacks can be successfully performed in a short period of time (e.g., average 3 minutes in [1]), which may not cause any noticeable impact on the system. Third, the attackers do not need high privileges to launch an attack. All the operations are within their authorized privilege level. Fourth, side-channels exist widely in different systems. Power [2, 3], electromagnetic radiation [4, 5], timing [6], etc., can all be exploited by the attacker to infer the inaccessible critical information. Fifth, the observable side-channel information is due to the inherent physical features of the system, so it is very difficult to eliminate these side-channels.

A popular target of side channel attacks is the hardware cache. Caches are one of the most important features for improving performance in modern processors. Their use in the memory hierarchy significantly reduces the memory access time. However, the different access times due to fast cache hits versus slow cache misses can be exploited in cache side channel attacks to leak information. [1, 7–9]. It is typically unacceptable to eliminate this side channel by disabling the cache, because of the severe performance degradation that would result. Recent work has also shown the possibility of cross-VM side-channel attacks in cloud computing [10], making side-channel attacks a real and serious threat.

To defend against cache side-channel attacks, a variety of software defenses [1,8,11,12] have been proposed. These defenses tend to degrade performance severely [11], and are not applicable to arbitrary programs [12]. Furthermore, software defenses are not completely secure, since software has no control over hardware caches. For example, a software solution [1,8,11] may add instructions to load an entire security-sensitive table into the cache, before each access to this table, in the software-implemented cipher. However, this does not prevent the cipher from being context-switched out and having the cache lines containing table entries evicted by another program (possibly the attacker). Also, hardware cache controllers have the freedom to invalidate, evict or replace any cache lines – beyond the control of any software.

Hardware defenses have also been proposed to mitigate information leakage through cache side-channels, with the goal of providing more security and higher performance than software defenses. The idea is to rethink cache architectures to thwart certain types of side-channel attacks without impacting the essential performance provided by caches. Different secure cache architectures have been proposed [13–15]. The performance of these architectures can be tested by performance benchmarks, but their security effectiveness have only been analyzed qualitatively. Currently, there are no reliable methods for comparing the effectiveness of different secure cache architecture approaches. In this paper, we propose a new way of modeling caches to answer two questions: (1) *Do these secure caches really defend against cache side-channel attacks?* and (2) *What are the relative vulnerabilities of different cache architectures to different side-channel attacks?*

Our approach is to model the security aspects of different cache architectures as finite state machines, and accurately

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.
ACSAC '14, December 08 - 12 2014, New Orleans, LA, USA
Copyright 2014 ACM 978-1-4503-3005-3/14/12 ...$15.00
http://dx.doi.org/10.1145/2664243.2664273.

model the *interference* property which can show preservation or breaches of confidentiality under side-channel attacks. Our cache state machines include different *subjects* and *states*. Transitions between different *states* can leak confidential information between different *subjects*, when the *non-interference* property is violated. In addition, we propose some quantitative representations of the potential vulnerabilities of the modeled cache architectures to the side channel attacks. To the best of our knowledge, our method of modeling the security aspects of cache architectures and interferences is novel, as are our quantitative measures of side-channel vulnerability which allow comparison of different cache defense strategies and analysis of the root causes of side-channel leakage.

The key contributions of this paper are:

- Showing how to apply the principle of *non-interference* to the modeling of side-channel information leakage;
- Exploiting *mutual information* for measuring the side-channel information leakage, and identifying three conditions for achieving *non-interference* properties;
- A new way of building finite-state machines for modeling the security aspects of cache architectures, and using a model-checking tool for quantitative characterization of the systems' side-channel vulnerabilities;
- Verifying our cache security models and their relative side-channel vulnerabilities with real attacks.

The rest of the paper is organized as follows: Section 2 gives the background of cache side-channel attacks and different secure cache defenses. Section 3 describes our side-channel leakage models and quantification for measuring information leakage. In Section 4, we show how to build new security models of cache architectures. In Section 5, we integrate the side-channel leakage model into the cache security models and quantify the caches' potential information leakage. We consider other side-channel attacks in Section 6. Section 7 uses experimental data with real attacks to validate our security models of the different cache architectures. Section 8 discusses related work. Section 9 gives our conclusions and suggestions for future work.

2. BACKGROUND

2.1 Cache Side-channel Attacks

Cache side-channel attacks are particularly dangerous as they are simple software attacks on essential hardware components. The attacker does not need to share the address space with the victim and access its memory. However, he shares the hardware caches with the victim, which provides the attacker a side channel to observe the victim's secret information: a victim's programs executing on the system may have different cache behaviors (hits or misses) when memory accesses are made. These behaviors have different timing characteristics. The attacker tries to capture these characteristics, and deduce the victim's memory accesses that might help him recover the key and break the ciphers.

A large number of cache side-channel attacks have been proposed during the past few years [1, 7–9, 16–18]. The root cause of all the existing attacks is due to *interference*: either external interference between the attacker's program and the victim's program, or internal interference inside the victim's own program [13]. Combined with the cache behaviors the attackers want to observe (cache misses or hits), we have four cache side-channel attack categories [19] shown in Table 1 and described below.

Table 1: Cache side-channel attack categories

	External Interference	Internal Interference
Cache Misses	I. Access-based attacks e.g., Percival's attack	II. Timing-based attacks e.g., Bernstein's attack
Cache Hits	III. Access-based attacks e.g., Shared library	IV. Timing-based attacks e.g., Bonneau's attack

Type I: Attacks based on Cache Misses due to External Interference. In this class of attacks, the attacker and the victim run their processes on the same processor, and they share the same data cache. So the victim's process may evict the cache lines holding the attacker's data, which will cause the attacker future cache misses and give the attacker the chance to infer the victim's cache accesses. Some access-based cache attacks belong to this class, and a typical one is Percival's attack [7].

Type II: Attacks based on Cache Misses due to Internal Interference. In this class, the attacker does not run programs simultaneously with the victim. Instead, he only measures the total execution time of the victim, e.g., for encryption of one plaintext block. A longer execution time indicates there may be more cache misses from the victim's own execution; this can give the attacker information about the victim's memory accesses. Some timing-based cache attacks belong to this class, such as Bernstein's attack [8].

Type III: Attacks based on Cache Hits due to External Interference. In this class, the attacker and the victim share some memory space (e.g, a shared cryptography library). First, the attacker evicts all, or some, shared memory blocks out of the cache. After a certain time interval of the victim's execution, the attacker reads the shared memory blocks and measures the access time. A short time means the attacker has a cache hit, indicating that this cache line has been accessed by the victim during that interval and re-fetched into the cache by the victim. Then the attacker can infer the memory addresses the victim has accessed. The access-based attack in [1] belongs to this class.

Type IV: Attacks based on Cache Hits due to Internal Interference. Similar to type II attacks, the attacker still only needs to measure the total execution time of the victim. But he only cares about cache hits inside the victim's code. If the attacker measures a shorter execution time, it may be due to more cache hits during the victim's execution. So the attacker may be able to infer information about the encryption keys through the "cache collision" (i.e., cache hits) of memory accesses. Some timing-based attacks belong to this class, such as Bonneau's attack [9].

2.2 Cache Defenses and Architectures

Different cache defenses and secure cache architectures have been proposed to protect against cache side-channel attacks. Basically these designs follow one of two strategies: *Partitioning* or *Randomization* [20]:

2.2.1 Partitioning.

Caches can be exploited as side-channels in the external interference attacks because the attacker and the victim can share the caches, and thus interfere with each other's cache

usage. So one straightforward approach to prevent information leakage is to prevent the cache sharing by dividing the cache into different zones for different processes. We have the following cache designs using this idea:

Static-Partitioning (SP) cache: This cache is statically divided into two parts either by ways (like columns) or by sets (like rows). In set-associative caches partitioned by ways, each way is reserved for either the victim or the attacker program. The cache can also be partitioned by sets, where each set is reserved for either the victim or the attacker program. Due to the elimination of cache line sharing, SP caches can effectively prevent external interference, but at the well-known cost of degrading the computer's performance because of the static cache partitions.

Partition-Locked (PL) cache: PL cache [13] uses a finer-grained dynamic cache partitioning policy. In PL cache, each memory line has a protection bit to represent if it needs to be locked in the cache. Once the protected line (e.g., the victim's critical data) is locked in the cache, it can not be replaced by an unprotected line (e.g., the attacker's data). Instead, the attacker's data will be directly sent between the processor and the memory, without filling the cache. This replacement policy will thwart the attacker's plot to spy on the victim's cache accesses to security-critical data lines (e.g., those containing AES tables). This leverages, for cache security, the Lock-bit already provided by some caches to improve cache performance for frequently accessed data. The proper use of a PL cache is to preload the sensitive cache lines (e.g., AES table) before encryption begins.

2.2.2 Randomization.

In this approach, side-channel information is randomized, thus no accurate information is leaked out from caches. There are at least two ways to realize randomization: adding random noise to the attacker's observations and randomizing the mappings from memory addresses to cache sets.

Random-Eviction (RE) cache: a RE cache periodically selects a random cache line to evict. This can add random noise into the attacker's observations so he cannot tell if an observed cache miss is due to the cache line replacement or the system's random eviction policy. This will increase the attacker's difficulty in recovering secret information like a cipher key.

Random-Permutation (RP) cache: RP cache [13] uses random memory-to-cache mappings to defend against side-channel attacks. There is a *permutation table* for each process. This enables a dynamic mapping from memory addresses to hardware-remapped cache sets. When one process **A** wants to insert a new line D into the cache, it checks **A**'s *permutation table* and finds the corresponding cache line R in set S. If this R belongs to another process **B**, instead of evicting R, thus revealing information to outsiders, a random line R' in a random set S' is selected, evicted and replaced by D. At the same time, the sets S and S' in **A**'s *permutation table* are swapped, and the lines in these two sets belonging to **A** are invalidated. Since process **A**'s memory-to-cache mappings are dynamic, random and unknown to process **B**, process **B** cannot tell which memory addresses process **A** actually accessed. This is different from conventional caches, which have static and fixed memory-to-cache mappings, rather than dynamic and randomized mappings.

NewCache: NewCache [14, 21] randomizes the memory-to-cache mappings by introducing the concept of a *Logical Direct-Mapped Cache (LDM)*, which does not physically exist. The mapping from memory addresses to the LDM cache is direct-mapped, with the benefits of simplicity and speed. The mapping from the LDM cache to the physical cache is fully-associative and is realized using *Line Number Registers* (LNreg's). This dynamic and random mapping enhances the security against information leakage, as each memory line can be mapped to any physical cache line with equal probability; and the cache access pattern changes with each execution of the same program. Furthermore, the performance is enhanced since the LDM cache can be much larger than the physical cache, by merely adding extra index bits to each LNreg [14]. For the replacement policy, if the incoming line D cannot find any line in the physical cache with the same index (called an index miss), it will randomly choose a line R to replace. If the incoming line can find a line R in the physical cache with the same index, but the tag of the line (i.e. the rest of the memory address minus the index bits) is different (called a tag miss), then D may replace R [21]. The advantage of NewCache is achieving the security benefits of dynamic randomized mapping of memory lines to cache lines, with the same (or even better) performance as conventional caches [14].

3. SIDE-CHANNEL LEAKAGE MODELING

We now build a model for side-channel leakage. First we consider a general system which can be modeled as a finite state machine. The state machine consists of a set of *subjects* and *states*. Each *subject* may provide some *actions* to the machine, causing it to transition from one *state* to another, and generate some *observations*. In order to study the information flow between different *subjects* in this machine, we use the concept of *non-interference* [22], as defined below:

Definition: *Given a state machine $\mathcal{M}$, and its subjects $\mathcal{S}$ and $\mathcal{S}'$, we say $\mathcal{S}$ does not interfere with (or is non-interfering with) $\mathcal{S}'$, if the actions of $\mathcal{S}$ on $\mathcal{M}$ do not affect the observations of $\mathcal{S}'$.*

Then we have the following principle to judge the information flow in the state machine:

Principle: *Given a state machine $\mathcal{M}$, and its subjects $\mathcal{S}$ and $\mathcal{S}'$, if $\mathcal{S}$ is non-interfering with $\mathcal{S}'$, then there is no information flow from $\mathcal{S}$ to $\mathcal{S}'$.*

We now consider the side-channel information leakage. We treat a side channel $\mathcal{C}$ as a state machine. It is a connection between two *subjects*: a victim who performs some *actions* as inputs $\mathcal{I}$ to one side of the channel, and an attacker who retrieves certain *observations* as outputs $\mathcal{O}$ from the other side of the channel. The input $\mathcal{I}$ can change $\mathcal{C}$'s *state*, and affect the output $\mathcal{O}$. So the measurement of side-channel information leakage is equivalent to the evaluation of the *non-interfering* property between the channel's input and output: if $\mathcal{I}$ is non-interfering with $\mathcal{O}$, then there is no side-channel information leakage through $\mathcal{C}$.

Quantification: Side-channel leakage is a statistical process, so we use *mutual information* [23] to quantify the *non-interference* property between the channel's input and output. We denote $P_{\mathcal{I}}(i)$ as the probability that the side channel $\mathcal{C}$ is fed with the input i, $P_{\mathcal{O}}(o)$ as the probability that $\mathcal{C}$ produces the output o, $P_{\mathcal{I},\mathcal{O}}(i,o)$ as the joint probability that $\mathcal{C}$ produces the output o with the input i, $P_{\mathcal{O}|\mathcal{I}}(o|i)$ as the conditional probability that $\mathcal{C}$ produces the output o given the input i, and $P_{\mathcal{I}|\mathcal{O}}(i|o)$ as the conditional probability that the input of $\mathcal{C}$ is i given the output is o. Then the

mutual information between $\mathcal{I}$ and $\mathcal{O}$ is defined below:

$$I(\mathcal{I};\mathcal{O}) = \sum_{i\in\mathcal{I}}\sum_{o\in\mathcal{O}} P_{\mathcal{I},\mathcal{O}}(i,o)\ log\left(\frac{P_{\mathcal{I},\mathcal{O}}(i,o)}{P_{\mathcal{I}}(i)P_{\mathcal{O}}(o)}\right) \quad (1a)$$

$$= \sum_{i\in\mathcal{I}}\sum_{o\in\mathcal{O}} P_{\mathcal{O}}(o)\ P_{\mathcal{I}|\mathcal{O}}(i|o)\ log\left(\frac{P_{\mathcal{I}|\mathcal{O}}(i|o)}{P_{\mathcal{I}}(i)}\right) \quad (1b)$$

$I(\mathcal{I};\mathcal{O})$ measures how much information about $\mathcal{I}$ is leaked to $\mathcal{O}$ through the side channel. If $I(\mathcal{I};\mathcal{O})$ is close to zero, then $\mathcal{I}$ is non-interfering with $\mathcal{O}$, and there is little side-channel information flow from $\mathcal{I}$ to $\mathcal{O}$.

Equation 1(b) is the product of three terms. To reduce the mutual information to zero, we can make any of the three terms equal zero. So we have three conditions that can realize the *non-interference* property:

C1 (Output Elimination) *The channel $\mathcal{C}$ does not produce any output. Then the input does not interfere with the output.* We have the following expression:

$$\forall o \in \mathcal{O},\ \ P_{\mathcal{O}}(o) \approx 0 \quad (2)$$

C2 (Noise Domination) *For any output o, if its generation is due to the channel's inherent noise instead of the input, then the output is not affected by the input:*

$$\forall o \in \mathcal{O}, i \in \mathcal{I},\ \ P_{\mathcal{I}|\mathcal{O}}(i|o) \approx 0 \quad (3)$$

C3 (Input Ambiguity) *For any output o, the input of the channel $\mathcal{C}$ can be any i with the same probability. Then the input does not interfere with the output.* That is:

$$\forall i \in \mathcal{I}, o \in \mathcal{O},\ \ P_{\mathcal{I}|\mathcal{O}}(i|o) \approx const$$

Plugging this into the probability equation gives

$$P_{\mathcal{I}}(i) = \sum_{o'\in\mathcal{O}} P_{\mathcal{I},\mathcal{O}}(i,o') = \sum_{o'\in\mathcal{O}} P_{\mathcal{I}|\mathcal{O}}(i|o')P_{\mathcal{O}}(o')$$
$$\approx P_{\mathcal{I}|\mathcal{O}}(i|o)\sum_{o'\in\mathcal{O}} P_{\mathcal{O}}(o') = P_{\mathcal{I}|\mathcal{O}}(i|o)$$

This gives the desired expression of Input Ambiguity:

$$\forall o \in \mathcal{O}, i \in \mathcal{I},\ \ log\left(\frac{P_{\mathcal{I}|\mathcal{O}}(i|o)}{P_{\mathcal{I}}(i)}\right) \approx 0 \quad (4)$$

These three conditions are the key ideas for designing countermeasures to mitigate side-channel leakage. In Section 5, we show how to use these conditions to evaluate the side-channel leakage from different cache architectures.

4. SECURITY MODELING OF HARDWARE CACHE ARCHITECTURES

In this section, we show how to build finite state machines to model the security aspects of cache architectures.

4.1 Cache State Machine

We first consider the state machine of a single cache line shared by the attacker and the victim, which is shown in Figure 1. Each cache line can be in one of three states: **A** (occupied by the attacker), **V** (occupied by the victim) or **INV** (invalid - does not have any valid contents). There are five events that cause state transitions: *V_miss*, the victim has a cache miss for a memory line that maps into this cache line; *A_miss*, the attacker has a cache miss for data that maps into this cache line; and similarly, *V_hit* and *A_hit*, which indicate a cache hit for this cache line by the victim or the attacker, respectively; and *Invalidate*, the cache clears out the data to invalidate this line.

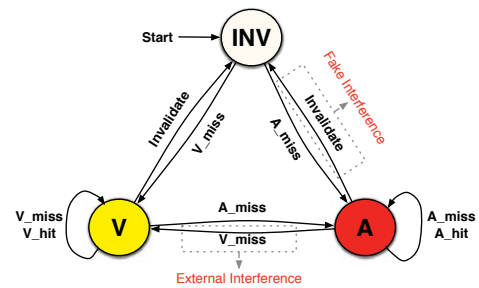


Figure 1: State machine for a single cache line

Now we consider how to model the whole cache as the combination of all the cache lines, and the dynamic operations within the cache state machine. Table 2 shows our model structure and transition rules for the cache states. It has four columns. For each **Event**, the cache will transition from the **Current State** to the **Next State**, and output the **Information Flow Log**. Assume m is the number of ways for set-associativity (columns), and n is the number of cache sets (rows). Then:

Current/Next State: *State Matrix* $S_{p,q} = \{\mathbf{A}, \mathbf{V}, \mathbf{INV}\}$ is the state of the cache line for set p and way q. *Replacement Matrix* $l_{p,q} = \{0, ..., m-1\}$ is used to model the LRU (Least Recently Used) replacement policy ordering of set p and way q for replacement, when a new line has to be brought into the cache set. If $l_{p,q} = 0$, the line in set p and way q has the highest priority to be replaced. If $l_{p,q} = m-1$, then this line has the lowest priority to be replaced in the set.

Event: We define the *Event Vector* as $T_p = \{A_miss, A_hit, V_miss, V_hit, Invalidate\}$, which is the cache action on set p that causes a state transition from $S_{p,q}$ to $S'_{p,q}$, following the rules in Figure 1.

We will discuss the **Information Flow Log** in detail in Section 5. For now, it suffices to say that it is a new output generated for the cache state machine that counts state transitions that violate the *non-interference* property. For example, external interference occurs when an attacker's cache line is replaced with a victim's cache line.

4.2 Secure Cache Modeling

The state machine of each secure cache architecture described in Section 2.2 can be built based on Figure 1 and Table 2. We now show the differences in the cache models based on the caches' unique features. Details can be found in our technical report [24].

Conventional Cache: Caches start out empty, which means all cache lines are in the **INV** state at the beginning of each experiment, transitioning to either **V** or **A** state on a *V_miss* or *A_miss* event, respectively. Subsequently, they transition between these 2 states. Conventional caches do not invalidate cache lines on load or store instructions. Hence, the *Invalidate* event only occurs on a special "Invalidate addresses" instruction – if this exists in the Instruction Set.

Table 2: Cache structure and state transition

Current State		Event	Information Flow Log	Next State	
State Matrix	*Replacement Matrix*	*Event Vector*	*Interference Matrix*	*State Matrix*	*Replacement Matrix*
$\begin{bmatrix} S_{0,0} & \cdots & S_{0,m-1} \\ \vdots & \ddots & \vdots \\ S_{n-1,0} & \cdots & S_{n-1,m-1} \end{bmatrix}$	$\begin{bmatrix} l_{0,0} & \cdots & l_{0,m-1} \\ \vdots & \ddots & \vdots \\ l_{n-1,0} & \cdots & l_{n-1,m-1} \end{bmatrix}$	$\begin{bmatrix} T_0 \\ \vdots \\ T_{n-1} \end{bmatrix}$	$\begin{bmatrix} I_0 \to O_0 & \cdots & I_{n-1} \to O_0 & I_{-1} \to O_0 \\ \vdots & \ddots & \vdots & \vdots \\ I_0 \to O_{n-1} & \cdots & I_{n-1} \to O_{n-1} & I_{-1} \to O_{n-1} \end{bmatrix}$	$\begin{bmatrix} S'_{0,0} & \cdots & S'_{0,m-1} \\ \vdots & \ddots & \vdots \\ S'_{n-1,0} & \cdots & S'_{n-1,m-1} \end{bmatrix}$	$\begin{bmatrix} l'_{0,0} & \cdots & l'_{0,m-1} \\ \vdots & \ddots & \vdots \\ l'_{n-1,0} & \cdots & l'_{n-1,m-1} \end{bmatrix}$

Static-Partitioning (SP) Cache: For SP cache, the difference with the conventional cache is that each cache line can only have two states (**INV** and **V**, or **INV** and **A**). So transitions of $\mathbf{A} \to \mathbf{V}$ or $\mathbf{V} \to \mathbf{A}$ can never happen.

Partition-Locked (PL) Cache: We consider two uses of PL cache: (1)*PL cache without preload* of the security-critical data before the victim's program begins: the cache is initially empty. Both the victim and attacker can fill the cache with its data. However, once the victim's critical cache lines are locked in the cache, they can not be replaced by the attacker. So we have the transition of $\mathbf{A} \to \mathbf{V}$, but $\mathbf{V} \to \mathbf{A}$ is forbidden. When the cache is in the state of **V** and encounters the event of *A_miss*, the attacker's data will be sent to the CPU directly, and the cache stays in **V**. (2)*PL cache with preload* of the security-critical data: the victim initially occupies the cache line and locks it in the cache. So neither transition of $\mathbf{A} \to \mathbf{V}$, nor $\mathbf{V} \to \mathbf{A}$ can happen. The *A_miss* cannot change the state of the PL cache and *A_hit* can never happen for these security-critical lines.

Random-Eviction (RE) Cache: Compared with a conventional cache, the RE cache state machine has two more transitions due to the introduction of random noise: $\mathbf{A} \to \mathbf{INV}$ (attacker's line is randomly chosen to be evicted) and $\mathbf{V} \to \mathbf{INV}$ (victim's line is randomly chosen to be evicted).

Random-Permutation (RP) Cache: An event for RP cache may involve mutiple cache lines when swapping the cache sets. When Line l in set s is in state **V** and encounters an *A_miss*, it still stays in state **V**. Instead, a random Line l' in a random set s' is selected and replaced by the incoming attacker's line, thus jumping to state **A** from whatever state it was in before. All lines of set s and s' in state **A** will be evicted out of the cache and go to state **INV**. In the meantime, the mappings of set s and s' will be swapped in the attacker's *permutation table*. A similar procedure happens when Line l in state **A** encounters a *V_miss*. Line l'' in set s'' is randomly selected and replaced, and all the lines of set s and s'' in state **V** go to **INV** state when swapping sets s and s'' in the victim's *permutation table*.

NewCache: We consider different cache events on a cache line l. When there is a cache hit for the victim (*V_index_hit* & *V_tag_hit*) or the attacker (*A_index_hit* & *A_tag_hit*), the cache will stay in state **V** or **A**, respectively. When there is an index hit but a tag miss for the victim (*V_index_hit* & *V_tag_miss*) or the attacker (*A_index_hit* & *A_tag_miss*) for line l, according to NewCache's replacement policy [14, 21], line l will be directly replaced by the incoming line, jumping from state **V** to **V** or from state **A** to **A**. When there is an index miss for the victim (*V_index_miss*) or the attacker (*A_index_miss*), a random cache line l' is selected to be replaced, and line l' will jump to state **V** or **A** respectively, from whatever state it was in previously.

5. LEAKAGE MEASUREMENT

To evaluate cache systems' side-channel vulnerabilities, we now integrate the leakage model (Section 3) into the cache state machine model (Section 4). We define the *Interference Probability* to measure the information leakage through cache side channels. As an example, we use our method to evaluate Type I attacks (observing cache misses due to external interference) in Table 1.

5.1 Side-channel Leakage Interpretation

As we stated in Section 3, the cache is shared by the victim and the attacker, and is treated as a potential side channel. The victim's *actions* are the inputs to the side channel, and the attacker's *observations* are the outputs from the side channel. To evaluate the side-channel leakage, we need to study the interference between the victim's *actions* and the attacker's *observations*. That is, how the victim's *actions* can affect the attacker's *observations*.

We define I_p as the victim's actions on cache set p ($0 \leq p < n$), and O_q as the attacker's observations of cache set q ($0 \leq q < n$). During the transitions between different cache states, we define a novel **Information Flow Log** to record the side channel's inputs and outputs, as shown in Table 2.

Information Flow Log: This log is used to track the root causes of interference. It defines a structure called **Interference Matrix**. Inside this matrix, $I_p \to O_q = \{1, 0\}$ depicts if this cache state transition happens with the victim's action on cache set p, which will lead to the attacker's later observation on cache set q. An extra input for noise, I_{-1}, and transition $I_{-1} \to O_q = \{1, 0\}$ depicts if the inherent noise from the cache channel will lead to the attacker's later observation on cache set q.

The mutual information (Equation 1) and 3 *non-interference* conditions (Equations 2, 3 and 4) can be used to evaluate the side-channel leakage. To do so, we need to go over all the possible cache states and count the number of each kind of interference, $N(I_p \to O_q)$, where $-1 \leq p < n$ and $0 \leq q < n$. Then we calculate the *Interference Probabilities*:

$$P_{\mathcal{I},\mathcal{O}}(I_p, O_q) = \frac{N(I_p \to O_q)}{\sum_{-1 \leq p' < n} \sum_{0 \leq q' < n} N(I_{p'} \to O_{q'})} \quad (5)$$

From Equations 1 and 5 we can evaluate the mutual information between the victim's *actions* and attacker's *observations*. In Equation 6, we only use the range ($0 \leq p' < n$), and omit the noise represented by $p' = -1$, which we consider *fake interference*. The mutual information can accurately reflect the interference between the attacker and the victim through cache side channels.

$$I(\mathcal{I}; \mathcal{O}) = \sum_{0 \leq p' < n} \sum_{0 \leq q' < n} P_{\mathcal{I},\mathcal{O}}(I_{p'}, O_{q'}) \, log \left(\frac{P_{\mathcal{I},\mathcal{O}}(I_{p'}, O_{q'})}{P_{\mathcal{I}}(I_{p'}) P_{\mathcal{O}}(O_{q'})} \right) \quad (6)$$

We can also use the *Interference Probabilities* to identify cases of no information leakage, based on the three *non-interference* conditions in Equations 2, 3 and 4.

C1 (Output Elimination) For all $0 \leq q < n$, $P_{\mathcal{O}}(O_q)$ is close to 0. Then the attacker can not observe any information through the cache side channel, and there is no information leakage.

C2 (Noise Domination) For all $0 \leq q < n$, $P_{\mathcal{I}|\mathcal{O}(I_{-1}|O_q)}$ is close to 1. Then all of the attacker's observations are caused by the channel's noise, the victim's actions do not affect the attacker's observations. Hence there is no information leakage.

C3 (Input Ambiguity) For all $0 \leq p < n$ and $0 \leq q < n$, $P_{\mathcal{I}|\mathcal{O}}(I_p|O_q)$ is close to $P_{\mathcal{I}}(I_p)$. Then the victim's actions cannot be distinguished by the attacker's observations, and there is no information leakage.

5.2 Case Study: Type I Attacks

In Section 2.1 we classify the side-channel attacks into four categories based on the root causes of the attacks: the interference due to cache behaviors. Our cache modeling technique targets the root causes, so it can cover all the categories. As a case study, we pick type I attacks based on cache misses due to external interference, and study different caches' vulnerability to this type of side-channel attacks. Other attack types can be evaluated in a similar way.

For Type I cache attacks, the attacker observes cache misses due to external interference. Whenever a cache line occupied by an attacker is over-written (replaced) by a cache line belonging to a victim, the attacker will have an observation by detecting a cache miss when he next accesses this cache line that he had previously filled with his own data. Specifically, when the victim's cache line with a cache index of p wants to replace the attacker's cache line with a cache index of q (for conventional caches, $p = q$), a cache state transition from **A** to **V** happens, and this will be an *External Interference* (shown in Figure 1). The cause of this *External Interference* is the victim's input I_p to the cache side channel, and the result of this *External Interference* is the attacker's output O_q from the cache side channel later. So this is the interference $I_p \rightarrow O_q$.

It is also possible that the eviction of the attacker's cache line is not due to the victim's actions. This happens in RE cache when the attacker's cache line is randomly selected to be evicted out, or in RP cache when the attacker's cache line is invalidated due to the update of permutation tables. Specifically, when the attacker's cache line with a cache index of q is evicted out of the cache due to such noise, a cache state transition from **A** to **INV** happens, and we call this *Fake Interference* (Figure 1). The cause of this *Fake Interference* is the side channel's inherent noise, and the result of this *Fake Interference* is the attacker's later observations of output O_q from the cache side channel. The interference is labelled $I_{-1} \rightarrow O_q$.

To evaluate the caches' vulnerability to Type I attacks, we count the number of each kind of *External Interference* and *Fake Interference*. Then we use Equation 5 to calculate *Interference Probabilities*. Finally we calculate the mutual information in Equation 6 and use the three non-interference conditions to confirm the absence of information leakage.

We use Murphi [25] to implement our cache security models with the interference property. Murphi is a finite state machine model checker, used to verify the invariants of the system by enumerating all the explicit states. Instead of checking invariants, we use Murphi to go over all the possible cache states and record the **Information Flow Log** for each transition. Without loss of generality, we assume a 3-set, 2-way set-associative conventional cache as the baseline for this study. We used 10 rounds of memory accesses, which was enough to get stable interference probabilities. Murphi will analyze all the possible states, for all the cache lines in the cache.

5.3 Evaluation Results

5.3.1 Analysis of Non-interference Conditions

For each cache, we collect counts for the **Information Flow Log**, including counts of the number of each type of interference. From this, we calculate the joint *Interference Probability* between each input and each output.

Conventional Cache: Table 3 gives the results for the baseline conventional cache. We observe that for a conventional cache, $P_{\mathcal{I},\mathcal{O}}(I_p, O_q)$ is very distinguishable for $(q = p)$ compared to $(q \neq p)$, and none of the three conditions **C1**, **C2** or **C3** are satisfied. So the input $\mathcal{I}$ interferes highly with the output $\mathcal{O}$. We conclude that *the conventional cache is very leaky, and hence insecure.*

Table 3: *Interference Probability* for conventional cache

$P_{\mathcal{I},\mathcal{O}}(I,O)$	I_o	I_1	I_2	I_{-1}
O_0	33.3%	0.0%	0.0%	0.0%
O_1	0.0%	33.3%	0.0%	0.0%
O_2	0.0%	0.0%	33.3%	0.0%

(27,996 interferences in total)

Static-Partitioning Cache: Table 4 shows the *Interference Probability* of an SP cache. There are no attacker's observations from SP cache, satisfying condition **C1**. So *SP cache can effectively reduce Type I side-channel leakage to zero.*

Table 4: *Interference Probability* for SP cache

$P_{\mathcal{I},\mathcal{O}}(I,O)$	I_0	I_1	I_2	I_{-1}
O_0	0.0%	0.0%	0.0%	0.0%
O_1	0.0%	0.0%	0.0%	0.0%
O_2	0.0%	0.0%	0.0%	0.0%

(0 interferences in total)

Partition-Locked Cache: Tables 5a and 5b display the *Interference Probability* of PL cache without and with preload. PL cache without preload has the same interference distribution as conventional caches, indicating that *PL cache without preload can leak information when loading the victim's cache lines into the cache for the first time.* PL cache with preload has the same interference distribution as SP cache, indicating that *with proper usage like preloading the victim's sensitive cache lines, PL cache prevents information leakage.*

This example demonstrates the power of our methodology: even though the PL cache without preload may survive the "Prime and Probe" attack during experimentation, our security modeling of PL cache still reveals its vulnerability: there can still be information leakage targeting the cache warm-up stage. This agrees with the observation in [26].

Table 5: *Interference Probability* for PL cache

(a) without preload

$P_{\mathcal{I},\mathcal{O}}(I,O)$	I_0	I_1	I_2	I_{-1}
O_0	33.3%	0.0%	0.0%	0.0%
O_1	0.0%	33.3%	0.0%	0.0%
O_2	0.0%	0.0%	33.3%	0.0%

(13,794 interferences in total)

(b) with preload

$P_{\mathcal{I},\mathcal{O}}(I,O)$	I_0	I_1	I_2	I_{-1}
O_0	0.0%	0.0%	0.0%	0.0%
O_1	0.0%	0.0%	0.0%	0.0%
O_2	0.0%	0.0%	0.0%	0.0%

(0 interferences in total)

Random-Eviction Cache: Now let us consider the randomization approach. The *Interference Probability* of RE cache is shown in Table 6. The results show that a large amount of interference is fake (70.8%). Although it is still possible for the attacker to retrieve side-channel information

from the rest of the interferences (29.2%), it will be a hard job to filter out the noise due to fake interference from the observations. According to **C2**, the larger the proportion of fake interference, the more difficult it is for the attacker to retrieve useful information.

Table 6: *Interference Probability* for RE cache

$P_{\mathcal{I},\mathcal{O}}(I,O)$	I_0	I_1	I_2	I_{-1}
O_0	9.7%	0.0%	0.0%	23.6%
O_1	0.0%	9.7%	0.0%	23.6%
O_2	0.0%	0.0%	9.7%	23.6%

(117,349,797 interferences in total)

Random-Permutation Cache: Table 7 displays the Interference Probability for RP cache. We can see that *Fake Interference* constitutes a very high percentage of the total interferences (80.4 %). This is due to the cache line invalidations done when swapping mappings in the *permutation table*. In addition, the interference of each I_p on each O_q has about the same probability. This is due to the random mapping from memory address to cache set. When the attacker observes a cache miss in set q, it is hard for him to tell which set is accessed by the victim. Both *non-interference* conditions **C2** and **C3** hold, thus enhancing *the security of RP cache against Type I side-channel leakage.*

Table 7: *Interference Probability* for RP cache

$P_{\mathcal{I},\mathcal{O}}(I,O)$	I_0	I_1	I_2	I_{-1}
O_0	2.17%	2.19%	2.19%	26.8%
O_1	2.19%	2.17%	2.19%	26.8%
O_2	2.19%	2.19%	2.17%	26.8%

(7,842,324 interferences in total)

NewCache: We simulate a NewCache with 3 logical cache lines and 2 physical cache lines. Table 8 shows the Interference Probability. Due to the fully-associative mappings from the Logical Direct Mapped cache to the physical cache, the interference of I_p on any O_q happens with the same probability. According to *non-interference* condition **C3**, the attacker's observations are not affected by the victim's actions. *This means NewCache does not leak information through Type I side-channel attacks.*

Table 8: *Interference Probability* for NewCache

$P_{\mathcal{I},\mathcal{O}}(I,O)$	I_0	I_1	I_2	I_{-1}
O_0	11.1%	11.1%	11.1%	0.0%
O_1	11.1%	11.1%	11.1%	0.0%
O_2	11.1%	11.1%	11.1%	0.0%

(1,368,954 interferences in total)

5.3.2 Mutual Information

For each cache architecture, we calculate the mutual information between the victim's actions and the attacker's observations based on Equation 6 (the logarithm base is 2 and information leakage is measured in bits), as shown in Table 9. This shows that the conventional cache and *PL cache without preload* leak the most information. RE cache has a smaller mutual information value, but still gives the attacker some chances to retrieve critical information. The mutual information of RP cache is close to zero, indicating that it will be hard for the attacker to leak secrets with Type I side-channel attacks. SP cache, *PL cache with preload* and NewCache have zero mutual information, so the attacker can not get the victim's secrets from his observations.

The conclusions from mutual information are consistent with our previous analysis of non-interference *conditions.*

Table 9: Mutual Information for each cache architecture

Cache Architecture	$I(I,O)$ (bits)
Conventional	1.585
SP	0.000
PL-w/o preload	1.585
PL-w/ preload	0.000
RE	0.461
RP	2.586×10^{-6}
New	0.000

5.4 Discussion

From the above analysis we observe that different defenses usually focus on different *non-interference* conditions. For the partitioning approach, the defenses usually try to realize **C1**. The attacker cannot observe output from the channel as it is isolated from the victim (SP cache and PL cache). For the randomization approach, the defenses usually try to realize **C2** and **C3**. For **C2**, the cache adds a large amount of random noise to the attacker's observations (RE cache and RP cache). For **C3**, the cache randomizes the mappings between the victim's actions and the attacker's observations (RP cache and NewCache). This ambiguity makes it hard for the attacker to get accurate conclusions about the victim's actions. Any of the three conditions is effective at reducing the side-channel leakage. We hope this will inspire researchers to propose more defenses beyond the partitioning and randomization approaches discussed in this paper.

Our modeling methodology provides different usages: (1) a general evaluation of a cache's vulnerability to side-channel attacks, as in this paper, considers all possible cache state transitions for successive rounds of memory accesses. This will cover all possible attacks on all ciphers; (2) an evaluation of a cache's vulnerability to a specific attack on a specific cipher, can be achieved by feeding the cache models with the victim's and attacker's actual memory access traces.

6. MODELING OTHER ATTACKS

We discuss how to apply our modeling methodology to other cache side-channel attacks and cache features.

6.1 Other Cache Attacks

Our case study focused on Type I attacks: the cache misses that cause external interference between the attacker and the victim. So we consider the transitions of V_miss, from state **A** to **V**. Our model can also be applied to the other three attack categories in Table 1.

(1) *Type II Attacks*: These are based on cache misses due to internal interference, so we consider the transitions of V_miss from state **V** to **V**. The channel's input I_p is the victim's access with a cache index of p, and the output O_q is the victim's replaced cache line with a cache index of q. A random cache mapping for the victim can make $I_p \rightarrow O_q$ ambiguous, reducing the vulnerability to leak information.

(2) *Type III Attacks*: These are based on cache hits due to external interference. We need a fourth type of state **A/V**, indicating this line contains a memory line shared by the victim and the attacker. Then we consider the transitions of A_hit from state **A/V** to **A/V**. The channel's input I_p is the victim's access of a shared line with address index p, and the output O_q is the event that the attacker gets a cache hit in the cache set q. $I_p \rightarrow O_q$ denotes the interference that the attacker's line with address index p gets a cache hit in the cache set q, due to the victim's placement of this shared line. We denote I_{-1} as the event that brings the victim's line into

the cache for non-critical operations like prefetching, then $I_{-1} \rightarrow O_q$ is a *Fake Interference* which can introduce noise.

(3) *Type IV Attacks*: These are based on cache hits due to internal interference. We consider the transitions of V_hit from state **V** to **V**. The input I_p to the channel is the event that the victim brings its line with index p into the cache, and the output O_q is the event that the victim gets a cache hit when accessing a cache set q. $I_p \rightarrow O_q$ denotes the interference that the victim's line with address index p gets a cache hit in the cache set q. Similarly I_{-1} is the non-critical operations that bring in the victim's cache line, then $I_{-1} \rightarrow O_q$ is a *Fake Interference* that adds random noise.

Relevant interference probabilities, mutual information and non-interference conditions can similarly be evaluated for these types of side-channel vulnerabilities.

6.2 Other System Features

Our cache modeling methodology can also be extended to model other features of modern microprocessors.

(1) *Multiple processes*: Other processes, not the victim nor the attacker, can affect the attacker's observations. We can introduce a new state **R** to indicate the cache line occupied by the Rest of the processes. A new cache event "R_miss" can transition the cache state from **A** to **R**, generate *Fake Interference* and add noise to the attacker's observations. **R** can also be superimposed on the **INV** state, and "R_miss" on the "$Invalidate$" transition from **A** to **INV** states, in Figure 1. This generalizes the *Fake Interference* we measure as "noise" coming from multiple sources - invalidations, other processes or other channel noise.

(2) *Prefetching*: This fetches data into cache before being requested, to avoid future cache misses. We add two new events "$V_prefetch$" and "$A_prefetch$" to model the data prefetching of victim and attacker. Prefetching can reduce the side-channel leakage by adding noise. For instance, in a Type I attack, $V_prefetch$ can evict the attacker's cache line and transition the cache state from **A** to **V**. This generates *Fake Interference* when the attacker has a cache miss later for this line, and may satisfy Condition **C2**. $A_prefetch$ can evict the victim's cache line and transition the cache state from **V** to **A**. Then the attacker will experience a cache hit for this line, and cannot observe the victim's previous actions. This may satisfy Condition **C1**.

(3) *AES-NI* [27]: Intel x86 processors introduced new instructions specifically for AES block encryption and decryption. This eliminates cache usage for those AES implementations rewritten using AES-NI instructions. Since this removes state **V** and all transitions related to it from Figure 1, the attacker cannot observe any interference from the victim and Condition **C1** is satisfied. Unfortunately, AES-NI does not apply to legacy code (or new code written without using AES-NI instructions). It does not mitigate the cache side-channel leakage from other ciphers, e.g., RSA. It also does not protect cache side-channel leakage in other processors, e.g., the dominant ARM processors used in mobile devices.

7. VALIDATION OF CACHE MODELS

We launch an actual attack program to see if this validates the results of our cache modeling case study. We use the *Probability Distribution of Candidate Keys* to quantify the feasibility of this attack, hence the system's vulnerability.

7.1 Probability Distribution of Candidate Keys

When an attacker attempts to break a cryptography computing platform, he can feed different plaintexts into the platform, and ask for encryption with the key he wants to steal. He then collects the side-channel observations during the encryptions and tries to infer the keys or narrow down its possible values. To improve the accuracy and fully recover the keys, the attacker usually repeats a bunch of attack rounds to obtain different candidate keys. He calculates the *Probability Distribution of Candidate Keys*, and selects the key with the highest probability. A successful attack is able to select the correct key based on its significantly higher probability in the *Probability Distribution of Candidate Keys*. So we use this to evaluate the success of an attack, and hence the systems' vulnerability to this attack.

7.2 Implementation

We launch an access-based side-channel attack on AES [17]. We use gem5 [28] to simulate this attack on different cache architectures, and compare their *Probability Distribution of Candidate Keys*. For each cache, we simulate L1 caches, with cache size of 32 Kbytes, line size of 32 bytes and set-associativity of 8-way, which is a typical configuration in modern processors. The victim runs AES encryption for 2^{18} random blocks in the *study* phase (the key is known to the attacker) and in the *attack* phase (the key is unknown to the attacker). The attacker *primes and probes* the cache continuously to collect the access time for each cache line, to infer the victim's memory accesses, and hence his encryption key.

7.3 Attack Results

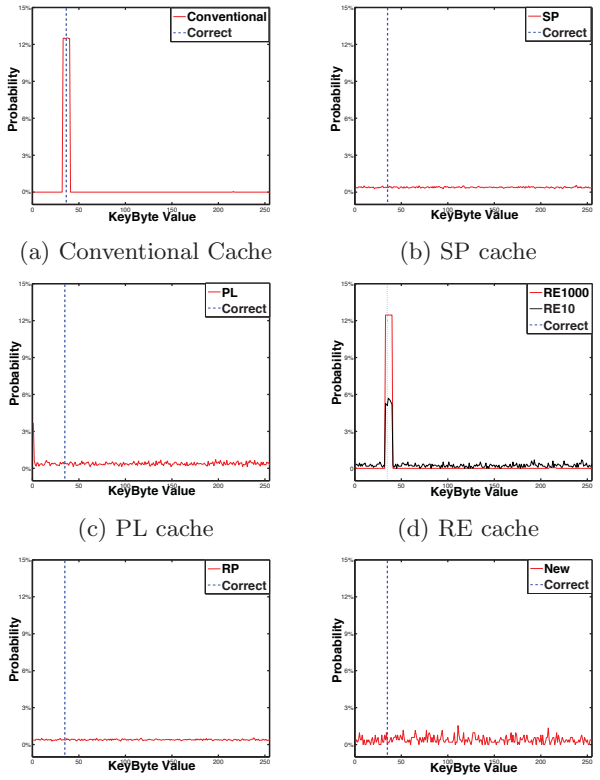
Figure 2 shows the attack results. We use a solid red line to show the *Probability Distribution of Candidate Keys*, and a dotted blue line to denote the correct encryption key.

For conventional caches (Figure 2a), eight keys (32-39) get more than 10 % probabilities while the rest are close to zero. The correct key-byte value 35 is among the top eight candidate keys, but the attacker cannot pick it out. This is because one cache line contains 8 AES entries, and the attacker is unable to differentiate which entry is actually accessed, when he observes a victim's access to this line. He needs other methods like brute-force or two-rounds attack [17] to find the correct key from the 8 possible values. We conclude this attack on the conventional cache succeeds.

Similar results can be observed for Random Eviction cache (Figure 2d). This shows two types of RE cache: RE1000 (a random cache line is evicted every 1000 memory accesses – in red) and RE10 (a line is evicted every 10 memory accesses – in black). RE cache also leaks eight candidate keys: RE1000 has the same probability distribution as conventional caches, while RE10 is much smaller. A RE cache with more frequent random evictions is more difficult to attack.

Figure 2b shows the distribution of candidate keys for SP cache. This shows that the attack does not produce any distinguished candidate keys. Thus this attack on SP cache fails at 2^{18} samples. We get the same conclusion for Partition-Locked Cache (Figure 2c), Random-Permutation Cache (Figure 2e) and Newcache (Figure 2f).

Comparing Table 9 and Figure 2, we see that the results we get from the two independent evaluation methods are consistent. For the partitioning approach, PL and SP caches can effectively defend against Type I side-channel attacks. For the randomization approach, RP and NewCache are also very effective in reducing Type I side-channel leakage. RE



(a) Conventional Cache (b) SP cache

(c) PL cache (d) RE cache

(e) RP cache (f) NewCache

Figure 2: Probability Distribution of Candidate Keys

caches are attackable, but if the eviction frequency increases (at the cost of performance), the attack becomes harder.

In several cases, our abstract cache models give more information than the actual attacks. For example, they can show the effect of PL cache with preloading of the sensitive table data (not vulnerable) versus PL cache with no preloading (vulnerable), as shown in Table 5.

8. RELATED WORK

Past work in evaluating side-channel attacks can be classified into several categories based on their features:

Mutual Information: [29] applied mutual information and Guessing Entropy to evaluate the feasibility of key-recovery. [30] proposed static analysis to establish formal security guarantees against cache side-channel attacks, and can estimate the upper bound of information leakage from side channels. [31] presented an information-theoretic metric for adaptive side-channel attacks, which can estimate the attacker's remaining uncertainty for adjusting his strategy.

Success Probability: In [29], Success Rate is defined as a general metric to evaluate the feasibility of side-channel key recovery. Then [32] defines the average Success Rate to evaluate the profiled cache timing attacks. It also builds an analytical model to estimate the Success Rate for determining the best attack strategy. [33] builds a predictive model for evaluating the side-channel leakage through caches. It calculates the probability that the attacker can correctly detect a memory access given a victim's critical memory access.

Correlation Metric: [34] proposed the Side-channel Vulnerability Factor (SVF) to measure a system's vulnerability to all side channels. It calculates the Pearson Correlation Coefficient between the Similarity Matrices of the victim's execution traces and the attacker's observation traces. [35] proposed the timing-SVF metric for timing-based cache side-channel attacks, which [34] did not address. In [36], a metric called Cache Side-channel Vulnerability (CSV) is designed to overcome SVF's issues in its scope, definition and measurements. It also states that using a single metric like SVF [34] to evaluate the system's vulnerability to all possible forms of side-channel information leakage is problematic as it may give misleading results and furthermore, it does not correctly determine which secure cache designs are more effective in defending against which side-channel attacks.

Formal Verification: Porras and Kemmerer designed the technique of covert flow trees to systematically detect and identify covert channels between processes [37]. [38] built models of timing side-channel leakage from the program code level. [39] uses the technique of self-composition to verify the non-interference properties of cryptographic software by considering two copies of the program.

Most of the above methods aim to evaluate the feasibility of the attacker's behaviors, instead of the system's intrinsic vulnerability. Unlike these past methods, we are the first to model the cache architectures and measure their leakage.

9. CONCLUSIONS

This work proposes a novel methodology to evaluate a cache system's vulnerability to side-channel attacks. We model side channel leakage from the *non-interference* property, and use mutual information with three *non-interference* conditions to guarantee no side-channel leakage. We then show how to model cache architectures, and integrate these with our side-channel leakage model. We also perform a real attack on each of our detailed secure cache architecture simulations, to see if our model is consistent with reality ("ground truth"), to validate our modeling methodology.

Our modeling methodology focuses on the root cause of cache side-channel leakage: the interference impacting cache behavior. It can theoretically cover all types of side-channel attacks (known or unknown). In the case study, we consider the side-channel attacks (Type I) based on cache misses due to external interference. But these models of caches and interferences can be extended to other types of attacks, as discussed in Section 6. Future work can also extend these evaluation methods to new cache architectures and other system features impacting cache behavior, as well as to other subsystems (not just caches) that may be vulnerable to side-channel or covert channel attacks.

10. ACKNOWLEDGMENTS

This work was supported in part by DHS/AFRL FA8750-12-2-0295 and NSF CNS-1218817.

11. REFERENCES

[1] D. Gullasch, E. Bangerter, and S. Krenn, "Cache games–bringing access-based cache attacks on aes to practice," in *IEEE Symp. on Security and Privacy*, 2011.

[2] T. S. Messerges, E. A. Dabbish, and R. H. Sloan, "Investigations of power analysis attacks on smartcards," in *USENIX Workshop on Smartcard Technology*, 1999.

[3] P. C. Kocher, J. Jaffe, and B. Jun, "Differential power analysis," in *Annual Intl. Cryptology Conference on Advances in Cryptology*, 1999.

[4] N. Homma, T. Aoki, and A. Satoh, "Electromagnetic information leakage for side-channel analysis of cryptographic modules," in *IEEE Intl. Symp. on Electromagnetic Compatibility*, 2010.

[5] P. Kocher, R. Lee, G. McGraw, and A. Raghunathan, "Security as a new dimension in embedded system design," in *Design Automation Conference*, 2004.

[6] J.-F. Dhem, F. Koeune, P.-A. Leroux, P. Mestré, J.-J. Quisquater, and J.-L. Willems, "A practical implementation of the timing attack," in *Intl. Conf. on Smart Card Research and Applications*, 2000.

[7] C. Percival, "Cache missing for fun and profit," in *Proc. of BSDCan*, 2005.

[8] D. J. Bernstein, "Cache-timing attacks on aes," tech. rep., 2005.

[9] J. Bonneau and I. Mironov, "Cache-collision timing attacks against aes," in *Lecture Notes in Computer Science series 4249*, Springer, 2006.

[10] Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart, "Cross-vm side channels and their use to extract private keys," in *ACM Conference on Computer and Communications Security*, 2012.

[11] E. Brickell, G. Graunke, M. Neve, and J.-P. Seifert, "Software mitigations to hedge aes against cache-based software side channel vulnerabilities," 2006.

[12] E. Käsper and P. Schwabe, "Faster and timing-attack resistant aes-gcm," in *Cryptographic Hardware and Embedded Systems*, 2009.

[13] Z. Wang and R. B. Lee, "New cache designs for thwarting software cache-based side channel attacks," in *ACM/IEEE Intl. Symp. on Computer Architecture*, 2007.

[14] Z. Wang and R. Lee, "A novel cache architecture with enhanced performance and security," in *IEEE/ACM Intl. Symp. on Microarchitecture*, 2008.

[15] L. Domnitser, A. Jaleel, J. Loew, N. Abu-Ghazaleh, and D. Ponomarev, "Non-monopolizable caches: Low-complexity mitigation of cache side channel attacks," *ACM Trans. Archit. Code Optim.*, 2012.

[16] O. Aciiçmez and c. K. Koç, "Trace-driven cache attacks on aes," in *Intl. Conference on Information and Communications Security*, 2006.

[17] D. A. Osvik, A. Shamir, and E. Tromer, "Cache attacks and countermeasures: the case of aes," in *RSA conference on Topics in Cryptology*, 2006.

[18] O. Aciiçmez, "Yet another microarchitectural attack: exploiting i-cache," in *ACM workshop on Computer security architecture*, 2007.

[19] Z. Wang, *Information Leakage Due to Cache and Processor Architectures*. PhD thesis, Princeton, 2012.

[20] Z. Wang and R. B. Lee, "Covert and side channels due to processor architecture," in *Annual Computer Security Applications Conference*, 2006.

[21] F. Liu and R. B. Lee, "Security testing of a secure cache design," in *Hardware and Architectural Support for Security and Privacy*, 2013.

[22] J. A. Goguen and J. Meseguer, "Security policies and security models," in *IEEE Symp. on Security and Privacy*, 1982.

[23] T. M. Cover and J. A. Thomas, *Elements of Information Theory*. Wiley InterScience, 2006.

[24] T. Zhang and R. B. Lee, "Secure Cache Modeling for Measuring Side-channel Leakage," in *Tech. Report, http://palms.ee.princeton.edu/node/428*.

[25] D. L. Dill, A. J. Drexler, A. J. Hu, and C. H. Yang, "Protocol verification as a hardware design aid," in *Intl. Conference on Computer Design: VLSI in Computer & Processors*, 1992.

[26] J. Kong, O. Aciicmez, J.-P. Seifert, and H. Zhou, "Hardware-software integrated approaches to defend against software cache-based side channel attacks," in *IEEE Intl. Symp. on High Performance Computer Architecture*, 2009.

[27] S. Gueron, "Intel advanced encryption standard (aes) instructions set," 2010.

[28] "The gem5 simulator system," in *http://www.gem5.org*.

[29] F.-X. Standaert, T. G. Malkin, and M. Yung, "A unified framework for the analysis of side-channel key recovery attacks," in *Annual Intl. Conference on Advances in Cryptology: the Theory and Applications of Cryptographic Techniques*, 2009.

[30] B. Köpf, L. Mauborgne, and M. Ochoa, "Automatic quantification of cache side-channels," in *Intl. Conference on Computer Aided Verification*, 2012.

[31] B. Köpf and D. Basin, "An information-theoretic model for adaptive side-channel attacks," in *ACM Conf. on Computer and Comms. Security*, 2007.

[32] C. Rebeiro and D. Mukhopadhyay, "Boosting profiled cache timing attacks with a priori analysis," *IEEE Trans. on Information Forensics and Security*, 2012.

[33] L. Domnitser, N. Abu-Ghazaleh, and D. Ponomarev, "A predictive model for cache-based side channels in multicore and multithreaded microprocessors," in *Intl. Conference on Mathematical Methods, Models and Architectures for Computer Network Security*, 2010.

[34] J. Demme, R. Martin, A. Waksman, and S. Sethumadhavan, "Side-channel vulnerability factor: a metric for measuring information leakage," in *ACM/IEEE Intl. Symp. on Computer Architecture*, 2012.

[35] S. Bhattacharya, C. Rebeiro, and D. Mukhopadhyay, "Hardware prefetchers leak: A revisit of SVF for cache-timing attacks," in *Hardware and Architectural Support for Security and Privacy*, 2012.

[36] T. Zhang, S. Chen, F. Liu, and R. B. Lee, "Side channel vulnerability metrics: the promise and the pitfalls," in *Hardware and Architectural Support for Security and Privacy*, 2013.

[37] P. Porras and R. Kemmerer, "Covert flow trees: a technique for identifying and analyzing covert storage channels," in *IEEE Computer Society Symp. on Research in Security and Privacy*, 1991.

[38] J. Svenningsson and D. Sands, "Specification and verification of side channel declassification," in *Intl. Conf. on Formal Aspects in Security and Trust*, 2010.

[39] J. Bacelar Almeida, M. Barbosa, J. S. Pinto, and B. Vieira, "Formal verification of side-channel countermeasures using self-composition," *Sci. Comput. Program.*, 2013.

ICE: A Passive, High-Speed, State-Continuity Scheme

Raoul Strackx
iMinds-Distrinet, KU Leuven
Celestijnenlaan 200A
3001 Heverlee

Bart Jacobs
iMinds-Distrinet, KU Leuven
Celestijnenlaan 200A
3001 Heverlee

Frank Piessens
iMinds-Distrinet, KU Leuven
Celestijnenlaan 200A
3001 Heverlee

first.last@cs.kuleuven.be

ABSTRACT

The amount of trust that can be placed in commodity computing platforms is limited by the likelihood of vulnerabilities in their huge software stacks. Protected-module architectures, such as Intel SGX, provide an interesting alternative by isolating the execution of software modules. To minimize the amount of code that provides support for the protected-module architecture, persistent storage of (confidentiality and integrity protected) states of modules can be delegated to the untrusted operating system. But precautions should be taken to ensure *state continuity*: an attacker should not be able to cause a module to use stale states (a so-called *rollback attack*), and while the system is not under attack, a module should always be able to make progress, even when the system could crash or lose power at unexpected, random points in time (i.e., the system should be *crash resilient*).

Providing state-continuity support is non-trivial as many algorithms are vulnerable to attack, require on-chip non-volatile memory, wear-out existing off-chip secure non-volatile memory and/or are too slow for many applications.

We present ICE, a system and algorithm providing state-continuity guarantees to protected modules. ICE's novelty lies in the facts that (1) it does not rely on secure non-volatile storage for every state update (e.g., the slow TPM chip). (2) ICE is a passive security measure. An attacker interrupting the main power supply or any other source of power, cannot break state-continuity. (3) Benchmarks show that ICE already enables state-continuous updates almost 5x faster than writing to TPM NVRAM. With dedicated hardware, performance can be increased 2 orders of magnitude.

ICE's security properties are guaranteed by means of a machine-checked proof and a prototype implementation is evaluated on commodity hardware.

1. INTRODUCTION

Protection of sensitive data in commodity computing platforms is extremely challenging. Modern operating systems provide process isolation primitives, but the kernel is too large to be implemented free from vulnerabilities. Moreover, commodity systems are prone to physical attacks, even by ill-equipped and resource-constrained home users. These vulnerabilities limit the amount of trust that can be placed in commodity systems. In servers these limitations are remedied by programmable hardware security modules (HSMs). On client devices, highly-sensitive applications such as online banking or e-government often resort to smart cards. Unfortunately, these solutions are expensive, cumbersome and the security guarantees that they can provide to the overall applications are limited.

Two recent advances in computer security indicate that this situation may change in the near future. First, protected-module architectures (PMAs) have been developed that provide strong isolation directly to modules running at application level [3, 4, 16, 17, 19, 28, 29, 36]. The OS is still relied upon to provide services such as disk and network access, but they are *not* trusted. Protected modules' memory regions cannot be accessed from unprotected memory; modules are in complete control over their own content and can only be accessed through the interface they expose. Last year Intel disclosed their work on Software Guard eXtension (SGX) [2, 9, 18], their own hardware-implemented PMA for commodity processors. SGX goes even further than other state-of-the-art PMAs and also provides protection against hardware attacks; modules (called enclaves in SGX[1]) are only stored in plaintext within the CPU package. When they are evicted to main memory they are confidentiality, integrity and version protected.

Second, Agten et al. [1] and Patrignani et al. [21, 22] proposed fully-abstract compilation techniques to protected-module architectures. While the strong isolation guarantees offered by these architectures is vital, they are difficult to implement without compiler support. Care must be taken not to introduce software vulnerabilities during compilation. Fully-abstract compilation ensures just this; machine-code-level attacks exists *iff* also a corresponding attack at source-code level exists. This enables easy reasoning and verification of the security guarantees these modules provide.

Unfortunately an important attack vector has been largely overlooked. Protected-module architectures, including SGX, only provide strong isolation guarantees *while the system executes continuously*. Without support for state continuity, protected modules need to remain stateless, significantly

[1] We will use the term "protected module" when referring to isolated memory areas in *any* protected-module architecture and use "enclave" when referring to SGX specifically.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.
ACSAC '14, December 08–12, 2014, New Orleans, LA, USA.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-3005-3/14/12 ...$15.00.
http://dx.doi.org/10.1145/2664243.2664259.

hampering their applicability. Consider as a running example a password-checking module. To defend against dictionary attacks, the user will be locked out indefinitely after three failed attempts. The module confidentiality and integrity protects its state before handing it to the untrusted operating system for storage. But when the module needs to recover its state after a reboot, it cannot distinguish between a fresh and a stale state and the guess-limited security measure cannot be guaranteed.

In practice most applications and protocols rely on state-continuity guarantees; firewall settings most not be revertible, attackers must not be able to tamper with log files, revoked user credentials must not be rolled back, cryptographic nonces must never be re-used, etc. Support for state continuity may also provide stronger security guarantees. Chun et al. for example proposed append-only memory [5] to harden existing distributed algorithms and applications such as NFS. Acting as a trusted log, this memory protects against equivocation; the ability of a network node to make contradicting statements to different entities.

While at first glance having similarities with replay attacks, the state itself is replayed in a rollback attack. Providing support for state continuity is therefore much harder, especially when practical limitations are considered. Parno et al. [20] show that many seemingly obvious algorithms are flawed. Others are prone to simple hardware attacks. Attaching an uninterruptible power source (UPS), for example, may simply be disconnected. Or an in-kernel attacker may prevent the execution of the interrupt handlers it relies upon. Adding non-volatile memory on-chip could simplify a solution, but requires modification of manufacturing processes leading to increased manufacturing costs. Alternatively, using non-volatile memory off-chip (e.g., isolating disk space) may be susceptible to a clone attack where a hardware-level attacker may easily overwrite the state with a previously recorded stale state. Using TPM NVRAM or TPM monotonic counters instead, would foil such attacks, but would significantly impact performance and usability. Most implementations only provide 1,280 bytes of NVRAM that supports only 100,000 write cycles over the chip's lifetime [20]. Accessing NVRAM every second, would wear it out in less then 28 hours. Monotonic counters, on the other hand, only need to be incrementable every 5 seconds [35].

Hardware upgrades to the TPM chip could reduce some of these architectural constraints, at an economic cost. However, any solution placing the TPM on the performance-critical path, would require additional upgrades over time to bridge the ever growing TPM/CPU performance gap. We present ICE, an alternative solution that only requires TPM accesses at boot time and is thus *not* affected by TPM speed.

ICE avoids architectural challenges (1) by proposing a simple implementation technique where on-chip dedicated registers are backed off-chip by a capacitor and persistent memory. Upon a sudden loss of power, the contents of the dedicated registers is written to persistent memory. (2) ICE is a *passive* protection scheme; in the event of a crash or power loss, security is guaranteed instantly. A hardware attacker may disconnect the capacitor, but state continuity remains guaranteed. (3) At the moment freshness information is backed to persistent storage, it is considered public data. Overwriting it with stale freshness information will be detected upon recovery.

In summary, we make the following contributions:

- We present ICE, the first algorithm providing state-continuity guarantees with a minimal TCB that does *not* rely on the speed of secure, non-volatile memory (e.g., the (slow) TPM chip) nor does it rely on an uninterruptible power source.
- We formally verify and machine check the security properties of ICE using the Coq proof assistant.
- Because SGX-enabled machines or emulators are not yet available, we validate our claims based on a prototype implementation on top of Fides [28], an existing hypervisor-based protected module architecture similar to SGX. Benchmarks show that states can already be stored almost 5x faster on commodity hardware than writing to TPM NVRAM. Dedicated hardware support would increase performance substantially.

The remainder of this paper is structured as follows. First we detail our attack model and the security properties that we need to guarantee. Next in Sections 3 and 4, we present our algorithm and discuss two possible implementations. Finally, we evaluate the security and performance of ICE.

2. PROBLEM DEFINITION

2.1 Attacker Model

ICE can defend against an attacker with three powerful capabilities. First, we assume that an attacker is able to compromise the entire software stack, with the exception of ICE-implementing modules. This enables versatile attacks ranging from modifying the contents of the hard drive to preventing enclaves from ever resuming execution.

Second, we assume that an attacker has control over the system's power supply or is able to launch attacks leading to a similar result. Power-interruption attacks differ from kernel-level crashes as they also affect software modules executing in complete isolation from the rest of the system: modules may stop executing before they can commit their new state. SGX enclaves are especially vulnerable to such attacks. In order to prevent denial-of-service attacks by malicious enclaves that never return control to the kernel, SGX supports interruption of enclaves [11]. When the interrupt is handled in the untrusted kernel, an in-kernel attacker can easily prevent the enclave from ever resuming execution.

Third, we consider hardware attacks. We implement ICE as a library that modules can be statically linked with and take advantage of the security guarantees provided by the protected-module architecture. In case of SGX this implies that an attacker may place probes on memory buses or perform cold boot attacks [8]. Defending against physical attacks against the CPU package itself or the TPM chip [26,32,38] are orthogonal problems and not considered.

2.2 Security Properties

State continuity can be factored into two properties: safety and liveness. To ensure safety, ICE must be resilient against a *rollback attack* where an attacker provides the module with a valid, but stale state. A rollback attack is related to a replay attack but it is much harder to defend against. Where in a replay attack identical input is provided, the state of the module itself is replayed in a rollback attack.

The second property, liveness, states that benign events should never force the system into a state from which it can-

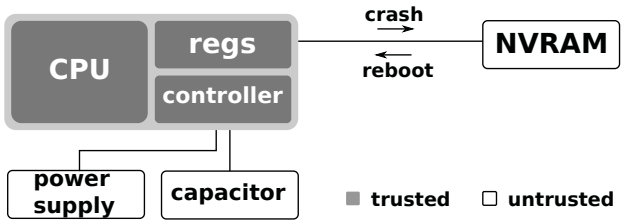


Figure 1: Architecture of guarded memory. When power suddenly fails on-chip dedicated registers are backed up to off-chip, shadow memory (NVRAM).

not progress. In practice this means that the system should be allowed to crash at any time during the operation of the algorithm, including when it is recovering from a previous crash. Note that this is not the same as protection against denial-of-service. Protection against denial-of-service is not in scope; in-kernel attackers can easily prevent the system from progressing (e.g., by breaking the kernel). Liveness only ensures progress is not hampered by *random* crashes, which may also occur when the system is not under attack.

3. STATE-CONTINUITY AS A LIBRARY

Before introducing a running example and describing ICE in full detail, we first introduce the system hardware we rely on and discuss how freshness information is recorded.

3.1 Architecture

Assuming ICE is implemented on top of Intel SGX, we only place trust in the CPU package and TPM chip. Attacks against any other component cannot compromise security.

Enclaves.

Intel SGX provides enclaves with total control over their own code and data by enforcing a specific access control mechanism; *only* when executing within the boundaries of an enclave can its content be accessed. Access attempts from code running at *any* privilege level outside the enclave (including from other enclaves), will be blocked. Enclaves can only be accessed through an explicitly exposed interface.

TPM.

We store long term secrets and freshness information in TPM NVRAM. These secrets should only be accessible from the SGX enclave that provided them.

Guarded Memory.

To enable fast state updates, we propose the addition of a small amount of *guarded memory*; dedicated registers on-chip that are backed off-chip by shadow, non-volatile memory (NVRAM) and a capacitor (see Fig. 1). When a controller detects that the main power supply is disconnected from the CPU package, it copies the registers' content to non-volatile memory. When power is re-applied, the registers are restored. Note that *only* on-chip components need to be trusted. Attacks against shadow memory, main power supply or the capacitor cannot break state continuity.

Persistent Storage.

ICE uses operating system services to access persistent storage. These services are not trusted: an attacker may copy, replace and destroy files. To differentiate between the actual state of a module and states stored on disk, we call the latter *(ICE) cubes* whenever ambiguity might arise.

3.2 Guards: Storing Freshness Info

Just as message authentication codes (MACs) can be used to guarantee message integrity, we will use *guards* to prove that a cube is fresh. Guards are 2-tuples:

$$\text{guard}_i(n) = (\underbrace{\text{Hash}^i(n)}_{\text{guard value}}, \underbrace{i}_{\text{guard index}})$$

where the first element, the *guard value* represents the hash value after hashing the base value i times, the *guard index*.

A guard is incremented by hashing the guard value and incrementing the index:

$$\text{guard}_i(n) = (\text{Hash}^i(n), i)$$
$$\text{guard}_{i+1}(n) = (\text{Hash}^{i+1}(n), i+1)$$

Based on the construction of guards, they possess two important properties: (1) two guards can be compared based on the guard index:

$$(n,i) \leq (m,j) \Leftrightarrow \begin{cases} n = m & \text{if } i = j \\ (\text{Hash}(n), i+1) \leq (m,j) & \text{if } i < j \end{cases}$$

and, (2) an attacker is unable to calculate any preceding guard as this would imply inverting the hash function.

3.3 ChkPassword: A Running Toy Example

Guaranteeing state-continuity is non-trivial and can only be accomplished by a module provider taking the required safety precautions. We only provide a library offering state-continuous storage. To demonstrate the subtle vulnerabilities that need to be resolved, consider as a running example `ChkPassword`, a password-checking module displayed in listing 1. It exposes an interface of two functions: `set_passwd` that modifies the user's password and `check_passwd`[2] that handles login attempts. To prevent dictionary attacks, `ChkPassword` will lock out a user indefinitely after 3 incorrect attempts. We assume that when the module is created, the `INIT` function is called before any service call is handled. When `ChkPassword` executes on the platform for the first time, a default password is selected (line 7), otherwise its previous state is restored (line 10).

To ensure state continuity, `ChkPassword` needs to fulfill three requirements. First, it must protect against subtle timing attacks. When an attacker is able to infer that the provided password is incorrect based on timing differences between a correct and incorrect password[3], she may be able to crash the system before the login attempt could be recorded. Ensuring that each execution path takes exactly the same amount of CPU cycles is hard. Similar to Parno et al. [20], we take a much simpler approach and store the state with the newly provided input *before* it is used in any computation. Hence, `ChkPassword` stores its current state

[2]Calling `ChkPassword` from unprotected memory would enable an attacker to intercept the provided password before it reaches the module. Users of `ChkPassword` should establish a secure channel from another module before exchanging sensitive data [2, 28]. Such considerations are out of scope.

[3]A similar attack exists when a (unique) callback to unprotected memory is made before an undesirable state is stored.

```
static int attempts_left;
static char *password;

void INIT( void ) {
  State *state;
  if (retrieve( &state ) == UNINITIALIZED){
      password = "default";
      attempts_left = 3;
  } else
      restore_and_restart( state );
}

int ENTRY_POINT check_passwd(char *guess) {
  State *state = new State();

  //store (input, state) tuple
  collect_state( state );
  collect_entry( state, "CHECK_PASSWD" );
  collect_input( state, guess );
  store( state );

  //check passwd
  if ( attempts_left > 0 &&
       strcmp( password, guess ) == 0 ) {
    attempts_left = 3;
    return OK;
  } else {
    attempts_left = max(attempts_left -1,0);
    return INCORRECT;
  }
}

int ENTRY_POINT set_passwd(char *oldpwd,
    char *newpwd) { ... }
```

Listing 1: ChkPassword: A running example

(the number of attempts left and the correct password) together with the provided guess (line 17-20) before checking the provided password. An unexpected crash while the password is being verified (i.e., after line 20), will then result in the current state being restored and execution is restarted; another attempt is made to check the *same* provided password. We assume `restore_and_restart` restores the current state and restarts execution of the last called entry point (line 10). Alternatively, if the system crashed before the input could be recorded and thus was never used in any meaningful computation (i.e., before line 20), the provided guess can simply be discarded.

Second, in order to guarantee that re-execution of the same input on the same state always leads to an identical result, modules must be deterministic. This implies that modules must consider all sources of non-determinism (e.g., the result of a random number generator) as input and thus store such data before using it in any computation.

Third, an attacker must not be able to infer any value from the size of the stored states on disk; modules must ensure that all cubes are equal in size.

3.4 ICE Libraries

We will provide state-continuous storage in two steps. In Section 3.4.1 we introduce `libice0`, a library providing support at the cost of scarce platform resources for every instance. Then in Section 3.4.2, we present `libicen` that alleviates resource pressure by storing freshness information

```
void store( State *state ) {
  switch (ice.mode) {
    case Clear:
      return _init_state( state );
    case Activated:
      return _update_state( state );
} }

int retrieve( State **state ) {
  switch (tpm.mode) {
    case Clear:
      return UNINITIALIZED;
    case Activated:
      *state = _recovery_step();
      return RECOVERED;
} }
```

Listing 2: `libice0` relies on `tpm.mode` and `ice.mode` to distinguish between storing an initial state, updating a stored state and recovery

in a single, state-continuous module `ice0`. As all `libicen` library instances connect to the same, unique `ice0` instance, a virtually unlimited number of modules is supported.

Both `libice0` and `libicen` provide the same interface: `store(State *)` and `retrieve(State **)`. To avoid repeated TPM or `ice0` accesses, `libice0` and `libicen` keep a cached copy. In order to distinguish between these copies and explicitly state where they are stored, we will reference them similarly to fields of a struct. For example, the encryption and MAC keys stored in the TPM chip will be referenced as `tpm.keys`. The variables used by the ICE algorithm are referenced as `ice.keys` and so on. Besides storing keys and the guard we also keep track of the state of the algorithm using a `mode` variable. Stored inside the TPM chip (`tpm.mode`), this variable indicates whether ICE was once initiated correctly. In `libice0` (`ice.mode`) this variable is used to indicate whether ICE was initiated or recovered since reboot. We assume that when a module is resurrected after a crash, `ice.mode` is initialized with value `Clear`. As a shorthand, we also assume that setting this variable takes exclusive access of guarded memory. Listing 2 uses these variables to differentiate between an initial state being stored and a state being updated. Similarly, `tpm.mode` is used to determine whether a state was ever stored.

3.4.1 `libice0`: *State-Cont. Storage for One Module*

In order to provide state continuity, we must guarantee that an attacker is not able to fabricate recorded states (called cubes) and that no stale cubes can be provided as being fresh. The former is trivially guaranteed by including a message authentication code in each cube. Guaranteeing freshness is more challenging, but as modules maintain their state between invocations, we only need to consider power off and reboot events. Let's call events during such power cycles an *execution stream*. An execution stream starts by either storing an initial state of a module or when the state of a module is recovered after a crash. It ends when the system crashes or when it is shut down properly.

To keep track of the fresh cube, we will generate a (base) guard when the execution stream starts and store it securely in TPM NVRAM. For every state the module requests storage of in the current execution stream, we will increment the

```
void _init_state( State *state ) {
  ice.guard = gen_guard();
  ice.keys = gen_keys();
  hdd.write( new Cube( ice.guard, ice.keys,
       state ));
  ice.mode = Activated;
  gmem.guard = ice.guard;
  tpm.guard = ice.guard;
  tpm.keys = ice.keys;
  tpm.mode = Activated;
}
```

Listing 3: `libice0`: Storing the initial state

```
void _update_state( State *state ) {
  ice.guard = ++ice.guard;
  hdd.write( new Cube( ice.guard, ice.keys,
       state ));
  gmem.guard = ice.guard;
}
```

Listing 4: `libice0`: Updating a state

guard and include it in the generated cube. Using guarded memory we will ensure that *only* the guard included in the last (and thus fresh) cube is leaked at the moment the system crashes. As no preceding guards were leaked (and cannot be calculated), it serves as a pointer to the fresh cube. Upon recovery, knowledge of the guard[4] that is stored in the provided cube, proves that the cube is fresh.

Creation of an Initial State.

When storage of the initial state of the module is requested, a new base guard and keys are generated (see listing 3). Next, a new cube is constructed and written to disk. Exclusive access of guarded memory is taken by setting the `ice.mode` variable to `Activated` and the fresh guard is written to guarded memory. In case exclusive access cannot be assigned (i.e., another module already received it), the module simply stops its execution. For clarity, such error handling is not displayed. Finally the keys and guard are stored in the TPM's NVRAM and `tpm.mode` is set to `Activated`, committing the start of a new execution stream.

Updating a State.

When storage of a new input-state pair is requested in the same execution stream, the previously used guard and keys are still stored in `libice0`'s memory and no TPM accesses are required. To safely store the input-state pair, a new cube is created with the subsequent guard and stored on disk (listing 4). Finally the fresh guard is written to guarded memory, committing the step.

Recovering from a Crash.

Recovering from a crash is more challenging and is performed in two steps (see listing 5, error handling is omitted for clarity). First, the last stored cube is read from disk. By verifying three properties its freshness is ensured:

[4]We must also check that this guard was created during the last execution stream as a matching guard/cube is found at the end of every execution stream.

```
1  State *_recovery_step() {
     Cube cube = hdd.read();
3    if ( is_fresh( &cube ) ) {
       State *state = extract(cube, tpm.keys);
5      ice.guard = gen_guard();
       ice.keys = tpm.keys;
7      hdd.write( new Cube( ice.guard,
         ice.keys, state ) );
9      ice.mode = Activated;
       gmem.guard = ice.guard;
11     tpm.guard = ice.guard;
       return state;
13   }
     else abort();
15 }

17 bool is_fresh( Cube *cube ) {
     return ( check_mac( cube, tpm.keys ) &&
19     tpm.guard ≤ cube->guard &&
       gmem.guard.value == cube->guard.value);
21 }

23 bool operator≤( Guard g1, Guard g2 ) {
     while ( g1.index < g2.index ) {
25     g1.value = Hash( g1.value );
       ++g1.index;
27   }
     return g1.value == g2.value;
29 }
```

Listing 5: `libice0`: Recovering from a crash.

- *Validity*: Cubes must not have been forged. This is ensured by the MAC stored in each cube and the accompanying key stored securely in the TPM (line 17).
- *Correct execution stream*: The cube received from the untrusted OS must have been created during the last execution stream. Starting each execution stream, a new base guard is generated and stored safely in TPM NVRAM. All guards used during this execution stream are successors of this base guard. Hence, the cube was created during the last execution stream iff (line 18):

$$\text{tpm.guard} \leq \text{cube.guard}$$

- *Public guard*: `libice0` ensures that guarded memory always contains the same guard as the last (fresh) cube stored on disk[5], and that no preceding guards leak or can be calculated. Hence, if the guard stored in guarded memory matches the guard included in the cube at hand and the two previous properties hold as well, it is guaranteed that the cube is fresh (line 19).

In the second step the fresh state is re-stored as part of a new execution stream: `libice0`'s variables are restored from TPM NVRAM, a new base guard is generated, the fresh state packaged in a new cube and the base guard is written to guarded and TPM NVRAM memory. To ensure that after an unexpected crash during the execution of this step, recovery can be restarted, `libice0` must (1) backup the previous fresh guard before overwriting it in guarded

[5]There is one exception as writing cubes to disk and updating guarded memory cannot be executed atomically. This exception is resolved later in this section.

memory. As this value is public, any persistent storage can be used (for clarity not displayed in listing 5). (2) The new base guard is written to TPM NVRAM as the last step.

Let's reconsider `ChkPassword` and discuss how crashes are resolved. Depending on the timing of a crash, we can differentiate between three main situations. One, `ChkPassword` was just created and the user called `set_passwd` to change the default password. This led to the execution of `_init_state` but the system crashes before `tpm.mode` could be set (see listing 3, line 9). When `ChkPassword` is re-created, it requests its previous state (listing 1, line 6). As `tpm.mode` still read `Clear` (listing 2, line 11), the module will restart from its default settings. As no input was ever used, state-continuity is guaranteed trivially.

Two, the system didn't crash when the user modified the module's default password and now calls `check_passwd` providing `"attempt1"` as password. After `libice0` stores a new cube $C_{\texttt{attempt1}}$ on disk and updates guarded memory, the system crashes while the password is being verified (listing 1, line 23). The module is re-created and execution flow eventually executes `_recovery_step` (listing 5) As only a single cube is available containing the leaked guard from guarded memory (or a successor thereof), only cube $C_{\texttt{attempt1}}$ is considered fresh. After returning the stored input-state tuple in $C_{\texttt{attempt1}}$, `ChkPassword` will restore the `attempts_left` and `password` variables and execution is restarted with input `"attempt1"` (listing 1, line 10).

Three, assume that the previous password was incorrect and the user enters `"attempt2"` for her second attempt. After storing the new cube $C_{\texttt{attempt2}}$ on disk, the system crashes *before* the new (incremented) guard could be written to guarded memory (listing 4, line 4). This is an interesting point of failure as both cubes $C_{\texttt{attempt1}}$ as $C_{\texttt{attempt2}}$ can be considered fresh[6]. However, recovery based on either will preserve state continuity. This is obvious for cube $C_{\texttt{attempt2}}$ as this is the latest cube written to disk. Recovery from $C_{\texttt{attempt1}}$, however will purge any record of the login attempt made using `"attempt2"` as password. This is also safe as it was never used in any valuable computation (instructions after listing 1 line 24 were not executed yet). Hence, an attacker is not able to deduce any valuable information.

3.4.2 `libicen`: State-Cont. Storage for n Modules

By depending on scarce resources such as TPM NVRAM and guarded memory, `libice0` can in practice only provide state-continuous storage to a limited number of modules. `libicen` will alleviate this strain by using a single, unique `ice0` module to store freshness information on behalf of other modules. To safely exchange sensitive information between `libicen` and the `ice0` module, inter-module communication must guarantee endpoint authentication and confidentiality, integrity and freshness of messages. We will state this explicitly by passing a module identifier to `ice0` calls.

Creation of an initial state.

Similarly to `libice0`, an initial state of the module is stored by generating a new guard and cryptographic keys and writing a new cube to disk (see listing 6). Finally the `ice0` module is requested to store the keys and guard.

[6]The recovery step as displayed in listing 5, line 19, only accepts cube $C_{\texttt{attempt1}}$ as fresh. However, an attacker incrementing the guard stored in guarded memory, will trick `libice0` to accept cube $C_{\texttt{attempt2}}$ as being fresh as well.

```
1 void _init_step( State *state ) {
    mod.guard = gen_guard();
3   mod.keys = gen_keys();
    hdd.write( new Cube( mod.guard, mod.keys,
          state ));
5   mod.mode = Activated;
    ice0.store( mod.id, mod.keys, mod.guard);
7 }
```

Listing 6: `libicen`: Initialization of a new module

```
1 void _update_state( State *state ) {
    ++mod.guard;
3   hdd.write( new Cube( mod.guard, mod.keys,
          state ));
    ice0.store( mod.id, mod.guard );
5 }
```

Listing 7: `libicen`: Updating a state

Updating a state.

To update a state, `libicen` writes a new cube to disk, before the updated fresh guard is stored in `ice0` (see listing 7).

Recovering from a crash.

To recover from a crash, the (presumably) fresh cube is read from disk (see listing 8). Next, the keys and guard are requested from the `ice0` module. As the fresh guard is always stored safely in `ice0`, a cube with a correct MAC and that contains the fresh guard, must be fresh. Once the cube's freshness has been validated, `libicen` needs to generate a new guard, create and write a new cube to disk and store the new guard in `ice0` before a new step is taken. Storing the fresh cube with a newly generated guard is vital, even though the fresh guard never leaked. For details we refer the reader to the extended version of this paper [27].

4. IMPLEMENTATION

Given that SGX-enabled systems are not available yet, we implemented[7] ICE on top of Fides [28], a hypervisor-based protected-module architecture using CMOS memory as guarded memory. This setup enables microbenchmarks and detailed analysis of the costs of accessing the TPM chip, writing cubes to disk, performing cryptographic calculations and accessing guarded memory on *real-life* systems.

While Fides provides similar isolation mechanisms as Intel SGX, it cannot guarantee that protected modules, or its own implementation for that matter, do not leave the CPU package in plaintext. Hence, this setup cannot defend against a hardware attacker that is able to directly modify contents of main memory. Similarly, we assume that an attacker cannot modify contents of CMOS memory directly.

While *any* non-volatile memory can serve as an alternative, CMOS memory is an interesting candidate for guarded memory. As it already stores wall-clock time, it is updated every second and it must support a large number of write operations over its entire lifespan. Second, as it does not require a special communication protocol, it can be accessed easily and without much overhead. Being only accessible

[7]Our research prototype is available at `https://distrinet.cs.kuleuven.be/software/sce/`

```
State *_recovery_step() {
  Cube cube = hdd.read();
  ice0.retrieve( mod.id, &mod.keys, &mod.
      guard)
  if ( is_fresh( &cube ) ) {
    State *state = extract(cube.state, tpm.
        keys)
    mod.guard = gen_guard();
    hdd.write( new Cube( mod.guard, mod.
        keys, cube.state ) );
    mod.state = Activated;
    ice0.store(mod.id, mod.keys, mod.guard)
    return state;
  }
  else abort();
}

bool is_fresh( Cube *cube ) {
  return check_mac( cube, mod.keys ) &&
    mod.guard.value == cube.guard.value;
}
```

Listing 8: `libicen`: Recovering from a crash

	`libice0`		`libicen`	
	asm	C	asm	C
ICE	0	372	0	341
SHA-512	0	371	0	371
AES-NI	1,566	176	1,566	176
Total	1,566	919	1,566	888

Table 1: Breakdown of `libice0` and `libicen`.

through direct I/O, it can also be isolated easily by hardware virtualization support; only 21 lines of code (LOC) had to be added to the hypervisor. Another 61 LOCs were required to implement system calls to access CMOS memory from the `ice0` module. This totals the size of the hypervisor to 9,492 LOCs. While Fides at this moment does not support TPM chip accesses, we estimate, based on the Flicker [17] source code[8], that this straightforward effort would require an addition of less than 2,000 LOCs.

To implement `libice0` and `libicen`, we used the polarssl[9] library to calculate SHA-512 hash values and the Intel AES-NI reference implementation to take advantage of AES hardware support. This totals to 2,485 LOCs and 2,454 LOCs for `libice0` and `libicen` respectively (table 1).

5. EVALUATION

5.1 Security Evaluation

Recall that we wish to guarantee both safety and liveness properties. We discuss both aspects separately.

5.1.1 Safety Properties

To ensure that our presented algorithm does guarantee state continuity, we developed a formal proof of correctness. While alternative formalizations (e.g., LS^2 [6]) may have reduced our workload, we formalized our system using rely-guarantee reasoning [12] as it enables explicit reasoning about attack steps. A machine-checked proof was created using the Coq proof assistant. The proof required 118 definitions, 201 lemmas and totals 37,726 lines. The extended version of this paper [27] discusses the proof in more detail.

5.1.2 Liveness Properties

In order to guarantee liveness, `libice0` and `libicen` must always be able to recover from a crash. We discuss how this is accomplished during their different phases.

`libice0`'s liveness properties.

An important distinction can be made based on the value of `tpm.mode`. This value indicates whether the algorithm has been initialized correctly. A crash before this value is set, will result in a re-execution of the initialization step. After setting this value, all crashes will result in the execution of the recovery step. To ensure that the initialization step may succeed eventually, we store the fresh cube on disk, take exclusive access of guarded memory, write the matching guards to guarded and TPM NVRAM memory and store cryptographic keys in the TPM chip *before* setting `tpm.mode`.

After initialization we may update the state or we have to recover the fresh state. In the former case we make sure to first store the cube before we update the content of guarded memory. Recovery of a state is more challenging as we have to modify the guards in both guarded and TPM NVRAM memory. After creating a new cube with the fresh state of the module and storing it on disk, we take exclusive access of guarded memory and write the new guard to it before we update TPM NVRAM memory. This has an important consequence: in case the system crashes during the execution of the recovery step before it is completed, the old guard may have been overwritten. This would prevent the re-execution of the recovery step. Therefore we require that this (public) guard is written to disk *before* the recovery step is called.

`libicen`'s liveness properties.

Ensuring liveness of `libicen`'s algorithm is straightforward as we only have to deal with two non-volatile data objects: cubes and the `ice0` module to store freshness information. For obvious reasons we ensure that new cubes are stored on disk first. `ice0`'s implementation and the `libice0` library it uses, guarantee that its state updates can be considered atomically and they are always retrievable.

5.2 Performance Evaluation

In this section we evaluate the performance of our prototype implementation. To compare the performance impact of a solid state drive (SSD) against a rotating hard drive (HDD), we used two machines with comparable hardware. The first machine, a Dell Latitude E6510, a mid-end consumer laptop, is equipped with an Intel Core i5 560M processor running at 2.67 GHz and 4 GiB of RAM. It is also equipped with a magnetic hard disk (HDD), a Broadcom TPMv1.2 chip and CMOS memory. The second testing laptop is a Dell Latitude E6520, has an Intel Core i5-2520M CPU running at 2.50GHz and is equipped with an SSD.

Hardware Benchmarks.

To better understand the performance cost of ICE compared to TPM operations, we performed 4 benchmarks on the Latitude E6510: read/write accessing TPM NVRAM, extending PCR registers and generating random numbers. To perform these tests, we developed small TPM appli-

[8] `https://sparrow.ece.cmu.edu/group/flicker.html`
[9] `http://polarssl.org/`

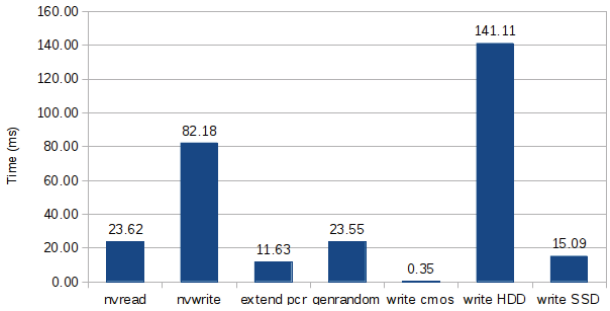


Figure 2: Microbenchmarks of various TPM operations show a significant difference in performance cost of CMOS and disk accesses. Where applicable, 128 bytes were transferred.

cations using the TrouSerS[10] open-source software stack. We also modified the tpm_tis driver to keep timing measurements. Each test was run 100 times and transferred 128 bytes to/from the TPM. Figure 2 displays the median time for each test. All operations take a significant amount of time to complete. Especially writing to TPM NVRAM takes 4x longer than reading from it. Related work shows similar results for TPM chips from other vendors [20].

We also performed a similar benchmark on CMOS memory. We performed 10,000 one-byte write operations and measured the time using the `rdtscp` instruction. Writing to CMOS takes about 3μs/byte, significantly faster than writing to TPM NVRAM. We attribute this difference to the fact that CMOS memory is connected to the SPI-bus [10] and does not require a heavy communication protocol as does the LPC-connected TPM chip.

Finally, we measured the median time of writing 10,000 128 bytes files to both HDD and SSD disks. As Figure 2 shows, accessing the SSD disk is 5.4 times faster than writing to TPM NVRAM. Writing to a magnetic disk is more costly.

Microbenchmarks.

To measure the performance of both `libice0` and `libicen` libraries, we implemented two modules. The first module implements a password verification function and limits the number of attempts that can be made before the user is locked out indefinitely. The benchmark provided this module with 10,000 wrong password guesses and measured the median time per guess. Measurements show (see Table 2) that for a single step only 0.06ms (0.43%) were spent on computation when the module was linked with the `libice0` library. When we used `libicen`'s services, two cubes need to be created and computation time increased to 0.13ms (0.71%). To securely write guards to CMOS memory, 0.33ms were spent (2.17% and 1.82% for `libice0` and `libicen` resp.). This shows a much higher cost to write guards to CMOS compared to calculation time. But most of the time was spent committing cubes to solid state disk (97.40% and 97.47% for `libice0` and `libicen` resp.). `libicen` does *not* spend twice the amount of time writing cubes to disk. Cubes only need to be committed before a guard is incremented. Hence, `libicen`'s cubes can be stored temporarily in memory and transferred to disk together with `ice0`'s new cube *without* modifying the algorithm (see listing 7, lines 3-4), reducing disk access times.

While most TPM chips NVRAM area is limited to 1,280 bytes [20], it could be used to provide (state-continuous) storage to a single module to avoid disk overhead. To show that such a module would still benefit from ICE, we implemented a second benchmark called Noop. It does not perform any computation but only stores a state of 1,280 bytes. As expected given the performance of SHA-512 and Intel's AES hardware support, the increase in computation cost is negligibly with only 0.01ms. As cubes are still smaller than disk sectors, costs of disk accesses are comparable to the Password benchmark. This totals the cost of storing new data in Noop at 15.05ms to 17.65ms for `libice0` and `libicen` resp; significantly faster than 82.18ms to access TPM NVRAM. Finally we performed these tests on the Latitude E6510 which is equipped with a magnetic HDD. As expected, the cost of writing cubes to disk increased significantly and now accounts for 99.63%-99.74%. For both benchmarks `libicen` consistently takes more time writing cubes to disk than `libice0`. We attribute this behavior to the way we implemented its write function: merging `ice0`'s and `libicen`'s cubes takes us 3 `write` system calls before system buffers are flushed.

Expected Impact of Dedicated Hardware.

These benchmarks show that only up to 0.14% of time is spent on computation. With dedicated hardware performance can be increased significantly.

Writing guards to CMOS memory is about 2.4 times more costly than computation and takes up to 0.31% of the time in case of a revolving HDD and up to 2.17% on our SSD testing platform. Hardware support for guarded memory, as described in detail in Section 3.1, would reduce overhead of this operation to almost zero.

But committing cubes to disk forms the real bottleneck, requiring up to 97.47% (for SSD) to 99.74% (for HDD) of the time. Recently Viking Technology [34] and Micron Technology [33] announced that they will ship capacitor-backed RAM to market. Operating similar to guarded memory, these hardware components contain fast, volatile memory that is written to flash memory when power is suddenly lost. Adding these hardware components to our system would eliminate disk access completely.

In summary, benchmarks show that our prototype implementation on commodity hardware already outperforms TPM NVRAM write operations by almost 5 times. Adding dedicated hardware support for guarded memory and capacitor-backed RAM, may even enable state updates 587 times faster than TPM NVRAM accesses!

6. RELATED WORK

With the exception of Parno et al. [20], state-continuous storage has largely been overlooked. Most research prototypes rely on a huge TCB or are vulnerable to crash attacks.

Hardware Modifications.

XOM [15] protects against an attacker that is able to snoop buses and modify memory by encrypting data and code before it is sent to memory. While it makes it significantly more difficult to successfully attack the system, Suh et al. [30] argue correctly that it is vulnerable to a memory replay attack where stale memory pages are returned to

[10] `http://trousers.sourceforge.net/`

	Password		Noop	
SSD (in ms)	`-lice0`	`-licen`	`-lice0`	`-licen`
computation	0.06	0.13	0.07	0.14
writing guard	0.33	0.33	0.33	0.33
writing cubes	14.61	17.42	14.65	17.19
total	15.00	17.87	15.05	17.65

HDD (in ms)	`-lice0`	`-licen`	`-lice0`	`-licen`
computation	0.06	0.12	0.07	0.13
writing guard	0.35	0.35	0.35	0.35
writing cubes	112.80	183.23	111.54	183.83
total	113.21	183.71	111.96	184.31

Table 2: Microbenchmarks for `libice0` and `libicen`

the processor. Their Aegis architecture mitigates this replay attack by storing hash trees of memory pages in a secure location. When a memory page is loaded into the processor's cache, its freshness is checked by recalculating and comparing the hash values. Subsequent research results also defend against replay attacks [4, 11, 37].

Memory replay attacks differ from rollback attacks in that memory contents is replayed *while* the system is up and running. This enables much easier security measures.

Schellekens et al. [24] propose an embedded-systems architecture to store a trusted module's persistent state in invasive-attack-resistent, non-volatile memory. Their solution implements a light-weight authenticated channel between the trusted module and non-volatile memory. Freshness of the stored data is guaranteed per read/write instruction and based on a monotonic counter. As their approach assumes that write instructions to non-volatile memory and increments of the monotonic counter are atomic, unexpected loss of power enables a rollback attack. We believe that their approach can be fixed by keeping a log of instructions in secure non-volatile memory that need to be completed in case power suddenly fails. On higher-end systems however, only the TPM NVRAM can be used for such purposes and their approach would lead to significant performance overhead. ICE, in contrast, is not affected by TPM performance.

Research Systems Isolating Persistent Storage.

Many architectures rely on a large TCB that includes isolation of persistent storage [7, 25, 31]. In such cases protection against rollback attacks are trivial: modules/programs can overwrite their state on disk. In practice however, software vulnerabilities in their TCB may be exploited and state-continuity support is hard to guarantee. These systems are also not able to defend against disk clone attacks. In contrast, ICE provides strong guarantees while only relying on a very limited TCB.

Protected-Module Architectures.

Many security architectures with minimal TCB have been proposed that only providing strong module isolation guarantees [3, 11, 16, 17, 23, 28]. Persistent storage can only be accessed via services provided by the untrusted operating system. None of them address the issue of state continuity.

Many of these systems can be adapted to use the state-continuity approach presented by Parno et al. [20]. This seminal work called Memoir, is to the best of our knowledge the first and only work that addresses the issue of state continuity in protected-module architectures. Based on Flicker [17], Memoir uses TPM NVRAM to store freshness information upon every state update. This significantly limits the applicability of their solution as NVRAM is slow and only required to support up to 100K writes. The authors acknowledge this constrained and propose two solutions: (1) adding capacitor-backed RAM to the TPM chip and (2) Memoir-Opt, an alternative approach that stores freshness information in (volatile) TPM PCR registers that are written to NVRAM when power is lost unexpectedly. Both solutions rely on an uninterruptible power source to safely store freshness when power suddenly fails. Failure in this mechanism can lead to a rollback attack. ICE, in contrast, is a passive state-continuity system that does not rely on an uninterruptible power source to guarantee security; detaching the capacitor would only prevent stateful modules from recovering their state but states could not be rolled back. Moreover, in ICE the speed of updates to state-continuous modules is only limited by the processor and (untrusted) non-volatile memory, not by the TPM chip.

Special-Purpose Applications.

Chun et al. proposed the creation of append-only memory [5] to prevent that nodes in a distributed system can make different statements to different nodes. An implementation with a minimal TCB was left as future work.

Levin et al. propose TrInc [14], a specialized system to attest successive monotonic counters, to achieve similar results. TrInc assumes a dedicated device that is able to locally store attestation requests of monotonic counters. After power was suddenly lost, clients can request the last signed attestations. This approach is similar to solutions were disk space is isolated, but incurs only a limited TCB. ICE provides a more generic, low-overhead alternative with only limited hardware modifications.

More recently Kotla et al. proposed a system [13] that allows offline data access while guaranteeing that (1) a user cannot deny offline accesses without failing an audit and (2) after proving that a user did not access the data, it cannot be accessed in the future. While their solution is interesting and does not require any software to be trusted, it only solves state-continuity in this specific setting.

7. CONCLUSION

Providing support for state continuity is challenging as including non-volatile memory on-chip requires modification of fabrication processes. But off-chip storage of freshness information can be slow (e.g. TPM NVRAM) or vulnerable to attack. We presented ICE, a state-continuous system and algorithm with two important properties: (1) only at boot time is the (slow) TPM chip accessed. State updates after the system booted only require updates to dedicated registers backed off-chip by a capacitor and non-volatile memory. (2) ICE is a passive security measure. An attacker interrupting the main power supply or any other source of power, cannot break state-continuity. We believe that the importance of ICE lies in the fact that it shows that with only limited and cheap hardware support, it enables the development of software-only implementations of trusted computing primitives. This presents an interesting direction for future versions or revisions of hardware security modules (e.g., the TPM) and may provide an interesting approach to increase security in low-end, resource-constrained applications.

Acknowledgments

The authors thank all reviewers and proofreaders of the paper for their useful comments. We also thank Frédéric Vogels and Dominique Devriese for their help with Coq.

This work has been supported in part by the Intel Lab's University Research Office. This research is also partially funded by the Research Fund KU Leuven, and by the EU FP7 project NESSoS. Raoul Strackx holds a PhD grant from the Agency for Innovation by Science and Technology in Flanders (IWT).

8. REFERENCES

[1] P. Agten, R. Strackx, B. Jacobs, and F. Piessens. Secure compilation to modern processors. In *CSF'12*.

[2] I. Anati, S. Gueron, S. Johnson, and V. Scarlata. Innovative technology for CPU based attestation and sealing. In *HASP'13*.

[3] A. Azab, P. Ning, and X. Zhang. SICE: a hardware-level strongly isolated computing environment for x86 multi-core platforms. In *CCS'11*.

[4] D. Champagne and R. Lee. Scalable architectural support for trusted software. In *HPCA'10*.

[5] B.-G. Chun, P. Maniatis, S. Shenker, and J. Kubiatowicz. Attested append-only memory: Making adversaries stick to their word. In *OSR'07*.

[6] A. Datta, J. Franklin, D. Garg, and D. Kaynar. A logic of secure systems and its application to trusted computing. In *30th IEEE Symposium on Security and Privacy*, pages 221–236. IEEE, 2009.

[7] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh. Terra: A virtual machine-based platform for trusted computing. In *OSR'03*.

[8] J. Halderman, S. Schoen, N. Heninger, W. Clarkson, W. Paul, J. Calandrino, A. Feldman, J. Appelbaum, and E. Felten. Lest we remember: Cold boot attacks on encryption keys. In *USENIX'08*.

[9] M. Hoekstra, R. Lal, P. Pappachan, V. Phegade, and J. Del Cuvillo. Using innovative instructions to create trustworthy software solutions. In *HASP'13*.

[10] Intel Corporation. *Intel 6 Series Chipset and Intel C200 Series Chipset*, 2011.

[11] Intel Corporation. *Software Guard Extensions Programming Reference*, 2013.

[12] C. Jones. Tentative steps toward a development method for interfering programs.

[13] R. Kotla, T. Rodeheffer, I. Roy, P. Stuedi, and B. Wester. Pasture: secure offline data access using commodity trusted hardware. In *OSDI'12*.

[14] D. Levin, J. R. Douceur, J. R. Lorch, and T. Moscibroda. Trinc: Small trusted hardware for large distributed systems. In *NSDI'09*.

[15] D. Lie, T. Chandramohan, M. Mark, L. Patrick, B. Dan, M. John, and H. Mark. Architectural support for copy and tamper resistant software. In *ASPLOS'00*.

[16] J. M. McCune, Y. Li, N. Qu, Z. Zhou, A. Datta, V. Gligor, and A. Perrig. TrustVisor: Efficient TCB reduction and attestation. In *S&P'10*.

[17] J. M. McCune, B. Parno, A. Perrig, M. K. Reiter, and H. Isozaki. Flicker: An execution infrastructure for TCB minimization. In *EuroSys'08*.

[18] F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue, and U. R. Savagaonkar. Innovative instructions and software model for isolated execution. In *HASP'13*.

[19] J. Noorman, P. Agten, W. Daniels, R. Strackx, A. V. Herrewege, C. Huygens, B. Preneel, I. Verbauwhede, and F. Piessens. Sancus: Low-cost trustworthy extensible networked devices with a zero-software trusted computing base. In *Usenix'13*.

[20] B. Parno, J. R. Lorch, J. R. Douceur, J. Mickens, and J. M. McCune. Memoir: Practical state continuity for protected modules. In *S&P'11*.

[21] M. Patrignani, P. Agten, R. Strackx, B. Jacobs, D. Clarke, and F. Piessens. Secure compilation to protected module architectures. In *Accepted for publication in ACM Transactions on Programming Languages and Systems (TOPLAS)*.

[22] M. Patrignani, D. Clarke, and F. Piessens. Secure Compilation of Object-Oriented Components to Protected Module Architectures. In *APLAS'13*.

[23] D. P. Sahita R, Warrier U. Protecting Critical Applications on Mobile Platforms. Intel.

[24] D. Schellekens, P. Tuyls, and B. Preneel. Embedded trusted computing with authenticated non-volatile memory. In *TRUST'08*.

[25] L. Singaravelu, C. Pu, H. Härtig, and C. Helmuth. Reducing TCB complexity for security-sensitive applications: three case studies. In *EuroSys '06*.

[26] E. R. Sparks. A security assessment of trusted platform modules. Technical report.

[27] R. Strackx, B. Jacobs, and F. Piessens. ICE: A passive, high-speed, state-continuity scheme (extended version). Technical report, KU Leuven, Sept. 2014.

[28] R. Strackx and F. Piessens. Fides: Selectively hardening software application components against kernel-level or process-level malware. In *CCS'12*.

[29] R. Strackx, F. Piessens, and B. Preneel. Efficient Isolation of Trusted Subsystems in Embedded Systems. In *SecureComm'10*.

[30] G. E. Suh, D. Clarke, B. Gassend, M. van Dijk, and S. Devadas. AEGIS: architecture for tamper-evident and tamper-resistant processing. In *ICS'03*.

[31] R. Ta-Min, L. Litty, and D. Lie. Splitting interfaces: Making trust between applications and operating systems configurable. In *OSDI'06*.

[32] C. Tarnovsky. Deconstructing a "secure" processor. In *Black Hat'10*.

[33] M. Technology. Hybrid memory - bridging the gap between DRAM speed and NAND nonvolatility.

[34] V. Technology. NV-DIMM: Achieving greater ROI from SSDs. Technical report.

[35] Trusted Computing Group. Design Principles Specification Version 1.2. 2011.

[36] 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 *S&P'13*.

[37] P. Williams and R. Boivie. CPU support for secure executables. In *TRUST'11*.

[38] J. Winter and K. Dietrich. A hijacker's guide to the LPC bus. In *EuroPKI'11*.

Interrupt-oriented Bugdoor Programming: A Minimalist Approach to Bugdooring Embedded Systems Firmware

Samuel Junjie Tan
Google, Inc.
samueltan@gmail.com

Sergey Bratus
Dartmouth College
sergey@cs.dartmouth.edu

Travis Goodspeed
Straw Hat
travis@radiantmachines.com

ABSTRACT

We demonstrate a simple set of interrupt-related vulnerability primitives that, despite being apparently innocuous, give attackers full control of a microcontroller platform. We then present a novel, minimalist approach to constructing deniable bugdoors for microcontroller firmware, and contrast this approach with the current focus of exploitation research on demonstrations of maximum computational power that malicious computation can achieve. Since the introduction of Return-oriented programming, an ever-increasing number of targets have been demonstrated to unintentionally yield Turing-complete computation environments to attackers controlling the target's various input channels, under ever more restrictive sets of limitations. Yet although modern OS defensive measures indeed require complex computations to bypass, this focus on maximum expressiveness of exploit programming models leads researchers to overlook other research directions for platforms that lack strong defensive measure but occur in mission-critical systems, namely, microcontrollers. In these systems, common exploiter goals such as sensitive code and data exfiltration or arbitrary code execution do not typically require complex computation; instead, a minimal computation is preferred and a simple set of vulnerability primitives typically suffices. We discuss examples of vulnerabilities and the new kinds of tools needed to avoid them in future firmware.

Categories and Subject Descriptors

K.6 [**Management of Computing and Information Systems**]: Miscellaneous; K.6.5 [**Management of Computing and Information Systems**]: Security and ProtectionUnauthorized access (e.g., hacking, phreaking)

General Terms

Security

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.
ACSAC '14, December 08 - 12 2014, New Orleans, LA, USA
Copyright 2014 ACM 978-1-4503-3005-3/14/12...$15.00
http://dx.doi.org/10.1145/2664243.2664268

Keywords

ACSAC Proceedings, Security, Hacking, Microprocessor exploitation

1. INTRODUCTION

This paper examines what is arguably the most constrained "weird"[1] programming model to date: one using side-effects of interrupts. This model has, to the best of our knowledge, been overlooked. Although this model is weaker than any previous exploit programming models, it is significant because of the increasing ubiquity of microcontrollers and the ease with which this novel class of innocent-looking "bug doors" can be planted into firmware.

In this paper, we demonstrate this model on synthetic vulnerabilities, and provide prototype tools to help locate similar vulnerabilities on firmware in the wild, which we intend for future work. Moreover, we describe a proof-of-concept "bugdoor" implemented by making the interrupt handler of an MSP430 TinyOS-based firmware application re-entrant. This "bugdoor" is controlled solely by means of well-timed, attacker-triggered interrupts.

Bug doors.

Unlike a traditional backdoor, where specialized code is left in the program to surreptitiously control it, a bugdoor does not leave undeniably malicious code in the target program; it requires the attacker to supply specific inputs to the target to insert the malicious program (e.g. shellcode) and so perform unintended computation. The former type of vulnerability is easier to detect and trace to the source, as was the case with Alcatel's network switch operating system `telnet` backdoor in 2002 [10]. The latter is harder to discover and to definitively frame as deliberate exploitation.

With the building blocks featured in our synthetic vulnerabilities situated in firmware, an attacker would have a deniable bugdoor that involves no common forms of memory corruption and would therefore likely be missed by vulnerability scanning tools. Our prototype tools, on the other hand, show a basic set of tell-tale conditions associated with such hypothetical bugdoors.

"Two gadgets is all you need".

In the following two sections we put our programming model in historical perspective of exploitation programming. We then discuss the microcontroller threat model, the tar-

[1] In the sense of "weird machines", unexpected programming modes of trusted systems that we discuss later.

geted processor MSP430, and our "interrupt-oriented programming" model itself (a pun on ROP), starting with its primitives and continuing with the bug door that realizes it. We then describe our (inconclusive) attempts at finding such code in the wild, and wrap up with ideas for follow-on work.

2. MOTIVATION AND HISTORY: WHITHER EXPLOITATION?

2.1 Exploitation and its programming models

Research in exploitation techniques traditionally focused on feature-rich targets such as servers, desktops, and smart phones, where exploitation co-evolved with protective measures. Indeed, techniques such as return-to-libc [27], return-to-PLT [18], which generalized the reuse of code chunks by chaining stack frames pointing to them [20, 14], emerged in response to protective measures such as Openwall and PaX [28, 29] and co-evolved with such measures (e.g., [18, 8, 17]).

Interestingly, the more significant the barriers posed to attackers, the more expressive and general exploit programming models became. In 2000 [20] Gerardo Richarte observed that chunk chaining techniques could apparently encode "any program"; in 2007, Shacham Hovav [24] proved that x86 stack-based libc end-of-function block chaining was a Turing-complete programming model, at the same time coining the term *Return-Oriented Programming* (ROP) for such models. Since then, completeness has been shown to exist with less and less resources, such as with only certain classes of instructions [6], only constant-length instructions on platforms such as RISC [5]. Completeness has also been shown to exist when payloads are restricted to certain character sets [23]—even to just English dictionary words [15]—and when the payloads had to work on several ISAs at once [9].

This co-evolution highlighted the nature of exploits as *programs written for and assisted by certain combinations of features and bugs in the targets*—so-called primitives [21, 22]—used in a manner similar to assembly instructions (e.g., [13]). These unusual, emergent target machines and programming modes for exploit-programs became known as "weird machines" [4, 7, 2]. Typically, weird machines' existence was a surprise for the target's developers, but deliberate creation of them to provide a deliberate "debug door" or simply a "bug door" is also possible.

2.2 The distraction of Turing-completeness

Most weird machines were proved to be Turing-complete, following in the footsteps of [24]—partly because it pleases a certain kind of computer scientist to have a theorem like a cherry on top of a paper, and partly because other kinds of computer scientists saw offensive papers merely describing a novel exploit technique as incremental and unscientific. Arguably, the original proof of libc-based ROP completeness was necessary to change the academic view of exploits as limited and ad-hoc; it certainly made the great step forward of replacing the outmoded threat model of "malicious code" (and the unhelpful focus on detecting such code) with the more relevant model of malicious computation inherent in most general-purpose systems.

However, the computation performed by an exploit only needs to be powerful enough to achieve the exploiter's actual goals; there are no points awarded for additional expressive power, nor for extra complexity. The golden rule of exploitation is that when a thing can be reliably accomplished with minimal means, it should be accomplished with just such means. However, practical minimality has been overshadowed by pursuing techniques with maximum computational power. This disconnect is the starkest for *microcontrollers*, upon which our trusted and mission-critical systems continually increase their reliance.

2.3 Microcontroller exploitation

It was necessity that drove practical evolution of ROP and other techniques for feature-rich targets towards more expressive programming models. Essentially, protective measures forced exploits to emulate the OS' native linkers (cf. [3] on the "Bring Your Own Linker" pattern), which perform complex computations (e.g., GNU/Linux dynamic linker is a Turing-complete platform for its metadata considered as a program [25]). Accordingly, a rich set of primitives adding up to a computation model in which (nearly) arbitrary algorithms could be expressed helped in practical exploitation of such targets. In other words, computational power (up to Turing completeness) helped even when it wasn't strictly required.

For microcontrollers, the exploiters' situation is quite different. In absence of countermeasures, microcontroller vulnerability primitives need not provide a computationally rich environment; they just need to enable typical goals such as *exfiltration of firmware code*, *exfiltration of sensitive data such as cryptographic keys or configuration parameters* or *code execution in firmware contexts*. Exploit computation in microcontrollers need not be complex, and the primitives it requires to succeed may in fact be limited to a single write to a specific memory address or a single control flow redirection.

Microcontroller exploitation thus poses a different class of research problems, which, to the best of our knowledge, has been overlooked. Namely: what are the minimal combinations of primitives that still serve the attacker's goals (rather than, as in feature-rich targets, allowing the attacker to encode arbitrary algorithms)?

We can pose this question more precisely in the case of *bugdoors*, vulnerabilities deliberately introduced into a system to provide a deniable backdoor. What is a minimal patch that would provide such a backdoor and would be least likely to be spotted by bug-finding tools looking for memory corruption bugs?

3. RELATED WORK

Exploitation of computationally rich targets has a rich history, which was outlined above. The overall direction of this history has been doing more with less—essentially, keeping full control of the target and (Turing-)complete expressiveness of exploit programs with an ever-shrinking set of primitives. A good history sketch of primitives and attacks can be found in [16]. It turned out that many kinds of special-purpose data could also drive Turing-complete computations on the respective standard code that interpreted such data (e.g., [19, 25, 1]).

At the same time, for microcontrollers, the meaning of exploitation was different, reflecting the difference in threat models. In servers, etc., exploitation's typical goal is full control of the target ("root shell", for short); in microcon-

trollers, it is the exfiltration of firmware or data. For most microcontroller deployments, firmware is the important thing; once the attacker has the firmware, the rest of his goals are easily accomplished. This may or may not be due to firmware's general buggyness and lack of higher-grade protection such as NX memory or randomization; sometimes the firmware even contains passwords or private keys.

Goodspeed and Francillon demonstrated a good example of advanced microcontroller attacks in [11]. Their "half-blind" ROP attack circumvents protections of the MSP430, using only knowledge of the MSP430 Bootstrap Loader (BSL) code. The BSL is a small program present in ROM or protected flash of the MSP430 that allows developers to program the memory of the MSP430, typically for installing firmware updates. BSL operations that read memory are password protected in order to prevent unauthorized readout of code or data. In this attack, the entry point of a specific ROP gadget is guessed at random with a 1% success rate, and a buffer overflow is found by fuzzing. Given both this ROP gadget entry point and the buffer overflow vulnerability, a ROP payload can be deployed to circumvent protections in the BSL.

4. THREAT MODEL FOR MICROCONTROLLERS

The security architecture of microcontroller was designed with a specific threat model in mind. The secrecy of microcontroller firmware is crucial to prevent the production of counterfeit products with identical firmwares and theft of service through the alteration of firmware (e.g. changing power usage records on a "smart meter" to reduce electricity bills). To preserve the secrecy of firmware data while still allowing devices to receive firmware updates, microcontroller manufacturers typically use a bootloader and a combination of "fuses"—special write-once registers that contain permissions—to enforce access control on read/write operations on ROM.

An attacker targeting microcontrollers therefore aims to exfiltrate data, such as firmware, from the chip package. We assume that the attacker controls the processor, i.e., can slow down the clock, observe General Purpose Input/Output (GPIO), send signals to any pins, etc. However, we do not assume that the attacker possesses the capability to carry out sophisticated hardware attacks such as the one described in [12], where the integrated circuit board itself is penetrated to expose the device's circuits and allow data exfiltration through microprobing.

Our synthetic attacks assume that the attacker possesses some knowledge of the behavior of firmware, particularly that of the interrupt handlers. This scenario applies to common cases where the attacker has access to source code of older or similar versions of the target device firmware, but does not have access to the source code of the firmware running on the device itself. Given the fact that interrupt handlers are usually boilerplate code that are rarely changed between firmware versions, such a scenario is very feasible.

Our "bugdoor" attack assumes that the attacker can view and modify firmware source code, but not in a manner that is obviously malicious. This scenario applies to cases where the target microcontroller runs open source firmware, such as TinyOS, that an attacker could contribute his own modifications to, subject to peer review.

5. ROP, MEET "IOP"

The classic way to exploit a computing system is via its inputs. Any processed input is really a program; an exploit is merely a program with unexpected effects, format, composition modes, and, sometimes, unexpected input and output channels.

For our *Interrupt-oriented Programming* trick (IOP, a pun on ROP), we will use the fact that, other than data, a processor also takes **interrupts** as inputs. More specifically, the processor can be induced to execute interrupt handler code by events that can often be triggered externally by an attacker (e.g., sending an edge to a specific pin). In the same way that an a vulnerable target can be "programmed" by an attacker with a carefully crafted stack frame passed to it as input, we "program" a processor with a series of well-timed interrupts.

If our timings are precise enough to interrupt the processor while it is executing specific instructions, we can treat consecutive chunks of instructions starting at interrupt vector entry points and ending at another interrupt as our "gadgets", except that the starts of these "gadgets" are fixed, while the endings depend on our ability to deliver the interrupting interrupt at just the right instruction.[2]

Achieving such instruction-level timing precision on state-of-the-art server processors is clearly out of reach, since their high clock speeds make timing interrupts infeasible. However, this technique is feasible for microcontrollers that have far lower clock speeds and support downclocked, low-power processor states. For example, this technique will be applicable to the various, ubiquitous microcontroller-based sensors and "smart" devices, such as personal medical devices, industrial control systems, and "smart homes", on which we are getting increasingly reliant today.

Of course, interesting instructions should be found close enough to the beginning of interrupt vector chunks in the targeted firmware to be of use. It turns out that, with the minimalist primitives we discuss in section 7, we can plant an effective "bug door" there (which we describe in section 10).

Thus our objective is to explore programming with timed sequences of interrupt-triggering signals that would result in the execution of unintended, attacker-controlled computation. Formally speaking, our IOP programs are sequence of pairs (t_i, s_j), where $t_0, t_1, ..., t_{n-1}, t_n$ is the sequence of timings corresponding to CPU clock cycles, and $s_0, s_1, ..., s_{k-1}, s_k$ is the set of interrupt-triggering signals. t_0 is the time before the execution of the first instruction in the target program (clearly, timings t_i only make sense relative to a moment when the processor is in known state, such as its initial reset). For each pair pairs (t_i, s_j), the interrupt that signal s_j triggers must enabled at time t_i (see the discussion of interrupt-enabling bits below).

For this study, we looked at several MSP430-powered devices: EZ430U, EZ430URF, FET430UIF, MSP430FET, and GoodFET. We will describe this processor now, before we launch into IOP primitives.

6. TARGET PROCESSOR: TEXAS INSTRUMENTS MSP430

[2]In other words, our gadgets are almost the exact opposite of ROP ones; there needs to be nothing special about the instruction on which we break out of the gadget, no control flow semantics of RET or JMP.

The Texas Instruments (TI) MSP430 is a 16-bit ultra-low power microcontroller. The native MSP430 CPU implements a Reduced Instruction Set Computer (RISC) architecture with 27 instructions, 7 addressing modes, and 16-bit addressing. The extended MSP430X architecture supports 20-bit addressing, allowing it to directly address a 1-MB address range without paging. TI MSP430 microcontrollers draw low amounts of electric current in idle mode, and support several low-power modes. The MSP430 CPU can be downclocked to further reduce power consumption. More details about the features of the MSP430 family of microcontrollers can be found in [30].

The TI MSP430's low cost, low power consumption and power saving features has made it a popular choice for use in low power embedded devices, such as the Shimmer Wireless Sensor Platform [26] and the Bodymedia FITTM wearable medical devices (e.g., [34]).

For study, we specifically examined the MSP430F2618. The MSP430F2618 uses the MSP430X architecture, and is the embedded processor used in the GoodFET42, an open source JTAG adapter developed by Travis Goodspeed.

6.1 How interrupts work in the MSP430

There are two types of interrupts in the TI MSP430: maskable interrupts and non-maskable interrupts (NMIs).

Maskable interrupts are caused by peripherals with interrupt capabilities. Maskable interrupts can either be disabled individually by a per-peripheral interrupt enable bit, or all maskable interrupts can be disabled by the General Interrupt Enable (GIE) bit in the Status Register (SR). The SR is cleared when a maskable interrupt is received, thus clearing the GIE bit and disabling any further interrupts. Maskable interrupts can be nested only if the GIE bit is set inside of an interrupt handler.

NMIs cannot be disabled by the GIE bit. Instead, they are enabled by individual interrupt enable bits. When an NMI is accepted, all individual interrupt enable bits are reset, disabling any further NMIs. Software must manually set NMI enable bits to re-enable NMIs.

NMIs can only be generated from three sources: an edge on the RST/NMI (reset) pin, an oscillator fault, or an access violation to flash memory.

When an interrupt occurs, the program counter (PC) and SR are pushed onto the stack, interrupts are disabled, and the contents of the interrupt vector of the interrupt with the highest priority are loaded into PC so control flow is transferred to the interrupt handler. This takes 5 or 6 CPU cycles in the MSP430X and MSP430 CPUs respectively.

Interrupt handlers terminate with the `reti` instruction, which pops SR and PC off the stack to restore program state and execution context to the point where the interrupt was triggered. This takes 3 or 5 CPU cycles in the MSP430X and MSP430 CPUs respectively ([30]).

7. IOP PRIMITIVES

For any series of interrupts to affect the processor as we described above, there must exist some interdependency of state between the interrupt handlers' code, and some accumulation of state in interrupt handlers. We call such sequences of instructions *IOP primitives*. These primitives are "pre-vulns", in that they may, in certain combinations, lead to a vulnerability exploitable with IOP; however, by themselves they may well be innocuous and never triggered in common firmware execution patterns.

The following is a non-exhaustive list of IOP primitives:

- **State accumulation.** An instruction sequence that writes to a register/memory location without first setting it or restoring its original value afterwards. Such an instruction sequence would accumulate state in this register/memory location on each invocation of the interrupt handler.
- **Memory write.** An instruction sequence that writes an immediate value, register value, or value in memory to a register/memory location without overwriting the written value afterward. This could be used to perform arbitrary writes.
- **Arithmetic.** An instruction sequence that performs arithmetic on two values from registers or memory. Combined with a state accumulation and memory write primitives, such a primitive would allow arithmetic to be performed on arbitrary values stored in registers/memory and accumulated for later use.
- **Stack growing.** An instruction sequence that grows the stack (i.e., writes to memory pointed to by SP and decrements SP). This could be used to overwrite memory and to instrument return values on the stack. If this instruction could be invoked an arbitrary number of times (e.g. through interrupt nesting), this would be an *unbounded* primitive. Otherwise, it is *bounded*.
- **Stack alignment.** An instruction sequence that grows by a non-standard number of bytes. This could be used with a stack growing primitive to grow the stack to a word-aligned address. For example, if we had an unbounded stack growing primitive that grew the stack by 4 bytes (2 words) each time and a bounded stack alignment primitive that grew the stack by 2 bytes, invoking the stack alignment primitive after invoking the stack growing primitive a certain number of times would allow the attacker to decrement SP to any word-aligned address lower than the current SP.

Unlike ROP gadgets, which produce the same enduring effects whenever they are used, IOP primitives behave differently depending on the context they are invoked in, and produce effects that are sometimes ephemeral. For example, an interrupt handler containing a state accumulating instruction sequence and ending with an state reset instruction can only be used as a state-accumulation primitive if the accumulated state is used in some way before it is reset. These constraints make useful IOP primitives difficult to identify.

8. TOOLS AND SETUP

8.1 Hardware

In order to deliver the timed-signal payload to the target TI MSP430F2618, we needed to be able to observe the state of the microcontroller and send interrupt-triggering signals to its pins. To achieve that, we connected two other microcontrollers to the target device—one to communicate with the hardware debugging interface of the target MSP430F2618, and the other to send signals to the pins of that target. We

used a hardware debugging interface in our laboratory setup to ensure that our attack was behaving correctly. In an actual IOP attack, the attacker would not have full debugging access to the microcontroller, only the ability to observe some of the microcontroller's state and send signals to it.

Our experimental setup can be found in Figure 1.

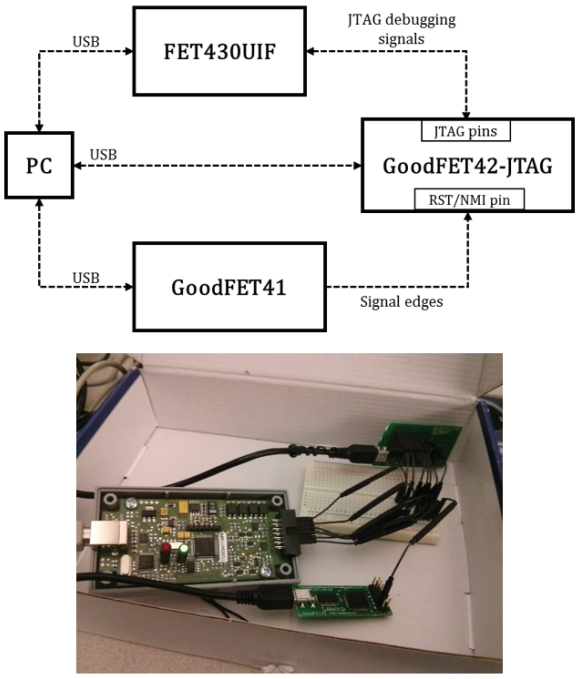


Figure 1: Laboratory setup

8.2 Software

To debug the target MSP430F2618 on the GoodFET42-JTAG, we used `mspdebug` version 0.22, a free debugger for MSP430 microcontrollers. We also extended the MSP430 simulator module in `mspdebug` to perform static analysis of MSP430 firmware.

To control and flash the GoodFET42-JTAG and GoodFET41, we utilized the GoodFET firmware and client software. We also modified the GoodFET firmware to enable the GoodFET41 to send falling edges from its GPIO pin to the RST/NMI pin of the GoodFET42-JTAG.

To disassemble target firmware binaries and perform static analysis on them, we used the IDA Pro interactive disassembler. We also wrote IDAPython scripts for IDA Pro to automate the static analysis of firmware binaries.

9. SPECIFIC EXPLOITS

In this section, we describe two IOP exploits (a.k.a. IOP programs) that utilize the primitive types described in Section 7. The targets of these attacks are simple MSP430 firmware images that we created specifically to allow these exploits to work; we later show how these can be combined into an actual "bug door" in common, working firmware. In all three examples, we use the LED on the GoodFET42-JTAG as a simple output mechanism to indicate the success of the exploit.

9.1 State accumulation exploit

An interrupt handler that predictably and incrementally modifies CPU state without saving or restoring it could be used to accumulate state. If the target program behavior can be influenced by this state, timed invocations of the interrupt handler could be used to control the program.

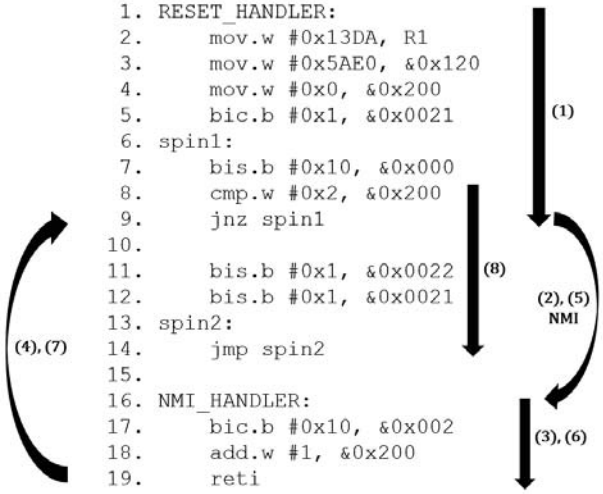


Figure 2: State accumulation exploit

The assembly source code of a simple firmware that contains such a vulnerability can be found in Figure 2. Numbered arrows represent actual instruction execution flow through the code, as caused by our crafted interrupts.

This IOP exploit uses a state accumulation primitive (line 18) in the NMI interrupt handler to cause the firmware to break out of an infinite loop (lines 6-9) and switch on its LED (lines 11-12). The IOP exploit takes place as follows:

(1) The SP (R1) is initialized at line 1, and the microcontroller is configured trigger NMI interrupts upon receiving a falling edge on the RST/NMI pin at line 2. The memory location that will be used to accumulate state, `0x200`, is cleared at line 4. The firmware then spins in lines 6-9, waiting for `0x2` to be accumulated in `0x200`.

(2) An NMI is triggered, which invokes the NMI handler at line 16.

(3) The NMI handler clears the NMI interrupt flag at line 17 to prevent a re- invocation of the NMI handler, and then accumulates `0x1` in `0x200`.

(4) The processor returns from the NMI interrupt to the spin loop. NMI is re-enabled at line 7 by setting the NMI interrupt bit. The firmware spins again, since `0x200` contains the value `0x1`

(5) An NMI is triggered, which invokes the NMI handler at line 16.

(6) NMI handler is invoked, accumulating another `0x1` in `0x200`.

(7) The processor returns from the NMI interrupt to the spin loop.

(8) This time, `0x200` contains the value `0x2`, so the firmware breaks out of the first spin loop, turns on the LED at lines 11 and 12, and spins at the second spin loop from lines 13-14

This exploit was designed to be carried out without any specific interrupt timings required. Simply triggering two NMIs would cause the GoodFET42-JTAG to turn on its LED.

9.2 Loop execution exploit

Besides containing state accumulation primitives, an interrupt handler may contain other useful primitives, such as one that writes to a useful memory address. Proper timing and nesting of interrupts could possibly allow both state accumulation and the writing of the accumulated value to other locations in memory, to be used in combination to affect the target.

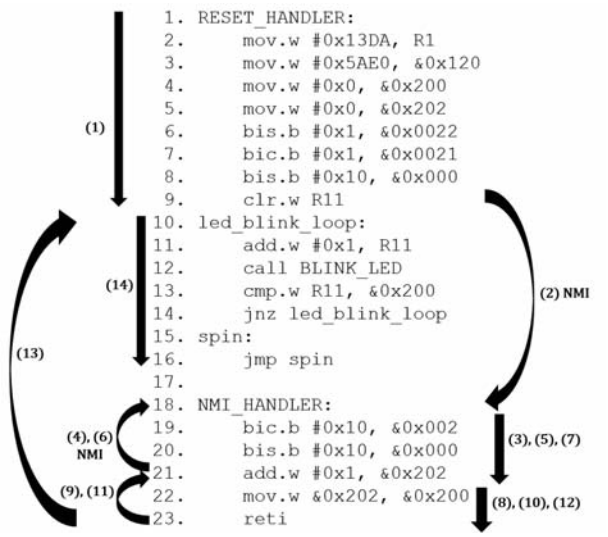


Figure 3: Loop execution exploit

The assembly source code of a simple firmware that contains such a primitive can be found in Figure 3. The interrupt handler in this firmware contains both a state accumulation primitive (line 21) and a memory write primitive (line 22). Using a series of well timed, nested interrupts, this IOP program first accumulates state, then writes it to a destination memory location that controls the number of times the firmware blinks its LEDs. The IOP exploit takes place as follows:

(1) The SP (R1) is initialized at line 1, and the microcontroller is configured trigger NMI interrupts upon receiving a falling edge on the RST/NMI pin at line 2. The memory locations that will be used for state accumulation and the final memory write, `0x200` and `0x202` are cleared in lines 4 and 5. The LED is turned off (if it is already on) from lines 6-7, and the NMI interrupt is enabled by setting the NMI interrupt enable bit at line 8. Finally, the register used as a counter for the LED blinking loop from lines 10-14 is cleared.

(2) An NMI is triggered, which invokes the NMI handler at line 18.

(3) The NMI handler clears the NMI flag at line 19, and re-enables NMIs at line 20 to allow for NMI nesting. It then accumulates `0x1` in `0x202` at line 21.

(4) Another NMI is triggered, which invokes the NMI handler at line 18.

(5) Step 3 is repeated.

(6) Another NMI is triggered, which invokes the NMI handler at line 18.

(7) Step 3 is repeated.

(8) The state accumulated in `0x202` (a value of `0x3`) is written to `0x200`.

(9) The firmware returns to line 21, where the nested NMI was triggered.

(10) Step 8 is repeated. Repeating the memory write is essentially a `nop`, since neither the source or destination has changed.

(11) Step 9 is repeated.

(12) Step 8 is repeated. Repeating the memory write is essentially a `nop`, since neither the source or destination has changed.

(13) The firmware returns to line 10, where the first NMI was triggered.

(14) The LED blinks as many times as the value stored in `0x200` (i.e. 3 times). The firmware then spins from lines 15-16.

This IOP program requires precise timing of nested NMIs to work. Repeating steps 3-4 an arbitrary number of times would allow the attacker to cause the GoodFET42-JTAG to blink its LED an arbitrary number of times.

9.3 Stack growing exploit

All MSP430 interrupt handlers are essentially stack-growing primitives. The invocation of an interrupt handler itself, as described earlier, causes the stack to grow by four bytes since the SR and PC are pushed onto the stack. If an interrupt could be nested (i.e., invoked while still executing its handler), the interrupt handler itself could be seen as a state accumulation tool that can decrement SP by 4 an arbitrary number of times.

The MSP430 has a linear memory layout, and the stack uses RAM, which precedes ROM in a higher address range (see Figure 4). If the stack was grown deep enough (i.e. towards lower addresses) by repeated invocations of the aforementioned interrupt handler until the stack reaches ROM, the `push` instructions executed by the microcontroller will decrement SP without actually writing to top of the stack—since ROM cannot be written to. Consequently, SP can be made to point to any arbitrary word in ROM. Upon execution of the next `ret` or `reti` instruction, a word from ROM will be loaded into PC, thereby redirecting control flow to the code that this word points to.

The hard and soft entry points for the MSP430's Bootstrap Loader (BSL) are stored in ROM at `0xC00` and `0xC02` respectively. As Goodspeed showed in [11], entering the BSL at its soft entry point without clearing the password check

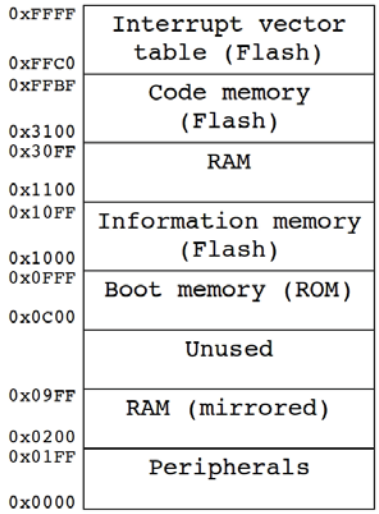


Figure 4: MSP430F2618 16-bit memory map

bit in `R11` allows the attacker to execute privileged BSL operations, such as reading out its firmware. Nesting interrupt handler invocations at a point in firmware where the bits in `R11` are set would therefore allow an attacker to enter the MSP430 BSL with administrative privileges.

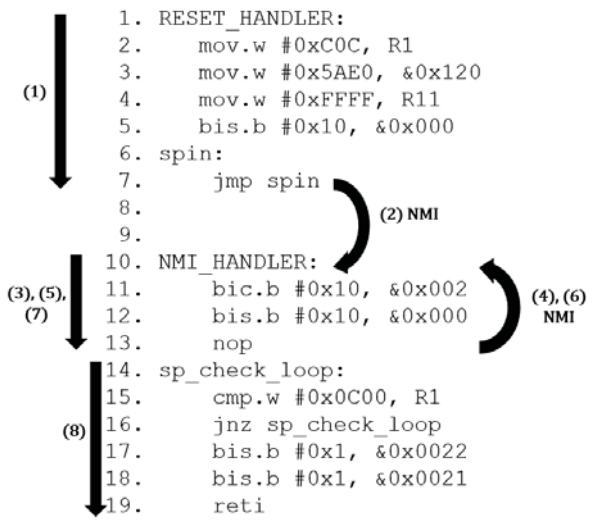


Figure 5: Stack growing exploit

The assembly source code of a simple firmware that contains such a vulnerability can be found in Figure 5. This IOP exploit first sets all bits in R11, then grows the stack using nested NMIs until it is deep enough for the next `reti` instruction to pop the BSL soft entry point into PC. The IOP exploit takes place as follows:

(1) The SP (R1) is initialized at line 1, and the microcontroller is configured trigger NMI interrupts upon receiving a falling edge on the RST/NMI pin at line 2. All bits in R1 are set at line 4, and NMIs are enabled at line 5. The firmware then spins at lines 6-7.

(2) An NMI is triggered, which invokes the NMI handler at line 10. Causing SP to be decremented by two words to `0xC08`. PC and SR are not actually written to memory, since SP is pointing to ROM.

(3) The NMI handler clears the NMI interrupt flag at line 11, and then re-enables NMIs by setting the NMI interrupt enable bit at line 12.

(4) An NMI is triggered, causing SP to be decremented by two words to `0xC04`.

(5) Step 3 is repeated.

(6) An NMI is triggered, causing SP to be decremented by two words to `0xC00`.

(7) Step 3 is repeated.

(8) Since SP is now `0xC00`, the firmware does not spin in the loop from lines 14-16, and instead proceeds to turn on the LED at lines 17-18 and execute the `reti` instruction at line 19. This first pops the word in ROM at `0xC00` into SR, then the word in ROM at `0xC02` (i.e. the BSL soft entry point) into PC. The microcontroller is therefore starts executing its BSL with full privileges (since the password check bit in `R11` is set).

This exploit was designed to be carried out without any specific interrupt timings required. Simply triggering three NMIs would cause the GoodFET42-JTAG to turn on its LED and enter the BSL with full privileges. However, even without the loop checking the value of SP (lines 14 to 16), this exploit is still possible to perform if the NMIs in step 2, 4, and 6 are properly timed.

10. PLANTING A BUGDOOR

An attacker able to read and modify target firmware source code could introduce minimal changes that would leave the target vulnerable to IOP attacks (i.e. leave a "bugdoor" in the firmware). To demonstrate this, we insert a bugdoor into TinyOS [32], an open source operating system targeting low-power wireless embedded systems. Since TinyOS was designed to be lightweight, there is no common kernel running in the background of each TinyOS program. Instead, TinyOS programs are compiled specifically for the target device along with the operating system abstractions required for the program to interface with device hardware. By planting the bugdoor into a TinyOS MSP430 interrupt handler, all MSP430 devices running TinyOS applications that support such an interrupt would be made vulnerable to IOP attacks.

We added two additional lines of code to the USART1 receive interrupt handler generated for the MSP430F1611 on the TelosB mote that makes the interrupt handler reentrant, as shown in Figure 6. This change transforms the USART1 receive interrupt into an unbounded stack growing primitive that can be used to perform the stack growing exploit described in Section 9.3.

The disassembled binary of a TinyOS application (MultihopOscilloscope) compiled for a MSP430 device (TelosB) target compiled with source code modification contains four additional lines of assembly (from `0xA3D2` to `0xA3D8`) in its interrupt handler, as shown in Figure 7.

This bugdoor, which enables privileged access to the MSP430 BSL, required the addition of a mere two lines of code.

```
TOSH_SIGNAL(UART1RX_VECTOR) {
  uint8_t temp = U1RXBUF;
  WRITE_SR( READ_SR | SR_GIE); //+set GIE
  ME2 |= URXE1;                //+set USART1RX IE
  signal Interrupts.rxDone(temp);
}
```

Figure 6: Modified TinyOS source code

```
A3C6    .def __isr_3
A3C6 __isr_3:
A3C6    push.w  R15
A3C6                        ; sig_UART1RX_VECTOR
A3C8    push.w  R14
A3CA    push.w  R13
A3CC    push.w  R12
A3CE    mov.b   &7Eh, R14
A3D2    mov.w   SR, R15     ; tmp = READ_SR
A3D4    bis.w   #8, R15     ; tmp | SR_GIE
A3D6    mov.w   R15, SR     ; SR = temp
A3D8    bis.b   #10h, &5    ; ME2 |= URXE1
...
```

Figure 7: Disassembly of binary produced by modified TinyOS source

This additional code is not obviously malicious, will unlikely be flagged as suspicious by static analyzers, and it will likely be passed off as a benign error if discovered by hand. This small, malicious change could conceivably be pushed to the TinyOS source tree, and if it was, would expose the many embedded MSP430 devices that use TinyOS to the stack-growing exploit.

11. IOP BUGDOORS IN THE WILD?

Now that we showed a novel technique for "bugdooring" firmware by injecting them with synthetic vulnerabilities exploitable by IOP, we can ask: do such vulnerabilities or backdoors actually occur in the wild?

Our answer for now has to be, "we looked, but we haven't spotted any yet." In the spirit of good science, even though our search attempts were inconclusive, we finish with describing our tools and methods, and hope that they will help other researchers who might be interested in IOP attacks.

11.1 Static analysis

To look for potential IOP programs in actual MSP430 firmware, we attempted static analysis of the firmware of five MSP430-powered devices: EZ430U, EZ430URF, FET430UIF, MSP430FET, and GoodFET. We did this manually using IDA Pro and also added some automation using IDAPython scripts and `mspdebug` functions that we wrote.

11.1.1 Manual examination in IDA Pro

In the firmware binaries we examined in IDA Pro, interrupts were not re-enabled within interrupt handlers themselves, therefore preventing the nesting of interrupts and precluding the stack-growing attack described in Section 9.3. Moreover, registers and memory always seemed to be properly set before use, therefore making state accumulation attack described in Section 9.1 unlikely. However, it was clear that interrupt handlers did not completely save and restore CPU state before and after execution, which meant that most interrupt handlers probably modified CPU state in some way after each invocation.

The following describes the static analysis tools we wrote to automate looking for potential IOP programs. None of these scripts/programs *fully* automated the IOP program discovery and payload creation process; they were written to automate certain portions of the static analysis process and to narrow down segments of code/interrupt handlers that might allow for the creation of an IOP program.

11.1.2 Unset register/memory use scanner

This is an IDAPython script that searches for instances where a register or memory address is used to affect state or program behavior without first setting it. For each instruction that uses a register/memory address as the source operand in an instruction that redirects control flow (e.g. `jmp`, `call`) or performs a write to a register/memory (e.g. `mov`, `add`, `push`), this script searches up to n instructions backwards (where n is provided by the user)—including all possible conditional branch paths to that instruction—to check if that source register/memory address is set (e.g. by a `mov` or `clr` instruction). This tool is meant to identify registers/memory addresses that might affect program behavior if their state is modified or accumulated.

11.1.3 Brute-force search

This is an `mspdebug` function we implemented specifically for the simulator module. Given a certain register value pair (R, V), this program performs a brute-force search of the space of possible CPU states, executing all possible sequences of timed interrupts up to n steps into the program (where n is provided by the user). For each of these CPU states the program reaches, if register R contains the value V, the program terminates and reports that this CPU state is reachable. Otherwise, the search continues until all possible interleavings of up to n steps into the program are searched, and reports that such a state cannot be found. We define a *step* as either a instruction in the target program or the execution of an interrupt handler (assumed to be atomic and thus counted as a single step). In theory, this search would be able to determine whether it is possible for a program and its interrupts to put the processor in a "bad", exploitable state. For example, this tool would be able to check if it is possible for SP(R1) to be set to `0xC00` during the execution of an MSP430 program, and therefore determine whether the program is vulnerable to the stack growing exploit described in Section 9.3.

However, given the fact that the space of possible CPU states grows exponentially as n increases, this brute-force search took long amounts of time for even modest values of n; brute-force searching entire firmware binaries was far out of our reach.

Moreover, the latest version of `mspdebug` as of the time of this study (version 0.22) did not support the MSP430X extended instruction set, which some of the target firmwares (e.g. GoodFET) utilize. Since `mspdebug` could not emulate the behavior of the MSP430 on these instructions, our tool was unable to accurately search the space of possible CPU states in these firmwares.

11.1.4 Interrupt state-change analyzer

This is an `mspdebug` function we implemented specifically for the simulator module to determine the state changes in-

duced by the execution of a given interrupt handler at any given point in the program. This function saves the state of the virtual MSP430 CPU before executing the interrupt, executes it, compares the resultant state with the saved state, and prints all the differences in memory and register values to screen. This tool is meant to make understanding the side-effects of interrupts easier, and thus allow interrupt handlers potentially useful in IOP to be easily identifiable.

As expected, most interrupt handlers found in the target firmwares changed modified CPU state without restoring it. However, most the state change induced by the interrupt handlers tended to vary depending on where in the target program the interrupt handler was invoked, as well as how many consecutive times the interrupt handler was invoked. For example, some interrupt handlers only induced state changes after their first invocations, and did not induce further state changes on subsequent, consecutive invocations. In the several cases we discovered where state is either accumulated or changed in a register/memory location, we either found that this register/memory location is only used internally within the handler (e.g. a counter variable), or could not find—by hand or with the help of the Unset Register/Memory Use Scanner—instances where this register/memory was used elsewhere without being set.

Moreover, as mentioned in Section 11.1.3, `mspdebug` does not support the MSP430X extended instruction set. This tool was therefore unable to accurately determine the state changes induced by interrupt handlers that use instructions unique to the MSP430X architecture.

11.2 Hardware analysis

With the help of the FET430UIF, we were able to control the MSP430F2618 and examine CPU state via the JTAG protocol.

We needed to check if MSP430's registers could potentially be used to accumulate state across resets. This could be possible if the registers were not cleared or loaded each time a power-on reset or reset interrupt was trigger. Since the MSP430 manual [30] only explicitly states the values SR and PC take after resets, we needed to observe CPU states across resets in order to determine whether or not the other registers are loaded or cleared after each reset.

A limitation of using the JTAG protocol to debug the MSP430 is that a JTAG reset is performed each time the microcontroller is taken under and released from JTAG control.[31] Therefore, the CPU state displayed upon entering a JTAG "session" in `mspdebug` is that after a JTAG reset, not after a "true" reset interrupt triggered by a signal edge on the RST/NMI pin. The former is different from the latter in that the former is triggered by signals sent over JTAG interface pins (TDO, TDI, TMS, and TCK) rather than by a signal edge on the RST/NMI pin. While documentation states that CPU state is the same after either reset is performed, we suspected that this claim might not hold.

In order to bypass this limitation, we wrote custom firmware that writes all register values to fixed locations in memory immediately after the reset interrupt handler is invoked, so that they could be read out in a subsequent JTAG debugging session.

Using this custom firmware, we found that the MSP430F2618 does indeed load fixed values into its registers (other than PC and SR) on each reset triggered by an edge on the RST/NMI pin.

These values were generally predictable, though we occasionally saw them vary between our experimental sessions. However, it appears safe to assume that fixed values are indeed loaded into registers between resets, therefore preventing state accumulation in registers across multiple resets.

12. FUTURE WORK

Future work should focus on generalizing and fully automating the IOP program discovery process. The first step to achieving this would be to compiling a formal catalog of IOP "gadgets". This would provide direction for finding the building blocks needed to construct IOP programs.

The next step would be to fully automate the IOP static analysis process. The scripts and programs that we have written only semi-automate the static analysis process; further research should focus devising algorithms to locate IOP gadgets in target program binaries reliably and efficiently. One possible avenue to pursue is using SMT solvers to search the CPU state space, rather than the inefficient brute-force search tool we developed for this study. SMT solvers have helped analyze various kinds of vulnerabilities and exploit programming models [33]; IOP seems a promising application for them.

This application would, of course, require a model of the target ISA and of its interactions with interrupts, a significant investment; once this is done, however, using an SMT solver might yield a feasible way to search and query the states achievable through the triggering of interrupts. Such fully automated analyzers would also be able to identify programs vulnerable to IOP, and alert programmers to specific state dependencies between interrupt handler and target code. This would allow microcontroller programmers to better protect their code against potential IOP attacks.

Future work in IOP exploitation should also aim to construct IOP attacks that assume no access to the any part of the firmware on the target device or similar devices. Such attacks might infer the behavior of interrupt handlers purely by observing their side-effects on CPU state, rather than relying on analyzing disassembled binaries. Successfully achieving this stronger form of an IOP attack would future emphasize the significance of this exploit programming model to microcontroller security.

13. CONCLUSION

We have illustrated the concept of interrupt-oriented programming and have demonstrated synthetic examples of firmware code where IOP programs that achieve unintended computation on a microcontroller could be constructed. Moreover, we have shown that these synthetic exploits could be planted as bugdoors into commercially deployed firmware, and have built tools that will help further researchers discover and explore IOP programs in MSP430 firmware.

14. ACKNOWLEDGMENTS

We would like to thank Daniel Beer for helping us to debug some initial problems we had with interfacing our hardware with `mspdebug`. We would also like to thank the members of Dartmouth's Trust Lab—particularly Rebecca 'bx' Shapiro, Peter Johnson, and Jason Reeves—for feedback and useful discussions. Finally, thanks are due to Aurélien Francillon for discussions on minimalist exploitation of microcontrollers.

15. REFERENCES

[1] J. Bangert, S. Bratus, R. Shapiro, and S. W. Smith. The Page-Fault Weird Machine: Lessons in Instruction-less Computation. In *7th USENIX Workshop of Offensive Technologies (WOOT)*, August 2013. https://www.usenix.org/system/files/conference/woot13/woot13-bangert.pdf.

[2] E. Bosman and H. Bos. Framing Signals – A Return to Portable Shellcode. IEEE Symposium on Security and Privacy, May 2014.

[3] S. Bratus, J. Bangert, A. Gabrovsky, A. Shubina, D. Bilar, and M. E. Locasto. Composition Patterns of Hacking. In *Proceedings of the 1st International Workshop on Cyber Patterns*, pages 80–85, Abingdon, Oxfordshire, UK, July 2012.

[4] S. Bratus, M. E. Locasto, M. L. Patterson, L. Sassaman, and A. Shubina. Exploit Programming: from Buffer Overflows to "Weird Machines" and Theory of Computation. *;login:*, December 2011.

[5] E. Buchanan, R. Roemer, H. Shacham, and S. Savage. When Good Instructions Go Bad: Generalizing Return-oriented Programming to RISC. In *Proceedings of the 15th ACM Conference on Computer and Communications Security*, pages 27–38, New York, NY, USA, 2008. ACM.

[6] S. Checkoway, L. Davi, A. Dmitrienko, A.-R. Sadeghi, H. Shacham, and M. Winandy. Return-oriented Programming Without Returns. In *Proceedings of the 17th ACM Conference on Computer and Communications Security*, pages 559–572, New York, NY, USA, 2010. ACM.

[7] T. Dullien. Exploitation and State Machines: Programming the "Weird Machine", revisited. http://www.immunityinc.com/infiltrate/presentations/Fundamentals_of_exploitation_revisited.pdf, April 2011. Infiltrate Conference.

[8] T. Durden. Bypassing PaX ASLR protection. *Phrack 59:9*, 59(9), Jul 2002.

[9] eugene. Architecture spanning shellcode. Phrack 57:14, November 2001. http://phrack.org/issues/57/17.html.

[10] J. Evers. Alcatel leaves LAN switch software back door wide open, 2002. http://www.networkworld.com/news/2002/1122alcatellan.html.

[11] T. Goodspeed and A. Francillon. Half-blind Attacks: Mask ROM Bootloaders Are Dangerous. In *Proceedings of the 3rd USENIX Conference on Offensive Technologies*, WOOT'09, Berkeley, CA, USA, 2009. USENIX Association.

[12] C. Helfmeier, D. Nedospasov, C. Tarnovsky, J. S. Krissler, C. Boit, and J.-P. Seifert. Breaking and entering through the silicon. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, CCS '13, pages 733–744, New York, NY, USA, 2013. ACM.

[13] jp. Advanced Doug Lea's malloc Exploits. Phrack 61:6. http://phrack.org/issues.html?issue=61&id=6.

[14] S. Krahmer. x86-64 buffer overflow exploits and the borrowed code chunks exploitation technique. http://users.suse.com/~krahmer/no-nx.pdf, September 2005.

[15] J. Mason, S. Small, F. Monrose, and G. MacManus. English shellcode. In *Proceedings of the 16th ACM Conference on Computer and Communications Security*, CCS '09, pages 524–533, New York, NY, USA, 2009. ACM.

[16] H. Meer. The (Almost) Complete History of Memory Corruption Attacks. BlackHat USA, August 2010.

[17] T. Müller. ASLR Smack & Laugh Reference, 2008.

[18] Nergal. The advanced return-into-lib(c) exploits: PaX case study. *Phrack*, 11(58), December 2001.

[19] J. Oakley and S. Bratus. Exploiting the hard-working dwarf: Trojan and exploit techniques with no native executable code. In *WOOT*, pages 91–102, 2011.

[20] G. Richarte. Re: Future of Buffer Overflows. Bugtraq, October 2000. http://seclists.org/bugtraq/2000/Nov/32.

[21] G. Richarte. About Exploits Writing. Core Security Technologies presentation, 2002.

[22] riq and gera. Advances in Format String Exploitation. *Phrack 59:7*, 59(7), Jul 2002.

[23] rix. Writing ia32 alphanumeric shellcodes. Phrack 57:15, November 2001. http://phrack.org/issues/57/18.html.

[24] H. Shacham. 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*, CCS '07, pages 552–561, New York, NY, USA, 2007. ACM.

[25] R. Shapiro, S. Bratus, and S. W. Smith. "Weird Machines" in ELF: A Spotlight on the Underappreciated Metadata. In *7th USENIX Workshop of Offensive Technologies (WOOT)*, August 2013. https://www.usenix.org/system/files/conference/woot13/woot13-shapiro.pdf.

[26] Shimmer. Shop: Shimmer Platform. http://www.shimmersensing.com/shop/shimmer-platform.

[27] Solar Designer. Getting around non-executable stack (and fix). *Bugtraq mailing list archives*, August 1997. http://seclists.org/bugtraq/1997/Aug/63.

[28] Solar Designer. Openwall linux kernel patch. Web, 1998. http://www.openwall.com/linux.

[29] B. Spengler. The Case for GrSecurity. https://grsecurity.net/papers.php, September 2012. H2HC, Sao Paulo, Brazil.

[30] Texas Instruments. *SLAU144I: MSP430x2xx Family: User's Guide*, January 2012.

[31] Texas Instruments. *SLAU320L: MSP430 Programming Via the JTAG Interface*, September 2013.

[32] TinyOS Documentation Wiki. TinyOS. Web. http://tinyos.stanford.edu/tinyos-wiki/index.php/Main_Page.

[33] J. Vanegue, S. Heelan, and R. Rolles. SMT solvers for software security. In *Proceedings of the 6th USENIX Conference on Offensive Technologies*, WOOT'12, Berkeley, CA, USA, 2012. USENIX Association.

[34] R. Verma. Designing Portable, Wearable and Implantable Medical Electronics with Ultra-Low-Power Microcontrollers. http://tinyurl.com/lc4rat5.

Through the Eye of the PLC: Semantic Security Monitoring for Industrial Processes

Dina Hadžiosmanović
Delft University of Technology
The Netherlands

Robin Sommer
ICSI/LBNL
Berkeley, CA
USA

Emmanuele Zambon
University of Twente
SecurityMatters B.V.
The Netherlands

Pieter H.Hartel
University of Twente
The Netherlands

ABSTRACT

Off-the-shelf intrusion detection systems prove an ill fit for protecting industrial control systems, as they do not take their process semantics into account. Specifically, current systems fail to detect recent process control attacks that manifest as unauthorized changes to the configuration of a plant's programmable logic controllers (PLCs). In this work we present a detector that continuously tracks updates to corresponding process variables to then derive variable-specific prediction models as the basis for assessing future activity. Taking a specification-agnostic approach, we passively monitor plant activity by extracting variable updates from the devices' network communication. We evaluate the capabilities of our detection approach with traffic recorded at two operational water treatment plants serving a total of about one million people in two urban areas. We show that the proposed approach can detect direct attacks on process control, and we further explore its potential to identify more sophisticated indirect attacks on field device measurements as well.

1. INTRODUCTION

Industrial control systems (ICS) monitor and control physical processes, often inside critical infrastructures like power plants and power grids; water, oil and gas distribution systems; and production systems for food, cars, ships and other products. As these environments differ from traditional IT systems, they also face unique security challenges that render effective protection challenging. Off-the-shelf intrusion detection systems (IDS) prove a particularly ill fit. Classic signature matching requires precise patterns of anticipated intrusions—an unrealistic assumption in a setting where attacks remain rare overall, yet may carefully target their victims—and existing behavioural approaches fail to incorporate the domain-specific context of operating in these specialized environments [18]. While a few network IDS now include ICS-specific protocol support, their capabilities remain limited to finding low-level technical attacks, such as protocol violations and buffer overflow exploits. They fail, however, to address the fundamentally different threat of malicious changes to a plant's *process control*. Stuxnet represents the most prominent example of such an attack: it manipulated the speed of centrifuges to run outside of their operational range, eventually causing physical damage. While Stuxnet demonstrates a level of sophistication that is feasible only for the most resourceful attackers, SABOT [26] lowers the bar for similar attacks by recovering the mapping between the memory of programmable logic controllers (PLC) and the actual process inputs.

To report process control modifications, an IDS can monitor the status of PLC variables: these attacks manifest as unauthorized changes to a device's parametrization (e.g., setpoints, status). In practice, however, it proves challenging to provide an otherwise independent IDS with access to such process-level PLC activity. The combination of a PLC's embedded nature, its critical role in a plant's operation, and its often proprietary internals, all but prohibits creating a direct interface to an external monitor. It turns out, however, that an alternative approach can provide the relevant information without touching the device: as any configuration changes must reach the PLC through the ICS' network infrastructure, one can passively monitor the network traffic for the corresponding update commands. Doing so proves non-obtrusive and, hence, easy to deploy and scale: it requires just a standard network tap that provides the IDS with a copy of the PLCs' traffic.

In this work we develop such a semantic, network-based IDS. Our system extracts process operations from a PLC's raw network packets, and then constructs a corresponding time series for each process variable to understand its expected activity. We derive variable-specific forecasting models that partially borrow from the process safety community's established domain expertise, and we assess their predictive power with an initial objective to understand where they work well, yet also cases that fail to yield a stable baseline. For our analysis we use Modbus traffic recorded over two-week periods at two operational water treatment plants serving a total of about one million people in two urban areas.

Our results show that this approach allows for straightforward detection of *direct* attacks on process control: they manifest as changes to variables that normally would stay stable, enabling to detect the corresponding attacks that the recent literature discusses. We further explore the potential for extending this approach to *indirect* process control attacks, which reflect only as deviations in field measurements,

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.
ACSAC '14 December 08 - 12 2014, New Orleans, LA, USA
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-3005-3/14/12 ...$15.00.
http://dx.doi.org/10.1145/2664243.2664277.

either because of tampering with sensors or due to a direct control change through an unobservable channel. At first, we find that here the real world introduces artefacts that limit the power of our initial models. However, when we investigate a series of illuminating cases in more detail, we identify possible improvements that can significantly increase their prediction capabilities.

Overall, we consider this work as a step towards non-obtrusive, semantic security monitoring for ICS networks. To the best of our knowledge, no prior work has yet examined operational industrial processes to a similar level of detail from a network vantage point. We structure the remainder of this paper as follows. In §2 we discuss process control attacks in more detail. In §3 we present our monitoring approach, along with a testbed scenario (§3.2) that we use for demonstrating the capabilities of the presented techniques. §4 summarizes our implementation, and §5 evaluates our approach. In §6 we present related work, and §7 provides concluding remarks.

2. DETECTING PROCESS ATTACKS

A typical ICS includes a number of common components, including a human-machine interface (HMI), supervisory infrastructure, field devices, and communication infrastructure. For our work we focus on programmable logic controllers (PLCs), which constitute embedded devices that run custom code to control the field devices (e.g., regulate the speed of a pump). For an attacker, the PLCs provide the most effective point to penetrate, as they control the target processes. To better understand the spectrum of possible network-based attacks on PLCs, we surveyed plausible scenarios carried out over the Modbus TCP protocol as a case study. In Table 1 we aggregate the attacks into three categories. Basic *Level 1* attacks operate at the IP or TCP level, such as manipulating packet sizes. *Level 2* attacks violate semantics of the Modbus protocol at the payload-level, such as breaking the protocol conventions or transmitting values outside the range of either the protocol specification or what the receiving side supports (e.g., invalid function codes). Finally, *Level 3* attacks represent activity that is legal at the protocol level, yet violates semantic constraints that a process imposes, including both semantically incorrect messages (e.g., conflicting commands) and operations that lead the site into an undesirable state (e.g., a command to open a pump when it must remain shut). We distinguish three subtypes of such *process* attacks: *(i)* reconnaissance (e.g., discovering process configuration), *(ii)* direct control (e.g., modifying process setpoints) and *(iii)* indirect control (e.g., tampering with exchanged variable values to trigger undesirable reaction). *Level 3* represents the most challenging target for detection, and three recent examples of (primarily) *direct control* attacks demonstrate the fundamental limitations of current systems to report them. First, Carcano et al. [10] present a proof-of concept malware that manipulates values be generating Modbus packets carrying syntactically correct commands, yet with inverted values. Second, SABOT [26] performs a semi-automated analysis of PLC code and high-level process descriptions to identify key variables for then crafting malicious attacks that divert a process; in contrast to the work of Carcano, this attack assumes a highly knowledgeable attacker. Finally, Stuxnet constitutes the most prominent real-life attack on process integrity. By attacking PLCs, it crossed a boundary as the first publicly known malware that injected semantically meaningful commands into a highly specific plant environment. Stuxnet managed to divert a process by generating malicious yet technically valid control commands that changed the behaviour of centrifuges. Additionally, Stuxnet had an indirect component tampering with the flow of information presented to the operators, causing them to miss the attack's progress.

All three examples manifest as illegitimate write operations that update process settings to harmful values. However, monitoring PLCs for such changes externally proves challenging as hardly any site will allow an IDS to interface with the devices directly. In this work we pursue an indirect, passive way of tracking PLC activity by recreating process semantics from their network communication. In the following we show that doing so enables inferring process behaviour at a level sufficient to detect the three direct attacks we discuss above. We further demonstrate the potential of network-level semantic monitoring for finding indirect process control attacks as well. Our approach does generally not depend on the attacker's knowledge about the plant. For direct attacks we assume that the logic for issuing PLC updates remains uncompromised, whereas for indirect attacks we do not impose any constraints.

3. APPROACH

We now present our approach to detect attacks that aim to manipulate the process control, both direct and indirect. It proceeds in three main phases: *(i) extraction* distills current variable values out of network traffic; *(ii) characterization* divides the observed process variables into three categories that we examine separately; and *(iii) modelling and detection* derives behavioural models for each variable and reports when new observations deviate from what they predict. We discuss these phases individually in §3.3–3.5, after first introducing technical background in §3.1 as well as a small testbed setup in §3.2 that we use for illustration.

3.1 Background

Conceptually, we find two semantic groups of network communication between ICS components: *(i)* process awareness, and *(ii)* process control. *Awareness* propagates status information about the controlled process across devices (also referred as *annunciators* in [26]). In particular, the ICS supervisory infrastructure requests regular updates from the PLCs to report the current plant status to its operators. In addition to escalating critical updates for timely reaction, awareness also collects trending data for long-term process analysis. A typical update cycle spans a few seconds. PLCs also propagate awareness information across themselves to ensure that each device learns sufficient information about critical variables before entering the next process stage (e.g., PLC 1 might require information about the state of a field device connected to PLC 2 before starting a subsequent process stage). *Process control* is generally exercised in one of two ways: *(i)* by PLCs according to their embedded logic; and *(ii)* by operator commands that override PLC internal logic. In either case it is the PLC that carries out the action, and hence will reflect the process change as an update to its internal state.

PLC's process variables characterise the current operation state. Examples of typical variables include the setpoint (i.e., configuration setting) for a physical process, the cur-

Table 1: Summary of plausible attacks against PLC implementations: Modbus example

Level	Impact		Attack description	Example
1	Data integrity		Corrupt integrity by adding data to the packet.	Craft a packet that has a different length than defined in parameters or in spec [2].
2	IT System	Reconnaissance	Analyse functionality a PLC implements.	Probe FC, listen for responses and exceptions [2].
		Integrity	Exploit lack of specification compliance.	Manipulate application parameters within spec (e.g., offset) or outside of spec (e.g., illegal FC) [2, 9, 37].
			Perform unauthorized use of an administrative command.	Use FC 8-0A to clear counters and diagnostics audit [2].
		Denial of service	Perform MITM to enforce system delay.	Send exception codes 05, 06 or FC 8-04 to enforce Listen mode [2].
			Perform unauthorized use of administrative command.	Use FC 8-01 to restart TCP communication [2, 9].
3	Process	Reconnaissance	Analyse structure of memory map.	Probe readable/writable points. Exceptions tell process implementation details [2].
		Direct control	Perform change on process variable.	Write inverted or min/max values [10]. Modify key setpoint variables [14, 26].
		Indirect control	Tamper with process values.	Replay values [14].

FC: Function code defining the type of functionality in Modbus. *MITM: Man-in-the-middle attack.*

rent value of a valve sensor, and the current position in a cycle of program steps. Process variables serve as input to the PLC code. For example, a variable value representing a high pressure level might trigger the start of a draining stage. Likewise, the PLC carries out operator commands by writing into corresponding variables. For example, a command to open a valve would update a variable that the program code is regularly checking; once it notices the update, it outputs the corresponding analogue signal to the physical device.

Within the device, process variables map to PLC memory cells through the data model of the PLC. The data model determines the representation of the process data both inside the PLC's memory and on the network level, usually tying in directly with the communication protocol that a vendor opts to use. The actual mapping is determined by a PLC programmer and thus remains specific to each PLC instance. Examples of data models in different protocols include: Modbus using *memory maps* of 16-bit *registers* and 1-bit *coils*; Profinet defining *slots*, *subslots*, and *channels*; and MMS using *objects* to address process variables [21].

3.2 Testbed Scenario

In §5 we validate our approach using network traffic from water purification operational plants as our work-load. However, to illustrate capabilities in a controlled environment, we here set up a small testbed consisting of one PLC and one HMI workstation, using a simplified version of a demonstration kit from a corresponding ICS vendor.

The process under control comprises six plant components (see Figure 1): a tank, a heater, two valves, a level sensor, and a temperature sensor. The process consists of

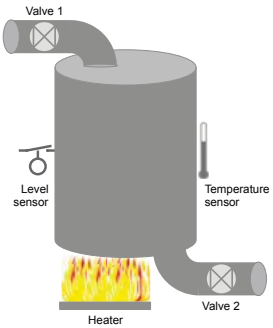


Figure 1: Process setup of the testbed environment

three operations that repeat continuously: tank filling, water heating, and tank draining. The HMI collects nine variables for measurements (tank level and water temperature), control (e.g., valve status, tank level setpoint), and reporting (tank level alarm). Table 2 shows the relevant parts of the PLC's memory map.

As a safety constraint, we implement an alarming system that triggers when process variables reach undesirable values. To illustrate the detection, we use two attack scenarios: *(i)* changing the level setpoint to overflow the tank; and *(ii)* sending the PLC tampered measurement information to trigger process changes or mislead operator awareness. These attacks closely relate to the surveyed attacks from the literature: the first resembles *direct process attacks* (e.g., SABOT and Stuxnet-like attacks) while the second corresponds to *indirect process attacks* (e.g., measurement tampering by Stuxnet).

Table 2: Testbed PLC memory map

Reg.	Name	Type	Desc
HR0010	V1On	bool	Status of valve 1
HR0011	V2On	bool	Status of valve 2
HR0012	HeaterOn	bool	Status of the heater
HR0020	TankLevelSP	fixpoint	SP tank level (L)
HR0021	TankLevel	fixpoint	Level of the tank (L)
HR0022	TempSP	fixpoint	SP water temp.
HR0023	Temp	fixpoint	Water temp (celsius)
HR0030	TankLevelAl	enum	Alarms tank level
HR0031	TempAl	enum	Alarms water temp.

SP: setpoint

3.3 Data Extraction

The data extraction phase is a preprocessing step that distills the values of process variables out of ICS network traffic. It consists of two subparts: *(i)* parsing the application-layer network protocol to extract the relevant commands, including all their parameters; and *(ii)* constructing *shadow memory maps* inside the analysis system that track the current state of all observed process variables, providing us with a mirror of the PLCs' internal memory.

For this work we focus on parsing Modbus, in which each command comes with a set of parameters as well as a data section. Basic parameters include function/sub-function codes that define the operation, an address reference specifying a memory location to operate on, and a word size giving the number of memory cells affected. The data section includes the actual values transmitted, i.e., the current value for a *read* operation and the intended update value for a *write*. We maintain shadow memory maps by interpreting each exchanged message according to its semantics, updating our current understanding of a PLC's variables accordingly.

The following (simplified) commands from our testbed setup (see §3.2) illustrate the extraction step.

```
Time 1: PLC 1, UID: 255, read variable 21, value: 10
Time 2: PLC 1, UID: 255, read variable 21, value: 14
Time 3: PLC 1, UID: 255, read variable 21, value: 18
```

After processing the messages, the shadow memory map will report 18 as the current value for variable 21. As we know from the testbed's configuration, that variable corresponds to the *tank level* and, hence, will reflect three distinct types of behaviour: an increasing trend during filling, a constant value during heating, and a decreasing trend during the draining phase. Indeed, our extraction step confirms this expectation: The following list represents a small excerpt from variable 21's values, as extracted from actual network traffic inside the testbed:

```
...490,492,494,496,498,500,500,500,500,500,500,
...496,492,488,484,480,476,472,468,464,460,456, ...
```

We have confirmed that these extracted values indeed match what the PLC stores internally over time.

3.4 Data Characterisation

Next, we perform a *characterization* phase that separates PLC variables into different categories. Based on interviews with plant engineers, we identify four general groups: *(i) control*: variables for configuring plant operation (e.g., device setpoints, configuration matrix); *(ii) reporting*: variables for reporting alarms and events to operators through HMI or other PLCs (e.g., pump load is too high); *(iii) measurement*: variables reflecting readings from field devices and sensors (e.g., current tank level, current water flow), *(iv) program state*: variables holding internal PLC state such as program counters, clocks, and timeouts. While the character of variables varies with their groups, we observe three cases that suggest specific models for predicting future behaviour: most variables either *(i)* change continuously, and gradually, over time; *(ii)* reflect attribute data that draws from a discrete set of possible values; or *(iii)* never change. The first is typical for sensor measurements; program state and reporting tend to use the second; and the third proves common for process settings (e.g., setpoints). Taking a specification-agnostic approach, there is no definite resource to directly tell what type of data a variable reflects—recall from §3.1 that memory maps are specific to each PLC instance. Thus, we apply heuristics to categorize process variables according to the behaviour we observe. In our testbed we can directly cross-check if the results indeed match the configuration.

More specifically, our heuristics leverage an observation—corroborated by plant engineers—that reporting variables are typically encoded in bitmaps that, depending on the number of distinct reporting events, map to discrete set of values. We use this as criteria for differentiating between the three categories by counting the number of distinct values we see for each variable across a training set. If we observe only a single value, we consider the variable to represent a constant value. We consider a set of up to 2^k distinct values, for a chosen k, as an attribute series. If we see more than 2^k values for a variable, we assume it holds a continuous range.

In our testbed environment, we set the parameter $k = 3$. Running the characterisation with 2 hrs of its network traffic leads to classifying 4 of the 9 variables as constant, 3 as attributes, and 2 as a continuous—which aligns correctly with their actual semantics.

3.5 Data Modelling and Detection

Now that we can distinguish between constant, attribute and continuous time series, we proceed with building behavioural models.

Modelling. To model constant and attribute data, we derive a set of expected values (i.e., the enumeration set for attribute data; the one observed value for constant series). To model continuous data, we leverage two complementary techniques, *autoregression modelling* and *control limits*. Since we do not tailor our model to any specific process description, we choose to leverage techniques that are *(i)* relevant to the context (i.e., commonly used in safety monitoring) and *(ii)* do not pose strong assumptions on the monitored time-series. Autoregressive modelling represents a common technique to capture the behaviour of a correlated series, such as successive observations of an industrial process [38]. An autoregressive model of order p states that x_i is the linear function of the previous p values of the series plus a prediction error term [8]:

$$x_i = \phi_0 + \phi_1 x_{i-1} + \phi_2 x_{i-2} + ... + \phi_p x_{i-p} + \epsilon_i$$

where $\phi_1, ..., \phi_p$ are suitably determined coefficients and ϵ_i is a normally distributed error term with zero mean and non zero variance σ^2. Using the autoregressive model, we can make one-step estimations for future values of the process variable underlying the time series. This way, the model can be used to detect stream deviations. However, as for any regression, a set of small changes can take the stream

outside of its operational limits without exhibiting regression deviations [38]. To address this, we apply a complementary strategy, namely Shewart control limits [38]. The control limits represent a pair of values $\{L_{min}, L_{max}\}$ that define the upper and lower operational limit of the process variable. Typically, the limits are calculated as values that are three standard deviations from the estimated mean.

While the presented techniques are generally deployed in time series analysis, including specifically in the safety domain, our application approach is *(i)* broader, applied indiscriminately on the full spectrum of exchanged messages (i.e., data series) and *(ii)* more generic, posing no assumption on the expected behaviour of the analysed data series. While necessarily exhibiting a lower accuracy than approaches would that tailor specifically to the semantics of individual variables, our modelling enables building a comprehensive network monitor for a wide range of processes.

Detection. For constant and attribute series we raise an alert if a value in a series reaches outside of the enumeration set. To detect deviations in continuous series, we raise an alert if the value *(i)* reaches outside of the control limits or *(ii)* produces a deviation in the prediction of the autoregressive model. More specifically, for estimating the deviation in the autoregressive model, we compare the residual variance (observed during training) with the prediction error variance (observed during testing). A prediction error variance that is significantly higher than the residual variance implies that the real stream has deviated from the estimated model, and thus we should raise an alert [19].

To illustrate the detection capabilities in a controlled environment, we test our approach on two semantic attacks crafted for the process operating in our testbed. The first attack effectively consists of a command that changes the tank level setpoint (HR0020 in Figure 2). As a result, the tank's filling phase (HR0021 in Figure 2) continues until the water level overflows the tank capacity. Results show that this attack is detected as *(i)* a deviation in the setpoint variable and *(ii)* a value reaching the maximal control limit $Lmax$. The second attack consists of a set of commands that tamper with the value for the temperature level (HR0023 in Figure 3). As a result, the tank's filling phase terminates early, the heating process begins but immediately stops, and then the draining process starts. As a consequence, both the heater and the boiler may get damaged. In this second scenario no alarm is generated by the PLC (since the observed levels never reach any undesirable value), yet our approach detects a deviation in the behaviour.

Our results demonstrate the expected capabilities (and limitations) of both approaches [38]. In particular, an autoregressive model is effective for detecting sudden changes (e.g., detection of the second attack) while control limits are generally a good strategy for maintaining the specific mean level of a process variable (thus, can detect the process drift in the first attack). In relation to the known attacks, the results demonstrate that our approach can in principle detect *direct* control modifications (our first attack closely resembles the corresponding semantic attacks in [10, 14, 26]). In the testbed we also show detecting indirect control attacks (the second attack manipulates measurements). However, as any operational ICS setting will pose more challenges than our small controlled environment, we analyse the feasibility of our approach with data from two real-world plants in §5.

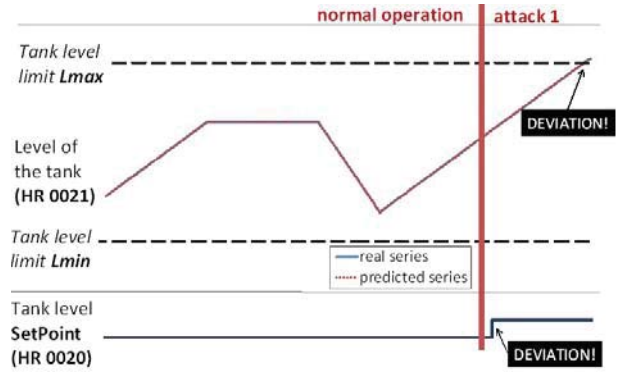


Figure 2: Illustration of the configuration change

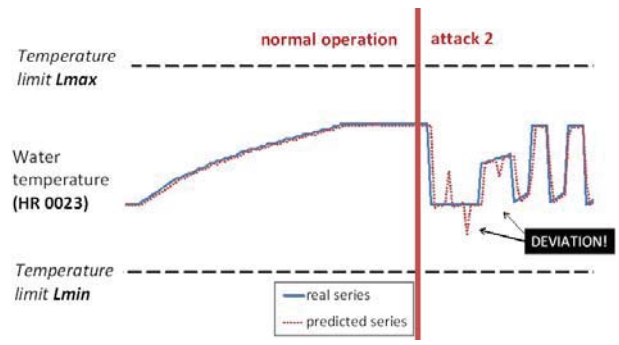


Figure 3: Illustration of measurement tampering

4. IMPLEMENTATION

We implemented a prototype of our approach using a combination of Bro [31] and custom C++ code. Bro performs the initial *data extraction* step. We developed a Modbus analyser for Bro that extracts the main protocol commands from network traffic and makes them available to scripts written in Bro's custom policy language. We leverage Bro's BinPAC [30] parser generator to automatically generate much of the Modbus-specific code from a corresponding grammar. The Bro project has integrated our Modbus analyser into Bro 2.2. We use a custom Bro script to record the parsed Modbus commands into an ASCII-based log file, which we then process with external C++ code implementing the characterisation, modelling, and detection phases. After a preliminary analysis testing different values of k for the data characterisation phase, we choose $k = 3$ for our tests with real environments as that value showed the least mismatches. For modelling continuous series, we build the autoregressive model and derive process control limits. We leverage an open source implementation of the autoregressive model.[1] To estimate autoregressive coefficients, we use Burg's method [19]. To estimate the order of the model, we use the common Akaike information criterion [34]. As control limits, we implement Shewart control limits following the description in [38]. For each continuous series we derive and estimate the behaviour with both the autoregressive and control limit models. Finally, for detecting deviations in continuous series via the autoregressive model, we use two variance hypothesis tests (commonly known as F-test). For both tests we set $p = 0.05\%$ as their significance level.

[1] Available at https://github.com/RhysU/ar.git

5. EVALUATION

Our work represents an initial step towards modelling the semantics of process variables from a network vantage point. To the best of our knowledge, we offer the first study analysing, at this level, week-long network traces from operational plants, where each PLC operates with thousands of process variables. As such, we are primarily interested in understanding properties of the setting that impact semantic approaches to security monitoring, and less in specific true/false positives rates. With that perspective, we evaluate our memory map modelling with two overarching objectives: *(i)* understand the degree to which our approach can successfully predict typical process behaviour; and *(ii)* gain insight into the underlying activity that improves, or weakens, its accuracy. In particular, we do not aim to evaluate detection rates, due to the inherent difficulty of achieving meaningful results in a realistic setting. As process attacks remain both novel and rare, we cannot expect our traces to contain malicious activity (and as far as we know, they do not). Yet it also remains unrealistic to inject crafted attacks into the traces, as we would remain limited to trivial, *direct control* cases like those already demonstrated in §3.5—which our detector would find for the same reasons as discussed there. It is hard to see how one could possibly inject *indirect control* attacks into existing, static traces, as these would normally trigger a chain of effects that only a full simulation of the plant could reproduce.

In the following we first introduce our two real-world environments in §5.1. Then in §5.2 we present the results of the characterisation and modelling phases. We do not analyse in depth the data extraction here as it constitutes primarily a pre-processing step.

5.1 Environments and Data Sets

Our data comes from two real-life water treatment plants that serve a total of about one million people in two urban areas; they are part of a larger system of over 30 sites controlled by one company. The two plants are of comparable size, and they perform semantically similar tasks (e.g., water pumping, purification, ozone treatment). Nevertheless their setups look quite distinct. They deploy different numbers of PLCs (3 vs. 7), and chose different strategies to divide processes among them. The PLC memory maps differ both between the two environments, and also among PLCs of the same site. While both plants use equipment from the same vendor, they deploy different software versions.[2] We have access to one 3-day and one 14-day long packet trace from the two plants, respectively, both containing the complete network traffic captured from mirroring ports of switches that connect the PLCs and the ICS servers. The traces include 64GB and 101GB of network traffic, respectively, with bandwidths varying between 9 and 360 packets/sec during the recorded periods. We find two ICS protocols in use: Modbus (between PLCs and ICS servers) and a vendor proprietary protocol (between ICSs server and HMI). In addition, we see six further non-ICS protocols in use by servers and workstations: VNC, SSL, FTP, HTTP, SMB, and DCOM. The non-ICS protocols constitute an negligible fraction of the overall network trace. Of the 20 and 28 hosts active in the two traces, 7 and 11 exchange Modbus messages. While we see the vendor proprietary protocol in use among hosts that are part of the supervisory infrastructure, we observe only Modbus for all communication involving PLCs. We see three types of read/write Modbus messages in the traces (function codes 3, 16 and 23).

To present our results, we select 5 PLCs taken from both plants, namely: all 3 PLCs from the first plant, and 2 PLCs from the second plant. The second plant operates with 7 PLCs in total, of differing complexity: the number of process variables goes as low as 135 for three of them and as high as 3500 in one; each of the remaining four PLCs operates with approximately 2200 variables. We select two PLCs representing the most simple and the most complex setup, respectively. The plant operators also provided us with *project files* that describe each PLC's memory map layout, exported by PLC programming environments in the form of CSV files holding information on addressing, data type, and process role for each process variable defined by a PLC. These files closely resemble the information shown by the example in Table 2, as well as in [32]. We do not use the project files for modelling but, instead, leverage them for explaining observed deviations at a later stage.

5.2 Data Characterisation and Modelling

First we run the characterisation (described in §3.4) on 3 days of network traffic from the first plant's PLCs. We find that it classifies 95.5% of all variables as constant series, 1.4% of variables as attribute, and 3.1% as continuous series. Further analysis with different batch choices shows that these results remain consistent over time for any intervals longer than one day. Next, to evaluate our modelling approach, we first generally measure the quality of our models. In the second, more interesting step, we then dig deeper into the results and focus on understanding the underlying reasons and situations in which our approach *(i)* indeed models process activity correctly; and *(ii)* fails to capture the plant's behaviour, flagging benign deviations as alarms. Our objective here is to gain insight into the capabilities and limitations of our approach applied in real life environments, as well more generally into potential and challenges of modelling process activity at the semantic level.

Generally, we consider our approach to generate an "alert" on a process variable when, at any time during testing a batch of data, an observation deviates from the prediction—i.e., when observing an unexpected value for constants and attributes, or a value outside the autoregressive/control limit models for a continuous time series. We perform 3-fold cross validation using the *rolling forecasting procedure* [20] on a set of 3-day batches of data extracted from the two plant's network traces. The first two batches of data come from the first plant, each representing a randomly chosen continuous range from the first and the second weeks, respectively. The third batch represents the complete trace from the second plant (recall that we only have 3 days of network traces available from that plant). At a technical level, a 3-day batch size gives us a reasonable volume of data suitable for processing repeatedly with our implementation. At an operational level, operators confirm to us that one day matches a typical PLC work cycle.

In Table 3 we summarize the results of the testing across different behaviour models.

[2]According to agreements with the two sites, we cannot name the equipment vendor.

Table 3: Testing model capabilities

	Deviating variables across different types of series (mean %/ st.dev)		
	Constant	*Attribute*	*Continuous*
PLC 1a	0.5/0.29	19.05/0.2	57.49/5.78
PLC 1b	0.31/0.04	19.80/1.4	44.64/5.41
PLC 1c	0.14/0.02	19.55/4.0	37.58/2.98
PLC 2a	0.64/0.0	26.92/0.0	63.63/0.0
PLC 2b	0.0/0.0	0.0/0.0	0.0/0.0

For each pair of category and PLC, we compute the percentage of alerts, showing mean and standard deviation over the batches. In the following, we discuss the three categories separately.

5.2.1 Constant Series

Our results show that by far the most variables that we classify as constant indeed stay stable over time. This corroborates the general perception that ICS configurations do not change often. Examining the small number of deviating variables (0.0% to 0.6%) in this category, we observe two main causes: *(i)* configuration changes, and *(ii)* misclassifications of the variable type. The former confirms that our approach indeed detects *direct* control modifications. While in this case the changes prove legitimate, they remain sufficiently rare that an IDS can generally report them for human inspection (or whitelisting). Examining the reported situations in more detail, we see that 60 variables all trigger at the same time due to an update to a "configuration matrix", a large data structure that defines an operation mode in terms of a set of values controlling multiple devices simultaneously (as it turns out, it is a single packet from the HMI that performs a "multiple register write"). We confirm that this is the only example of such a large configuration change in the analysed trace.

The second cause for unexpected deviation represents shortcomings of our data classification phase. For example, during one of the folds in PLC2a we find that it misclassifies 9 out of 15 measurement variables representing aggregated flow information as constant due to a lack of activity during the training period. Similarly, in the same fold we find that 7 out of 18 device statuses (which represent attribute data) get misclassified as constants. In both cases the values remain constant for more than 20 hours, yet then change during the testing period. Interestingly, we see several such situations that first trigger an alert for a configuration change (e.g., the status of a filter in PLC1b changes for 15mins after it has spent 21 hours in a previous state), followed by a burst of further ones reflecting the change being in effect now (i.e., variables representing activity linked to that filter start to deviate from constant behaviour: status, volume, throughput/hour, total throughput). In this case, the two main causes of errors are hence linked semantically. As others have observed, such *sequential variations* [26] represent cases that one could handle specifically, for example by merging into a single event.

Discussions with the plant operators confirm that daily *cleaning* activities on that PLC might cause such sudden changes for a short amount of time. It turns out that the next fold avoids this misclassification due to its longer training interval, confirming that when training spans the corresponding work cycle, the modelling of constants indeed works as expected.

5.2.2 Attribute Data Series

Our test shows a higher (0.0% to 26%) but stable number of deviating attribute series across all tested folds. By sampling a subset we find that their main cause of alerts concerns continuous variables which, due to slow process character, exhibit only a limited number of distinct values during a training interval, and are thus wrongly labelled as attribute data (e.g., variables describing time information in the form of date and hour). Apart from this special case, variables describing alarms and statuses do not report any alerts (i.e., there are no new types of alarms and events in the analysed trace).

With attribute data, sequences can carry important semantic information, as such variables often encode the current process state within a series of steps (e.g., alarm type X raised to operator, operator acknowledged, alarm cleared, state normal). Omitting specific steps can represent a threat, however the current analysis cannot capture such cases and applying a continuous model is counterproductive: while for attribute data ordering matters, timing does typically not (e.g., an operator may acknowledge an alarm at any time). In §5.3 we present an approach for further improving the analysis of attribute series.

5.2.3 Continuous Data Series

We now summarize results from the two models that consider continuous time series: control limits and autoregression (with deviations ranging between 0.0% and 63%). We observe that the autoregressive model generally alerts more frequently than control limits. In fact, the control limits contribute to only 28% of all deviations in continuous series.

Control Limits model.

Our results show that, apart from the overlap with the autoregressive model, control limits report an additional 3% of the series as deviating. Our inspection reveals that these variables represent increasing trends over the course of the whole trace. For such variables, approaches in statistical process control commonly accept that the series should be modelled according to their regression nature only, and not with control limits. This means that these variables should be whitelisted in this model. An alternative approach would be to obtain absolute process limits (e.g., from process engineers) and enforce those limits for series control.

Autoregressive model.

Our results show that the autoregressive model has no difficulty in modelling internal process stages and counter variables. Different from alarms and commands, which occur in relation to human interaction, these variables originate from automated process behaviour with highly correlated and regular sequences of values, which are straight-forward to capture. The results indeed show that automated process logic can be consistently monitored. Of all the deviations, we only find one related to a counter variable that delayed an increment for 2 seconds. While most likely such a deviation represents just a process irregularity, it could also indicate a replay attack as performed by Stuxnet.

The remaining deviations refer to high precision measurement variables. By inspecting them more closely, we can distinguish two groups. A first group of deviations refers to variables that autoregression fails to model well, independently of the training interval. This group accounts for

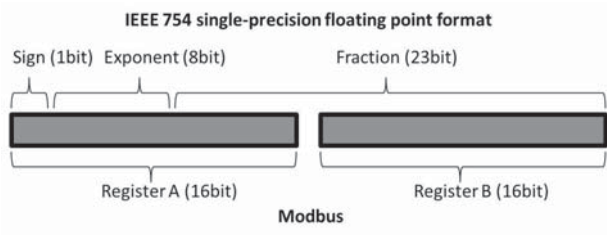


Figure 4: Representation of single precision floating point in Modbus

$\sim$ 80% of the deviations (consisting of the same variables reported consistently across all folds and batches). By sampling these variables, we find 70% exhibiting apparently random behaviour with high oscillations, while the remaining 30% remain nearly constant (or exhibit slow trends) with then sudden peaks. Referring to the project files, it turns out that they all correspond to *floating point* measurement values from the same set of field devices (e.g., measurements from devices concerning purification in PLC2a). Modbus represents floating point values with a set of registers. The vendor specification for our PLCs states that a single precision floating point is encoded according to the IEEE 754 standard [3] in *two* registers, which represent the actual value as a product of sign, exponent and mantissa. Figure 4 shows how these three components map onto a pair of registers. Taken independently, the two variables describing the behaviour of the two registers look quite different; while one variable can appear random, the second variable looks nearly constant. To address this problem, we would need to reconstruct the floating-point values as part of the data extraction phase. Unfortunately, it is technically challenging to identify the two halves of a floating point variable, as vendors use different approaches and even within the same vendor, programmers might follow different conventions due to the flat structure of the Modbus data model. Indeed, in the analysed PLC project files we observe the use of at least three different conventions: use consecutive pairs of registers with *(i)* the register with the even address as the upper register, *(ii)* the register with the odd address as the upper register, or *(iii)* for a set of variables, put all upper registers first and then all the lower registers. Working with a more structured protocol than Modbus (e.g., with IEC 60870-5-104) would help solve this problem.

The second major group of deviations concerns variables that behave differently over time. For example, one value remains nearly constant for 20 hours, then it fluctuates for 15 minutes and then it reverts back to a constant. This group of variables is the same group which was mischaracterised as constant series, as we describe in §5.2.1. The data series of these variables is not stationary in a wide sense, and thus it is generally not well suited for autoregressive modelling. The observed deviation further confirms that a unified approach can hardly address variables with different nature, origin and role. However, constituting a relatively small subset (up to 10 out of 2200 variables in one PLC), these could be treated as a special case.

5.3 Exploring Improvements

Summarizing the deviations from previous sections, we notice three distinct cases that impair our results: *(i)* characterisation issues, *(ii)* difficulties modelling attribute data, and *(iii)* dependencies amongst variables.

First, we observe that a large fraction of the reported deviations in Sections §5.2.1–5.2.3 come from the characterisation phase. A potential approach for introducing more semantics into the characterisation phase is the inclusion of knowledge contained in the project files. To explore the feasibility of this approach in practice, we translate the semantic information of the available project files into labels that define what we expect the characterisation phase to return. We assign the labels semi-automatically by constructing a table that maps keywords commonly found in the descriptions into the three categories. For example, we consider descriptions including "measurement", "counter", or "usage" to indicate variables holding continuous values; words like "command" or "alarm" suggest attribute data; "configuration" indicates a constant value. In total, we identify 24 keywords, which allow us to classify all PLC variables defined in the project files. Our analysis shows that we can cover 35% of all observed variables that way, with the main limiting factor being implicit definitions of multiple variables (e.g., PLC programmers use tailored data structures to define a range of variables by only defining the starting variable in project files). Comparing the labels with our heuristics, we see an excellent match for constant variables. However, only about half of the continuous variables match, and even less in the attribute category (around 30%). Closer inspection reveals as the main problem ambiguities in the project file that mislead the keyword-based heuristic. Generally, the descriptions are not standardised but depend on the PLC programmer, and hence keywords sometimes overlap. For example, one PLC has several fields that include the description "ControlForAlarm". Yet, we consider the keyword "control" to indicate a constant variable, and "alarm" to suggest an attribute series. While this example could be addressed easily, similar ambiguities would remain. This difficulty shows that in practice, even if project files are available, it is not straight-forward to extract meaningful semantic information from them.

Second, in §5.2.2 we point out that the current approach cannot capture the semantics or sequence of some attribute series (e.g., process alarms). However, during focus group sessions with plant engineers, we learned that alarms and commands are typically encoded in bitmaps, and we indeed find this reflected in the network traffic. For example, for a variable that the PLC project file refers to as *"various status notifications from PLC3 to server"* , we observe a series of what, at first, appears like an arbitrary set of values: 40960, 36864, 34816. However, when aligned in binary format, the values map to:

```
1010 0000 0000 0000
1001 0000 0000 0000
1000 1000 0000 0000
```

This representation reveals patterns of bits that are constant (e.g., the first bit indicates that PLC1c is active). If we integrated this structure into the characterisation step, we could refine the attribute modelling significantly. In other words, some variables require a different granularity than just their numerical value for capturing their semantics.

Third,we find distinct groups of variables that alarm at the same time. In all analysed cases, the deviations correspond to a limited set of devices with the behaviour that is *(i)* related (e.g., a peak of 9 deviating variables in PLC 2a semantically describe different aspects of only one field device, a pump) or *(ii)* identical (e.g., we see 10 ozone filters

whose flow is described by the same autoregressive model, and whose deviation occurred at the same time and thus resulted in a peak of deviations in one PLC). In either case, our preliminary analysis shows that the variables could be clustered (e.g., by performing per "point-in-time" vertical analysis) and presented as aggregated events.

6. RELATED WORK

A set of prior work focuses on understanding ICS communication patterns, showing that communication flows indeed reflect the regular and (semi-)automated character of process control systems [6].

Other efforts focus on analysing security threats in ICS. For example, some authors analyse protocol vulnerabilities [1, 4, 7], explore the lack of compliance to protocol specifications in different PLCs [9, 35], and analyse communication patterns for anomaly detection [22, 36]. However, the effects that one can find at the flow-level remain limited; detecting semantic process changes requires inspection of the application layer. Consequently, some authors propose to parse network protocols for extracting information that can highlight changes to the process environment. For example, authors perform partial protocol parsing to enumerate functionality that Modbus clients use, aiming to detect unexpected deviations in requests sent to PLCs [12], interpret events at a higher level [17], fingerprint current device configuration remotely [29, 33], and propose approaches for verifying PLC software [27, 28]. Düssel et al. [13] propose using application syntax (not *semantics*) for network-based anomaly detection, but do not further examine ICS-specific protocols.

In terms of classic IDS signatures, DigitalBond provides Snort preprocessors that add support for matching on Modbus/DNP3/EtherNetIP protocol fields [2].

McLaughlin and McDaniel in [24, 26] leverage PLC code and process descriptions to reconstruct the mapping of process variables in PLCs and thus craft efficient process attacks. In terms of analysis, this work is complementary to ours—the approach leverages PLC code while we use passive network monitoring to reconstruct process semantics. Conceptually, detecting attacks generated via SABOT [26] are within the scope of our approach. We show that key process variables (i.e., configuration settings) can be consistently monitored via our approach.

In [25] the authors present stateful policy enforcement which validates commands sent to physical devices from a PLC. While this approach can work even when the PLC gets compromised, the requirement of expert involvement is inevitable (while our approach requires little or no involvement).

Different authors analyse false data attacks against state estimation (thus targeting *indirect control* attacks) [5, 11, 23]. Similarly, yet from a network perspective, others evaluate accuracy of detectors on crafted attacks [16] and preconfigured undesirable states [15] within controlled testbeds on a set of selected process measurements. These works focus on understanding the accuracy of the modelling approach and leverage detailed specifications on the behaviour of the measurement variables in a controlled environment (e.g., including the specification of the process and state estimation).

By contrast, we perform an exploratory study to understand the behaviour of *all* process variables (i.e., both field and internal PLC variables) in a real world plant, providing insights into challenges for building automated process-aware, yet specification-agnostic, security detectors.

7. CONCLUSION

Our work ventures much deeper into ICS network traffic than prior efforts have offered. We monitor the communication of PLCs in two real-world water plants, extracting and modelling operational semantics at the level of individual process variables, to then flag unexpected deviations for manual inspection. Our motivation comes from recent *direct* process control attacks that aim to change static configuration parameters. Our approach detects these by watching for changes to variables that would normally stay constant. While that also reports legitimate changes, these generally either remain sufficiently rare—and, hence, an alert can act as a useful double-check—or amendable to whitelisting by staff if more frequent.

Going beyond static variables, we show that our approach can also detect *indirect* process control attacks by modelling the expected behaviour of non-constant variables. In practice, this proves more difficult, though, as real-world environments exhibit plenty of irregularities, semantic mismatches, and corner-cases that need care to get right. We offer a first step into that direction by focusing for now less on measuring detection rates, and more on gaining insight into conceptual capabilities and limitations. Consequently, we identify a range of reasons for our simple classifiers to fail in capturing dynamic process variables reliably, including mismatches between training and activity cycles, failure to correctly dissect a variable's data representation, and manual human intervention with the process. However, we also find that once we understand the root cause for a failure, we can often lay out a way forward to overcome that particular shortcoming (e.g., by exploiting the specific structure of a bitfield).

From a different perspective, when we derive models for thousands of real-world PLC variables, we find that indeed about 95% represent generally static configuration settings that our approach covers well. Including non-static variables, our approach manages to consistently build models of over 98% of the process variables. However, out results also suggest that in practice, due to their individual nature, one cannot expect a small set of generic classifiers to reliably capture them all equally well. We hence deem it more realistic to tailor a detector's focus to a smaller set of *high-value* variables that are *(i)* critical for the process and *(ii)* provide a complementary picture to any existing safety monitoring. As particularly interesting candidates we identify clusters of related variables that represent semantic dependencies within a process—existing safety systems do typically not consider them as a group.

Generally, we also plan to leverage further context to model individual process variables more tightly, including from more structured protocols (e.g., DNP3 and IEC 60870-5-104 define specific types for setpoints, counters, etc.) and further plant configuration files (e.g., Substation Configuration Language (SCL) files as used in IEC 61850 networks, ICS vendor project files). While generally the specific choice of variables that prove most valuable to monitor will remain site-specific, we see an economy of scale at a different level: one can formalise, and then replicate, the *procedure* of identifying and modelling suitable targets in hitherto unknown network environments.

Acknowledgments

This work has been partially supported by the European Commission through projects FP7-SEC-285477-CRISALIS and FP7-SEC-607093-PREEMPTIVE, funded by the 7th Framework Program; as well as by the U.S. National Science Foundation under grant 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.

8. REFERENCES

[1] Project basecamp. http://www.digitalbond.com/tools/basecamp/.

[2] Quickdraw SCADA IDS. http://www.digitalbond.com/tools/.

[3] IEEE Standard for Floating-Point Arithmetic. Technical report, Microprocessor Standards Committee of the IEEE Computer Society, 3 Park Avenue, New York, NY 10016-5997, USA, Aug. 2008.

[4] Common cybersecurity vulnerabilities in industrial control systems. U.S. Department of Homeland Security, 2011.

[5] S. Alajlouni and V. Rao. Anomaly Detection in Liquid Pipelines Using Modeling, Co-Simulation and Dynamical Estimation. In *Critical Infrastructure Protection VII*, volume 417 of *IFIP*, pages 111–124. Springer Berlin Heidelberg, 2013.

[6] R. R. Barbosa, R. Sadre, and A. Pras. Difficulties in modeling SCADA traffic: a comparative analysis. In *Proc. of the 13th international conference on Passive and Active Measurement*, PAM'12, pages 126–135, Berlin, Heidelberg, 2012. Springer-Verlag.

[7] C. Bellettini and J. L. Rrushi. Vulnerability analysis of SCADA protocol binaries through detection of memory access taintedness. In *Proc. 8th IEEE SMC Information Assurance Workshop*, pages 341–348. IEEE Press, 2007.

[8] G. E. P. Box and G. Jenkins. *Time Series Analysis, Forecasting and Control*. Holden-Day, Incorporated, 1990.

[9] E. Byres, D. Hoffman, and N. Kube. On shaky ground - a study of security vulnerabilities in control protocols. In *NPIC HMIT*, 2006.

[10] A. Carcano, I. N. Fovino, M. Masera, and A. Trombetta. Critical information infrastructure security. chapter Scada Malware, a Proof of Concept, pages 211–222. Springer-Verlag, Berlin, Heidelberg, 2009.

[11] A. A. Cárdenas, S. Amin, Z.-S. Lin, Y.-L. Huang, C.-Y. Huang, and S. Sastry. Attacks against process control systems: Risk assessment, detection, and response. In *Proc. of the 6th ACM Symposium on Information, Computer and Communications Security*, ASIACCS '11, pages 355–366, New York, NY, USA, 2011. ACM.

[12] S. Cheung, B. Dutertre, M. Fong, U. Lindqvist, K. Skinner, and A. Valdes. Using model-based intrusion detection for SCADA networks. In *Proceedings of the SCADA Security Scientific Symposium*. Digital Bond, 2007.

[13] P. Düssel, C. Gehl, P. Laskov, J.-U. Busser, C. Stormann, and J. Kastner. Cyber-critical infrastructure protection using real-time payload-based anomaly detection. In *Proc. of the 4th international conference on Critical information infrastructures security*, CRITIS'09, pages 85–97, Berlin, Heidelberg, 2010. Springer-Verlag.

[14] N. Falliere, L. O. Murchu, and E. Chien. W32.Stuxnet Dossier-symantec security response. [online], 2011.

[15] I. N. Fovino, A. Carcano, M. Masera, A. Trombetta, and T. Delacheze-Murel. Modbus/DNP3 state-based intrusion detection system. In *Proc. of the 2010 24th IEEE International Conference on Advanced Information Networking and Applications*, AINA '10, pages 729–736, Washington, DC, USA, 2010. IEEE Computer Society.

[16] W. Gao, T. Morris, B. Reaves, and D. Richey. On SCADA Control System Command and Response Injection and Intrusion Detection. In *eCrime Researchers Summit (eCrime)*, 2010.

[17] J. Gonzalez and M. Papa. Passive scanning in Modbus networks. In *Critical Infrastructure Protection*, volume 253 of *IFIP International Federation for Information Processing*, pages 175–187, 2007.

[18] D. Hadžiosmanović, L. Simionato, D. Bolzoni, Z. Zambon, and S. Etalle. N-Gram Against the Machine: On the Feasibility of the N-Gram Network Analysis for Binary Protocols. In *Proc. Research in Attacks, Intrusions, and Defenses*, volume 7462 of *LNCS*, pages 354–373. Springer Berlin Heidelberg, 2012.

[19] M. Hoon. Parameter Estimation of Nearly non-Stationary Autoregressive Processes. Technical report, Delft University of Technology, 1995.

[20] R. J. Hyndman and G. Athanasopoulos. Forecasting: principles and practice. An online textbook.

[21] P. International. Profinet application layer service definition, 2004. V. 1.95.

[22] O. Linda, T. Vollmer, and M. Manic. Neural network based intrusion detection system for critical infrastructures. In *Neural Networks, IJCNN International Joint Conference on*, pages 1827 –1834, june 2009.

[23] Y. Liu, P. Ning, and M. K. Reiter. False data injection attacks against state estimation in electric power grids. In *Proc. of the 16th ACM conference on Computer and communications security*, pages 21–32. ACM, 2009.

[24] S. McLaughlin. On dynamic malware payloads aimed at programmable logic controllers. In *Proc. of the 6th USENIX conference on Hot topics in security*, HotSec'11, pages 10–10, Berkeley, CA, USA, 2011. USENIX Association.

[25] S. McLaughlin. Cps: Stateful policy enforcement for control system device usage. In *Proc. of the 29th Annual Computer Security Applications Conference*, ACSAC '13, pages 109–118. ACM, 2013.

[26] S. McLaughlin and P. McDaniel. SABOT: specification-based payload generation for programmable logic controllers. In *ACM Conference on Computer and Communications Security*, pages 439–449. ACM, 2012.

[27] S. McLaughlin, Z. S., D. Pohly, and P. McDaniel. A Trusted Safety Verifier for Process Controller Code. In *In Proc. Network and Distributed System Security Symposium*. The Internet Society, 2014.

[28] S. Mohan, S. Bak, E. Betti, H. Yun, L. Sha, and M. Caccamo. S3A: Secure System Simplex Architecture for Enhanced Security of Cyber-Physical Systems. *CoRR*, abs/1202.5722, 2012.

[29] P. Oman and M. Phillips. Intrusion detection and event monitoring in SCADA networks. In *Critical Infrastructure Protection*, volume 253 of *IFIP*, pages 161–173. Springer US, 2007.

[30] R. Pang, V. Paxson, R. Sommer, and L. Peterson. binpac: A yacc for Writing Application Protocol Parsers. In *ACM Internet Measurement Conference*, 2006.

[31] V. Paxson. Bro: a system for detecting network intruders in real-time. *Computer Networks*, 31(23-24):2435–2463, 1999.

[32] Schneider Electric. *Modbus Protocol and Register Map for ION Devices*, 70022-0124-00 edition.

[33] R. Shayto, B. Porter, R. Chandia, M. Papa, and S. Shenoi. Assessing the integrity of field devices in Modbus networks. In *Critical Infrastructure Protection II*, volume 290 of *IFIP*, pages 115–128. Springer Boston, 2009.

[34] N. Sugiura. Further analysts of the data by Akaike' s information criterion and the finite corrections. *Communications in Statistics - Theory and Methods*, 7(1):13–26, 1978.

[35] A. Treytl, T. Sauter, and C. Schwaiger. Security measures for industrial fieldbus systems - state of the art and solutions for IP-based approaches. In *Proc. IEEE International Workshop on Factory Communication Systems*, pages 201 – 209, 2004.

[36] A. Valdes and S. Cheung. Communication pattern anomaly detection in process control systems. In *Proc. of International Conference on Technologies for Homeland Security*, Waltham, MA, May 11–12, 2009. IEEE.

[37] C. Vulnerabilities and Exposures. CVE-2010-4709 - heap-based buffer overflow in automated solutions Modbus/TCP Master OPC Server. http://www.cvedetails.com/cve/CVE-2010-4709/, 2011.

[38] G. B. Wetherill and D. W. Brown. *Statistical process control: theory and practice*. Chapman and Hall, London, New York, 1991.

CPS: Market Analysis of Attacks Against Demand Response in the Smart Grid

Carlos Barreto
University of Texas at Dallas
Richardson, TX, USA

Alvaro A. Cárdenas
University of Texas at Dallas
Richardson, TX, USA

Nicanor Quijano
Universidad de Los Andes
Bogotá, Colombia

Eduardo Mojica-Nava
National University of Colombia
Bogotá, Colombia

ABSTRACT

Demand response systems assume an electricity retail-market with strategic electricity consuming agents. The goal in these systems is to design load shaping mechanisms to achieve efficiency of resources and customer satisfaction. Recent research efforts have studied the impact of integrity attacks in simplified versions of the demand response problem, where neither the load consuming agents nor the adversary are strategic.

In this paper, we study the impact of integrity attacks considering strategic players (a social planner or a consumer) and a strategic attacker. We identify two types of attackers: (1) a malicious attacker who wants to damage the equipment in the power grid by producing sudden overloads, and (2) a selfish attacker that wants to defraud the system by compromising and then manipulating control (load shaping) signals. We then explore the resiliency of two different demand response systems to these fraudsters and malicious attackers. Our results provide guidelines for system operators deciding which type of demand-response system they want to implement, how to secure them, and directions for detecting these attacks.

1. INTRODUCTION

The smart grid refers to the modernization of current electric power networks to achieve better reliability, efficiency of resources, and to provide consumers more information and choices in the way they use electricity.

Research efforts have been mainly focused on the technological side of the smart grid; however, particular attention should be placed on individual consumer incentives, since individual agents (firms or people) within the smart grid are one of the enabling factors that will make the grid smart [13]. One of the particular smart grid programs that will rely on individual interactions between consumers and producers of electricity is *Demand Response* (DR) [11], a program that tries to address the retail electricity market inefficiencies.

Currently, the electricity price in the wholesale market (the bulk power grid) is updated periodically to match generation with dynamic demand between bulk power generators and bulk power consumers. When the transmission system is congested (which is the default state), Locational Marginal Prices (LMPs) are computed at each load and at each generation point to determine how much distribution utilities will pay the system operator (per Megawatt), and how much will the system operator pay the generation points. LMPs are traditionally computed every 5 to 10 minutes, but there is recent work (e.g., New York power system) for computing LMPs in real-time. In contrast, retail markets (which consist of an electric utility interacting with factories, buildings, homes, etc.) adopt static pricing schemes such as fixed and time-of-use tariffs. Under these contracts, consumers have limited incentives to adapt their electricity consumption to market conditions.

The goal of a DR program is to control consumer loads that are responsive to conditions in the electric power system, in order to achieve better efficiency of the *retail* market. Currently, the majority of DR programs are used by large commercial consumers, and companies such as EnerNOC manage DR services for large corporations and several government agencies in the U.S. Most active DR programs are designed for grid stability; however the focus of DR programs in the future is expected to be on energy efficiency. One of the primary goals of the smart grid is to make DR programs available to a much broader range of consumers [10].

1.1 Previous Work on Integrity Attacks in the Power Grid Markets

Integrity attacks (or false-data injection attacks) have been recently proposed as a way to analyze the vulnerability of cyber-physical systems in general, and electric networks in particular [18].

The area that we are interested in, is how false data injection attacks can affect the markets used in the smart grid and how false data injection can use the markets to drive the power grid to unsafe states (e.g., malicious attackers). Negrete-Pincetic et al. [21] were one of the first to study how false control signals can affect the social welfare of the electricity market. Related work by Xie et al. [30] studied how false data injection attacks can be used to defraud bulk electricity markets by modifying LMPs, and work by Liyan et al.[19] studied how false meter data in the bulk of the power grid can be used to cause the largest errors in LMP estimation.

All this work of false data analysis focuses on the bulk electricity market; however, the retail DR market has different models. In addition, it is more likely that attacks will happen in the retail market as there are many more

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.
ACSAC '14, December 08 - 12 2014, New Orleans, LA, USA
Copyright 2014 ACM 978-1-4503-3005-3/14/12 ...$15.00
http://dx.doi.org/10.1145/2664243.2664284

participants in retail with highly varying levels of trustworthiness. Finally, attributing attacks will be more difficult given a large number of participants (and thus attackers will have higher incentives for attacking retail markets than bulk-electricity markets).

Work on the impact of integrity attacks on the retail DR market were recently analyzed by Tan et al. [28], where they showed that an attacker who can modify the pricing signal sent to electricity consumers will affect the system and could cause severe oscillations of electricity load. They presented a new DR model and then experimented with two different attack models, scaling attacks and delay attacks.

While their model is an important step towards understanding the resiliency of DR programs against attacks, it has two limitations. First, Tan et al. [28] introduce a new model which has not been validated by the smart grid community. The current consensus for modeling DR problems is to incorporate market interactions of a multi-agent system where each agent has a nonlinear valuation of electricity [25, 24, 14, 26, 7, 15, 12, 9, 17]. In this paper we use more representative DR models and study their security. Furthermore, in addition to dynamic pricing DR which is the focus of study by Tan et al., we study direct-load control DR models as well.

The second limitation is that the attacks considered by Tan et al. are limited to be parametric models of the pricing signal $u(t)$; these are delay attacks $u(t-\tau)$ and scaling attacks $\alpha u(t)$. One contribution in this paper is to model a more powerful attacker that is not constrained to only two possible attack strategies, but that can select an arbitrary attack signal $\hat{u}(t)$. In addition, the previous model of the attacker is not strategic; in this paper we model a *strategic* attacker that will select an attack strategy in a principled way, and in order to achieve a specific attack goal. For example, 1) a malicious attacker will have a goal of damaging the power grid by generating sudden overload spikes, whereas 2) a selfish attacker will try to defraud DR programs.

1.2 Contributions

In this paper we address the limitations of previous work and propose new contributions.

- We introduce two attacker models against DR programs: (1) A fraudster who tries to steal electricity without trying to damage the power grid, and (2) a malicious attacker that tries to damage the power grid. Our models assume non-parametric adversary models and are therefore more powerful that adversaries considered in previous work.
- We provide a formal security analysis for the two types of adversary models for two DR programs: dynamic pricing and direct-load control, using models previously proposed by DR communities.
- We show that dynamic pricing is more resilient to fraud and malicious attacks than direct load control mechanisms.
- Previous work analyzing DR [7, 15, 12, 9, 17] consider only equilibrium points (i.e., optimal steady states); they do not consider the transient dynamics of the agents adjusting to different market conditions and learning optimal outcomes. In this paper we design an evolutionary game-theory implementation of the transient dynamics of the DR problem. This is necessary for studying malicious attacks that need to create *sudden* electricity peaks, and therefore require the study of transient dynamics to their attacks (instead of finding the steady-state equilibrium). Using this transient analysis we show how a sophisticated attacker can manipulate the market to achieve better results than naive attacks. In an effort to facilitate future research, we are making the dynamic implementation for these simulations available online as an open source BSD project [6].

2. DEMAND RESPONSE MODELS

There are two main forms of DR programs: direct load control and dynamic pricing.

2.1 Direct Load Control

Direct Load Control (DLC) [11] is a demand control system in which the utility (or a DR broker) negotiates with consumers the ability to directly control flexible loads in their homes, buildings or industries. The utility company or companies like Trilliant [29] can use remote appliance controllers to turn specific appliances on and off during peak demand periods and critical events. These remote controllers can manage water heaters, pool pumps, and air conditioners (among others) and can be programmed to respond to time-of-use tiers, critical peak pricing events, and direct load control events.

DLC has been a promising future direction for the smart grid for a variety of reasons. By controlling loads which can be modified without much impact on consumer satisfaction, we can allay many costs by shifting loads from peak demand and compensating for real-time load imbalances. For example, Pacific Gas & Electric deployed the SmartAC program in Spring 2007 [1]. Another provider of DR services has recruited over 1.25 million residential customers in DLC programs, and has deployed over 5 million DLC devices in the United States. In California, they have successfully curtailed over 25 MW of power consumption since 2007 [5]. In Hungary, for example, DLC accounts for 1600MW (25% of peak consumption).

2.2 Dynamic Pricing

Dynamic pricing programs [11] use incentives (e.g., via real-time pricing, rebates, etc.) to motivate consumers to reduce electricity consumption during peak hours. In contrast to direct-load control, consumers will be responsible for taking actions based on the incentive (control) signals; giving consumers a choice between cost and convenience. Therefore, while direct-load control is a *centralized* system, in dynamic pricing, agents make decisions in a *decentralized* way. By using dynamic pricing, utilities can create incentives for consumers to distribute their load more evenly—e.g., consume more energy when there is high wind or solar energy in the grid, and reduce consumption during peak demand times. This price-sensitive peak shaving will defer the need for grid expansion and will reduce the investments on generators that are only used for short peak demands.

2.3 Notation

We consider a population of N consumers of electricity. We divide a period of 24 hours in a set of T time intervals denoted $\tau = \{\tau_1, \ldots, \tau_T\}$; formally, we define the set τ as a partition of $[0, 24)$, where $\cup_{t\in\{1,\ldots,T\}}\tau_t = \tau$ and $\cap_{t\in\{1,\ldots,T\}}\tau_t = \varnothing$. Furthermore, we denote with q_i^t the electricity consumption of the ith user in the tth time interval. The daily electricity consumption of the ith user is represented by the vector $\boldsymbol{q}_i = [q_i^1, \ldots, q_i^T]^\top \in \mathbb{R}^T_{\geq 0}$. The population consumption at a given time t is defined by the vector $\boldsymbol{q}^t = [q_1^t, , q_2^t \ldots, q_N^t]^\top \in \mathbb{R}^N_{\geq 0}$, and the joint electricity consumption of the whole population is denoted by $\boldsymbol{q} = [\boldsymbol{q}_1^\top, \ldots, \boldsymbol{q}_N^\top]^\top$. The aggregated consumption at a given time t is defined as $||\boldsymbol{q}^t||_1 = \sum_{j=1}^N q_j^t$, where $||\cdot||_1$ is the

1-norm.

A *valuation function* $v_i^t(q_i^t)$ models the *valuation* that the ith user gives to an electricity consumption of q_i^t units in the tth time interval. Finally, let $p(\cdot) : \mathbb{R} \to \mathbb{R}$ be the price of electricity charged to consumers.

2.4 Direct Load Control Model

DLC assumes a perfect competition market where a social planner wants to maximize the social welfare of a population. This problem can be represented by the following optimization problem [16, 8, 17]:

$$\begin{aligned} \underset{\boldsymbol{q}}{\text{maximize}} \quad & \sum_{i=1}^{N} U_i(\boldsymbol{q}) = \sum_{i=1}^{N}\sum_{t=1}^{T} \left(v_i^t(q_i^t) - q_i^t p(||\boldsymbol{q}^t||_1)\right) \\ \text{subject to} \quad & q_i^t \geq 0, i = \{1, \ldots, N\}, t = \{1, \ldots, T\}, \end{aligned} \tag{1}$$

where $U_i(\boldsymbol{q})$ represents the profit (valuation of electricity consumption minus the electricity bill) of the ith customer in function of the population demand profile $\boldsymbol{q}$. Note that in this model users send their valuation for electricity to a central planner, and the central planner then decides the amount of electricity and price to charge to each agent. Here we make some assumptions on the problem characteristics in order to guarantee that the problem has a unique solution.

Assumption 1.

1. *The valuation function $v_i^t(\cdot)$ is differentiable, concave, and non-decreasing.*
2. *The price $p(\cdot)$ is differentiable, convex, and non-decreasing.*

Assumption 2. *The solution of the optimization problem in Eq. (1) is inside the feasible region, that is*

$$\frac{\partial}{\partial q_i^t} U_i(\mathbf{0}) > 0$$

Therefore, the First-Order Conditions (FOC) of this problem at the maximum, denoted by $\boldsymbol{\mu}$, are:

$$\begin{aligned} \frac{\partial}{\partial q_i^t} \sum_{i=1}^{N} U_i(\boldsymbol{q})\Big|_{\mathbf{q}=\boldsymbol{\mu}} &= \frac{\partial}{\partial q_i^t} v_i^t(q_i^t) \cdots \\ &\cdots - p(||\boldsymbol{q}^t||_1) - ||\boldsymbol{q}^t||_1 \frac{\partial}{\partial q_i^t} p(||\boldsymbol{q}^t||_1)\Big|_{\mathbf{q}=\boldsymbol{\mu}} = 0. \end{aligned} \tag{2}$$

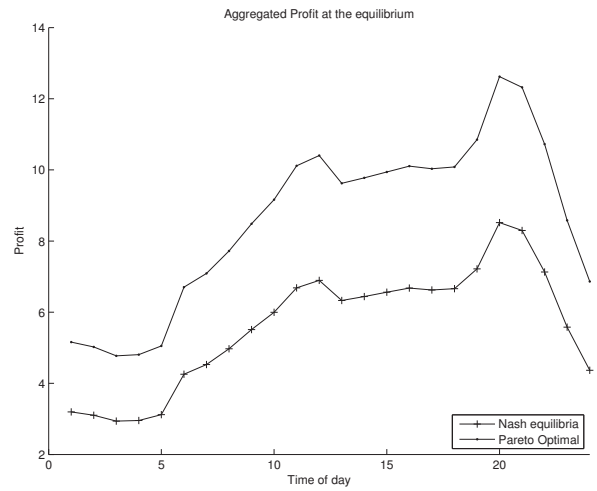
The consumption profile $\boldsymbol{\mu}$ is efficient in the sense of Pareto.

2.5 Dynamic Pricing Model

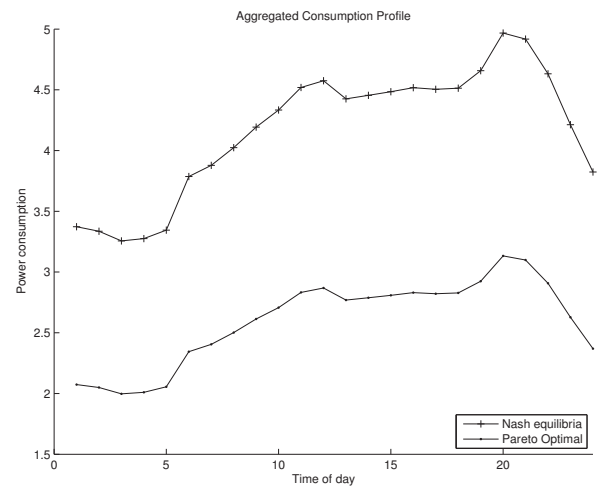
In a decentralized version of the model proposed in the previous section, each agent would need to maximize individually their utility [16]:

$$\begin{aligned} \underset{\boldsymbol{q}_i}{\text{maximize}} \quad & U_i(\boldsymbol{q}_i, \boldsymbol{q}_{-i}) = \sum_{t=1}^{T} \left(v_i^t(q_i^t) - q_i^t p(||\boldsymbol{q}^t||_1)\right) \\ \text{subject to} \quad & q_i^t \geq 0, i = \{1, \ldots, N\}, t = \{1, \ldots, T\}. \end{aligned} \tag{3}$$

The selfish actions of each individual might lead to the outcome $\boldsymbol{\xi}$, the Nash equilibrium of the game. Under Assumptions 1 and 2, we can show that the first order conditions for this selfish optimization problem guarantee a unique maximum, i.e., a unique Nash equilibrium. In addition, it is easy to show that these first order conditions (Nash equilibrium) do not match those from Eq. (2) (Pareto equilibrium). Hence, the Nash equilibrium $\boldsymbol{\xi}$ of the game is not efficient in the sense of Pareto. In general, in a strategic environment



(a) Aggregated utility.



(b) Aggregated consumption.

Figure 1: Aggregated utility and consumption of the population at the inefficient outcome (Nash equilibrium) and the optimal outcome (Pareto equilibrium).

that satisfies Assumptions 1 and 2, the outcome is inefficient and requires an external intervention, such as economic incentives [3, 4].

We now show a numerical example of how the Nash equilibrium is different than the Pareto equilibrium. We select some typical numerical values and functions previously used in the literature [23, 20]:

$$v_i^t(q_i^t) = \alpha_i^t \log(1 + q_i^t),\ \alpha_i^t > 0, \tag{4}$$

$$p(\|\boldsymbol{q}\|_1) = \beta\|\boldsymbol{q}\|_1,\ \beta > 0. \tag{5}$$

These functions satisfy Assumptions 1 and 2. In this case, we consider $T = 24$ time periods and define the valuations of each individual using consumption measurements provided by the Colombian electricity system administrator [**?**] (a detailed implementation of the simulations can be found in [4]). We can see how the Nash equilibrium produces less total utility for all parties (Fig. 1a) and produces more power consumption (Fig. 1b) than the Pareto equilibrium.

While the Nash equilibrium $\boldsymbol{\xi}$ is suboptimal, we can show,

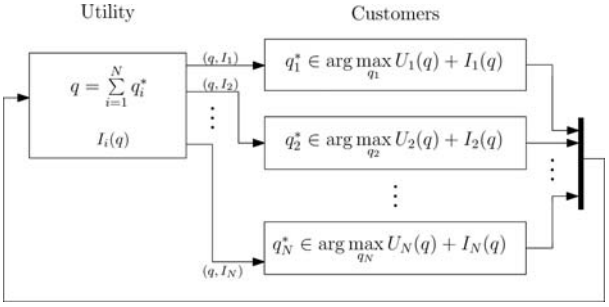


Figure 2: Dynamic pricing DR model.

however, that if we consider an added incentive to the individual cost function of each player, the Nash equilibrium of the game with incentives can be made efficient in the sense of Pareto. In order to incentivize agents to modify their behavior for the good of the population, the social planner sends them an incentive (e.g., a dynamic price signal or reward) to indirectly control their load; therefore the new cost function for the ith agent is

$$W_i(\boldsymbol{q}_i^t, \boldsymbol{q}_{-i}^t) = \sum_{t=1}^{T} \left(v_i^t(q_i^t) - q_i^t p\left(\left\|\boldsymbol{q}^t\right\|_1\right) + I_i(\boldsymbol{q}^t) \right). \quad (6)$$

where incentives are of the form:

$$I_i(\boldsymbol{q}^t) = \left\|\boldsymbol{q}_{-i}^t\right\|_1 \left(p\left(\frac{N}{N-1} \left\|\boldsymbol{q}_{-i}^t\right\|_1 \right) - p\left(\left\|\boldsymbol{q}^t\right\|_1\right) \right) \quad (7)$$

The form of this incentive is related to the price used in the Vickrey-Clarke-Groves mechanism [22] and some utility functions used in potential games [2]. Note that with these incentives it can be shown that the first-order conditions of Eq. (6) are the same to the first-order conditions of Eq. (2); and therefore, the Nash equilibrium of the system with incentives is equal to the optimal outcome of the DLC model.

This dynamic pricing model is depicted in Fig. 2, where the incentive I can represent the dynamic pricing signal. In this DR approach we consider that the utility sends a two dimensional signal to each customer, namely the total consumption and the incentive $(\boldsymbol{q}, I_i)$—in a practical implementation the utility would send the consumer the price of electricity at the current time interval: $p(\boldsymbol{q})$ instead of the consumption $\boldsymbol{q}$ but this does not affect our analysis—and each customer responds with some consumption $\boldsymbol{q}_i$. Note that the incentives modify the price paid by each user according to their relative consumption. Hence, two different users receive different incentives as long as their consumption is different. Specifically, users who introduce less externalities in the system receive larger incentives or rewards.

2.6 Transient (Evolutionary) Analysis

In the previous two sections we introduced solutions used in game theory to find equilibrium points (Pareto, Nash). These solutions provide information about the system in steady state, but overlook the trajectory followed to reach such solutions. Furthermore, while finding the steady state equilibrium is good for modeling the final outcome of an attack, we also need to consider the transient dynamics that show how consumers will behave to changes in the market and to maliciously injected signals. Thus, we can model malicious attackers that will try to create sudden electricity overloads in the system and damage power distribution equipment or produce cascading failures.

Population dynamics [4] can be used to model negotiation approaches between players of a game and are used to model the transient dynamics of the system evolution before it reaches a steady state. From the perspective of population games [27], we have a multi-population game, with the following characteristics:

- There are N populations. Each population is associated with an agent.
- The resource to be allocated in each population is the daily power consumption $||\boldsymbol{q}_i||_1$.
- The strategy of each user is the consumption at T time intervals plus the consumption q_i^{T+1} that represents a slack variable, i.e., the power not consumed in any time interval is represented as a consumption in the $(T+1)^{th}$ time interval. Hence, in each population there are $T+1$ possible strategies.
- The fitness functions are defined as the derivative of the utility function $U_i(\boldsymbol{q})$, e.g., the fitness (under normal operation) is the marginal utility, that is defined as:

$$f_i^t(\boldsymbol{q}^t) = \frac{\partial U_i^t(\boldsymbol{q}^t)}{\partial q_i^t} \quad (8)$$

  and $f_i^{T+1} = 0$.

We use replicator dynamics to solve the resource allocation problem in Eq. (3). These dynamics might be seen as a set of deterministic rules that guide the resource allocation process to find an outcome that maximizes the utility of each agent. In particular, the allocation is carried out by evaluating the convenience of consuming some resources at a given hour. An equilibrium is achieved when an agent cannot increase its profit by redistributing its resources (i.e., a Nash equilibrium).

Replicator dynamics are described by the differential equation

$$\dot{q}_i^t = q_i^t \left(f_i^t(\boldsymbol{q}^t) - \bar{f}_i^t(\boldsymbol{q}^t) \right), \quad (9)$$

where $\bar{f}_i^t(\boldsymbol{q}^t) = \sum_{t=1}^{T} q_i^t f_i^t(\boldsymbol{q}^t)$ is the average payoff the population i.

While any actor (including attackers) can change their actions arbitrarily, we consider that changes in power consumption are bounded; hence, we model smooth demand changes. A continuity notion that satisfies this requirement is Lipschitz continuity, which is also a requirement for the existence of a solution to a differential equation.

3. ADVERSARY MODEL

We assume an adversary model that compromises the central system where the control signals are computed. For 1) DLC, this means the attacker can arbitrarily send commands to curtail loads to consumers, and for 2) dynamic pricing, this means that the attacker can send arbitrary incentive price signals to consumers.

In addition, we assume two types of attackers: 1) a fraudster, whose objective is to defraud the system and pay less for electricity (a version of electricity theft, only that instead of falsifying their electricity consumption, it falsifies information sent to consumers), and 2) a malicious attacker, whose objective is to damage the electricity distribution system.

A fraudster does not necessarily want to attack the electric grid and the consumers of electricity (if the grid is down the fraudster would get no utility), but wants to exploit the system into behaving in unanticipated ways for personal gain, such as paying less electricity than others. We assume the attacker will still be charged at the correct price

for the electricity she consumes (that is the attacker has not compromised the metering system). Defrauding the utility company by compromising the control signals from DR algorithms might be even more beneficial for the attacker than compromising the meter readings, because if an attacker compromises the meter installed in its neighborhood and is detected, then the utility has evidence to attribute the attack; however, if the attack to the control signals is detected, the fraudster can still claim deniability of this attack as it is not immediately obvious who the culprit of the attack is (in particular, the attacker can mask itself in a large group of beneficiaries with a parameter γ that we will introduce later).

We define a malicious attacker as an adversary whose goal is to cause damage to the system and all their players. One practical way to achieve this is to cause the maximum sudden overload in the power distribution network, which can potentially cause blackouts because of equipment failures (e.g. burnt transformers) and circuit breakers opening.

We note that in contrast to previous work, our attack can be arbitrary, and thus can model realistic attackers not tied to pre-specified attacks (such as delays). In addition, we model two different strategic attackers that will try to achieve an objective, and will select the attack signal to achieve this objective. For example, in the dynamic pricing model the attacker will use mechanism design to generate the incentives that will achieve its own goals.

4. FRAUDSTER (SELFISH) ATTACKER

4.1 Direct Load Control

The goal of the fraudster is to achieve the best possible selfish utility by manipulating all direct-load control signals $\mathbf{q}$. We consider a scenario with full information. The best possible outcome for a fraudster with control of all control signals can be represented by the following optimization problem:

$$\begin{aligned} \underset{\boldsymbol{q}_i, \boldsymbol{q}_{-i}}{\text{maximize}} \quad & U_i(\boldsymbol{q}_i, \boldsymbol{q}_{-i}) \\ \text{subject to} \quad & q_i^t \geq 0, i = \{1, \ldots, N\}, t = \{1, \ldots, T\}. \end{aligned} \tag{10}$$

If Assumptions 1 and 2 are satisfied, then there is a unique solution to Eq. (10), denoted by q^*, that satisfies the following first order conditions:

$$\nabla U_i(\boldsymbol{q})\Big|_{\boldsymbol{q}=q^*} = 0.$$

Specifically, the gradient of the utility for an attacker (the ith agent) is

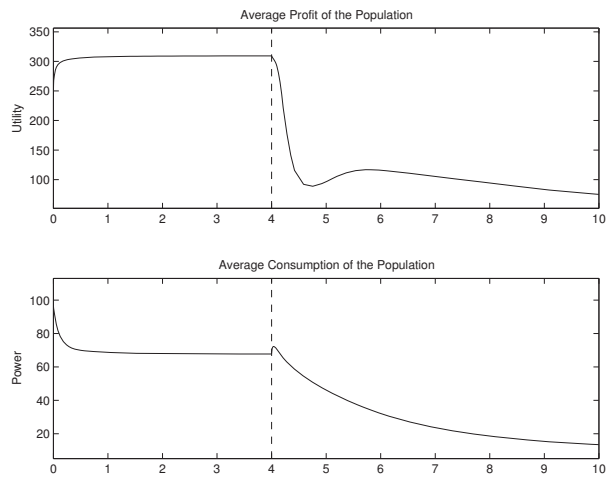
$$\frac{\partial}{\partial q_i^t} U_i(\boldsymbol{q}) = \frac{\partial}{\partial q_i^t} v_i^t(q_i^t) - p(||\boldsymbol{q}^t||_1) - q_i^t \frac{\partial}{\partial q_i^t} p(||\boldsymbol{q}^t||_1), \tag{11}$$

and the gradient for a victim j is

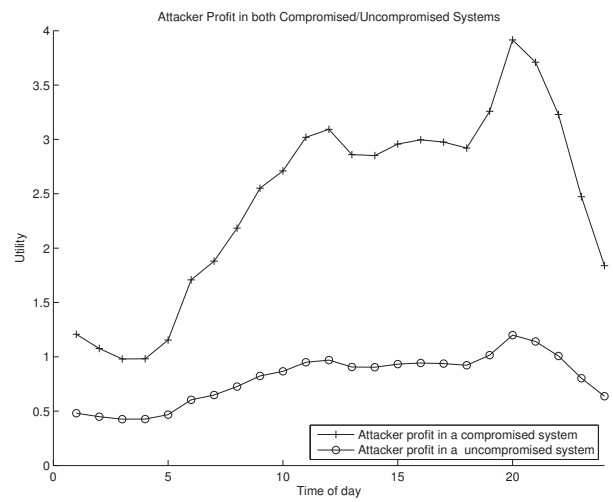
$$\frac{\partial}{\partial q_j^t} U_i(\boldsymbol{q}) = -q_i^t \frac{\partial}{\partial q_j^t} p(||\boldsymbol{q}^t||_1), \tag{12}$$

for all $i, j \in \mathcal{P}$, such that $j \neq i$ and $t \in \{1, \ldots, T\}$.

To implement these results, we consider a random state $\boldsymbol{q}$, from which the population evolves with the dynamics of a uncompromised system (see Eq. (2)). At simulation time 2, an attacker compromises the system, causing a switch in the system dynamics. The fitness functions implemented in the replicator dynamics are then changed from Eq. (2) to those defined by Eq. (11) and (12). Fig. 3a shows the evolution of both utility and consumption of the population. Fig. 3b show the final state of the attacker in a compromised and uncompromised system. From these figures we can see that the attack is successful in obtaining more electricity consumption while maximizing its utility; however, we can see that the average utility function of the population decreases while the aggregated consumption is decreased and the population is no longer in a social optimum.



(a) The utility function of the population decreases when the Fraudster launches its attack.



(b) The Fraudster can create an attack that maximizes its utility and allows it to consume more power than with no attack.

Figure 3: Fraudster attack for DLC model.

By maximizing Eq. (10), the fraudster would be the only one benefiting from the system, and the identity of the attacker might be uncovered by the central authority if it monitors the term q_i^t in the FOC of some victims–Eq. (12). Therefore, the attacker might want to mask its actions by increasing the utility of a subset $\mathcal{S}$ of the population (including the attacker); in this way, the attacker can gain plausible deniability (i.e., lack of evidence proving an allegation) as it is only one among a set of beneficiaries. In addition to make the attack more subtle (and thus harder to detect), the attacker can use a parameter λ to quantify how much of the utility function it wants to maximize compared to the utility that others will receive:

$$\begin{aligned} \underset{\boldsymbol{q}}{\text{maximize}} \quad & \lambda \sum_{h\in\mathcal{S}} U_h(\boldsymbol{q}) + \sum_{h\in\mathcal{V}} U_h(\boldsymbol{q}) \\ \text{subject to} \quad & q_i^t \geq 0, i = \{1,\ldots,N\}, t = \{1,\ldots,T\}, \end{aligned} \tag{13}$$

where $\lambda \geq 1$ represents the severity of the attack and $\mathcal{V}$ and $\mathcal{S}$ are two disjoint nonempty sets of consumers. The cardinality of each set is denoted as $N_v = |\mathcal{V}|$ and $N_S = |\mathcal{S}|$. Therefore, the FOC of the problem in Eq. (13) are:

$$\lambda \left(\frac{\partial}{\partial q_i^t} v_i^t(q_i^t) - p(||\boldsymbol{q}^t||_1) - \sum_{h\in\mathcal{S}} q_h^t \frac{\partial}{\partial q_i^t} p(||\boldsymbol{q}^t||_1) \right) \cdots$$
$$\cdots - \sum_{h\in\mathcal{V}} q_h^t \frac{\partial}{\partial q_i^t} p(||\boldsymbol{q}^t||_1) = 0. \tag{14}$$

for an agent $i \in \mathcal{S}$, and

$$\frac{\partial}{\partial q_j^t} v_j^t(q_j^t) - p(||\boldsymbol{q}^t||_1) - \sum_{h\in\mathcal{V}} q_h^t \frac{\partial}{\partial q_j^t} p(||\boldsymbol{q}^t||_1) \cdots$$
$$\cdots - \lambda \sum_{h\in\mathcal{S}} q_h^t \frac{\partial}{\partial q_j^t} p(||\boldsymbol{q}^t||_1) = 0. \tag{15}$$

for an agent (victim) $j \in \mathcal{V}$.

For illustration purposes, let us assume an homogeneous population in which agents have the same consumption preferences. Since the population is homogeneous, we know that the consumption of all the members of a set (either $\mathcal{S}$ or $\mathcal{V}$) is the same. That is, users that are in the same conditions must have the same consumption at the solution of Eq. (13), denoted by $\boldsymbol{x}$. In this case, we denote by x_s and x_v the consumption of users from either $\mathcal{S}$ or $\mathcal{V}$, respectively. We can take into account this property, as well as the form of the price function $p(z) = \beta z + b$, to obtain the following expressions from Eq (14) and (15):

$$\frac{x_s}{f_v(x_v)} = \frac{1}{\beta N_s(1+\lambda)}, \tag{16}$$

$$\frac{x_v}{f_s(x_s)} = \frac{\lambda}{\beta N_v(1+\lambda)} \tag{17}$$

where $f_s(x_s) = \dot{v}_i(x_s) - 2\beta N_s x_s - b$ and $f_v(x_v) = \dot{v}_j(x_v) - 2\beta N_v x_v - b$. Now, if we divide Eq. (16) and (17) we obtain

$$\frac{x_s f_s(x_s)}{x_v f_v(x_v)} = \frac{N_v}{\lambda N_s}. \tag{18}$$

Note that $f_\omega(x_\omega)$ is equivalent to the derivative of

$$\hat{U}_\omega(\boldsymbol{q}) = \sum_{h\in\omega} \left(v_h(q_h) - q_h p \left(\sum_{h\in\omega} q_h \right) \right) \tag{19}$$

with respect to some q_h, and evaluated at the equilibrium $\boldsymbol{x}$, for $\omega = \{\mathcal{S}, \mathcal{V}\}$. We can interpret $\hat{U}_\omega$ as the welfare of agents belonging to the subset ω. Hence, the term $x_\omega f_\omega(x_\omega)/2$ can be seen as an approximation of the utility of an isolated population in Eq. (19).

Summarizing, Eq. (18) gives information about the utility ratio between an attacker and a victim in function of λ and the number of agents in each subpopulaiton. If we consider $N_s = \gamma N$ and $N_v = N - N_s$, then Eq. (18) can be rewritten as

$$\frac{x_s f_s(x_s)}{x_v f_v(x_v)} = \frac{1}{\lambda} \frac{1-\gamma}{\gamma}. \tag{20}$$

Eq. (20) shows the attacker's utility as a function of the proportion of agents that benefit from the attack (γ). This relation holds as long as the consumption of the victims is different from zero.

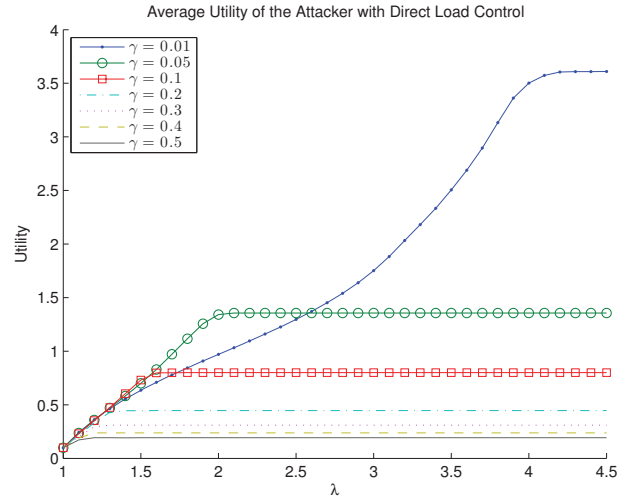


Figure 4: Utility of fraudster as a function of the parameter λ for different values of γ in the DLC model.

Fig. 4 shows the utility of the attacker for different values of λ (severity of attack) and γ (percentage of benefited consumers). Now, observe how the maximum benefit of the attacker decreases significantly with the proportion of agents benefiting from the attack (γ): in other words, the price that the attacker pays in order to maintain plausible deniabiliy if the attack is detected increases significantly. Fig. 4 also shows that the utility of the attacker does not increase indefinitely with λ but reaches a saturation point where the electricity consumption by the affected population is so low, that the attacker cannot gain any more by sending them control signals requesting lower consumption. It is interesting to note also that for small values of λ, the attacker does not gain much by being the single user benefiting from the attacks; in fact, for twice the importance of the selfish utility ($\lambda = 2$) and a population of 5% consumers benefiting form the attack ($\gamma = 0.05$), the attacker is better off than when it is the only recipient of the benefits ($\gamma = 0.01$ in a population of 100).

4.2 Dynamic Pricing

Recall that in the dynamic pricing model the central authority sends incentives I to drive the agents towards a Pareto optimal point.

$$\max_{\boldsymbol{q}_i} W_i(\boldsymbol{q}_i, \boldsymbol{q}_{-i}) = U_i(\boldsymbol{q}) + I_i(\boldsymbol{q}).$$

with respect to $\boldsymbol{q}_i$ (where I_i is the incentive signal sent by the utility to drive the system to a Pareto equilibrium). We formulate the goal of the fraudster in the DR with incentives case as the following optimization problem:

$$\underset{I, \boldsymbol{q}_i}{\text{maximize}} \quad U_i(\boldsymbol{q}) + I_i(\boldsymbol{q}), \tag{21}$$

Because in contrast to the DLC model, in the dynamic pricing model the central authority does not know the valuation functions v_i, the objective function in Eq. (21) cannot be optimized by an attacker, even if it compromises the central system. Therefore, we first use an approximation to this objective function that can be solved and then we compare how close it achieves the real objective in Eq. (21).

We now define the goal of the attacker in the dynamic pricing model to find an incentive signal I to drive the system towards the solution of Eq. (13). λ and γ are again parameters that the attacker can select.

Let $\boldsymbol{q}^s$ be the solution to Eq. (13). Recall that in this case the attacker cannot control $\boldsymbol{q}$ but instead controls I. Leveraging the theory of mechanism design we can show that an attacker can incentivize all agents to adopt $\boldsymbol{q}^s$ by sending the following false incentives:

$$I_j(\boldsymbol{q}) = \left(\sum_{h \in \mathcal{V}-j} q_h + \lambda \sum_{h \in \mathcal{S}} q_h \right) \left(\frac{N}{N-1} p(||\boldsymbol{q}_{-j}||_1) - p(||\boldsymbol{q}||_1) \right),$$

for all $j \in \mathcal{V}$ and

$$I_i(\boldsymbol{q}) = \left(\frac{1}{\lambda} \sum_{h \in \mathcal{V}} q_h + \sum_{h \in \mathcal{S}-i} q_h \right) \left(\frac{N}{N-1} p(||\boldsymbol{q}_{-i}||_1) - p(||\boldsymbol{q}||_1) \right),$$

for all $i \in \mathcal{S}$.

For λ large enough, this attack will (for practical purposes) maximize $U_i(\boldsymbol{q})$. Note that this attack does not require the valuation function of each user, but it needs to know the total consumption of either the users that benefit the attack or those who do not.

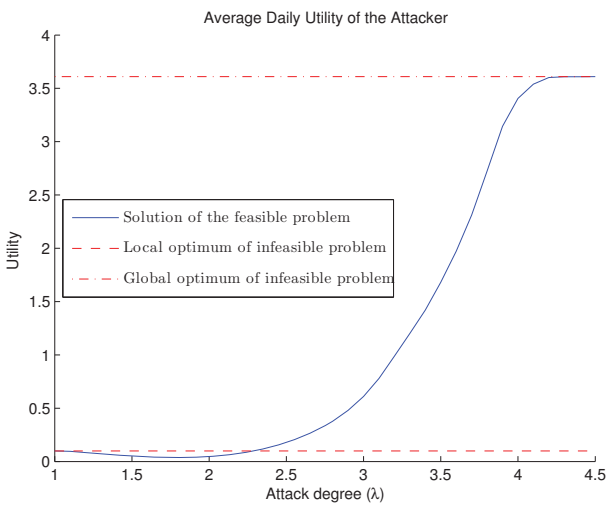


Figure 5: Impact of the attack in the fraudster's utility as a function of λ.

Now, let us study how well our approximate problem solves the ideal (but infeasible) objective function of the attacker. First, we note that Eq. (21) has multiple solutions, and only one local maximum corresponds to the Pareto optimal outcome. On the other hand, the optimization problem in Eq. (13) is feasible and has a unique solution, but might lead to a suboptimal attack.

Fig. 5 shows the utility of an attacker as a function of λ. With $\lambda = 1$ the attacker does not have any impact on the system, and the system is in the Pareto optimal outcome. As λ increases slightly, the utility of the attacker decreases as a consequence of incentives, that can be seen as taxes; however, the attack is profitable for larger values of λ, converging to the value of the infeasible problem in Eq. (10).

The attack has negative impact on the population, which is forced to reduce its consumption and consequently its utility (see Fig. 6).

A drawback of the dynamic pricing scheme is that it does not satisfy the budget balance property [22]. In other words,

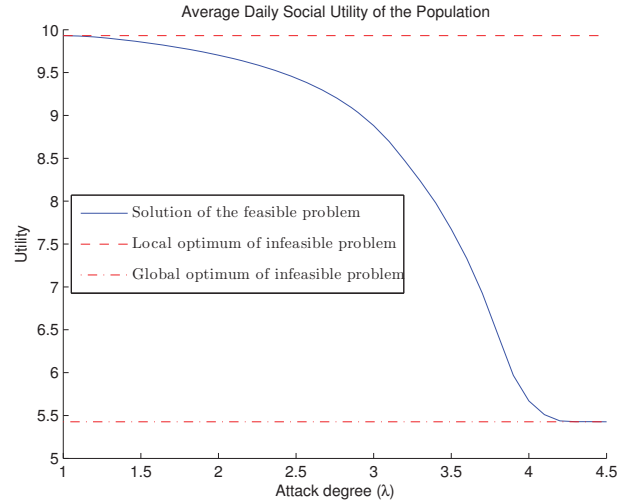


Figure 6: Impact of the attack in the social welfare utility in function of the parameter λ.

the scheme can be implemented if there is a source of external subsidies to fund part of the incentives. Note that an homogeneous population at the Pareto optimal outcome does not require external subsidies; however, if an attacker disrupts the equilibrium, the agents with low and large consumption receive positive and negative rewards, respectively. As the attack increases (λ), the demand imbalance is higher and consequently, the social planer has to increase the magnitude of the incentives. However, the taxes imposed to the attacker (due to its large consumption) are not enough to sustain the rewards to the victims and the amount of external subsidies tend to increase.

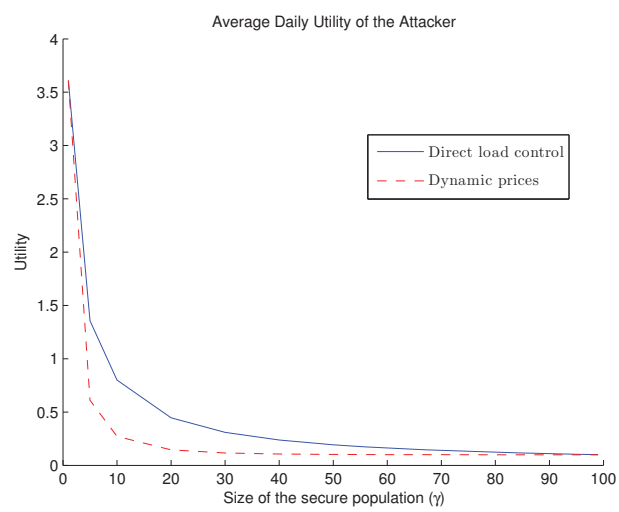


Figure 7: Fraudsters obtain more benefits from attacking DLC systems when compared to dynamic pricing.

We now compare in more detail the impact of fraudster attacks to DLC and dynamic pricing schemes. Fig. 7 and 8 show the impact of the proportion of agents that benefit from the attack γ (the ones that mask the identity of the attacker) in the utility of the attacker and the population, respectively. Note that the attacker earns less benefits as the size of the benefited population grows; however, it is

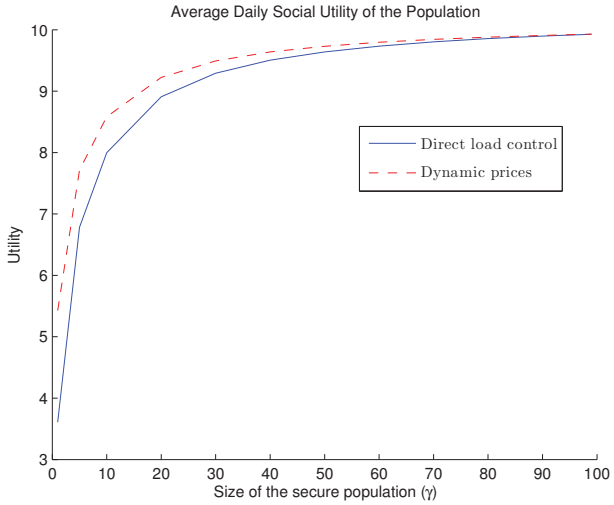


Figure 8: Consumers are better off by using dynamic pricing instead of DLC systems in the case of fraudster attacks.

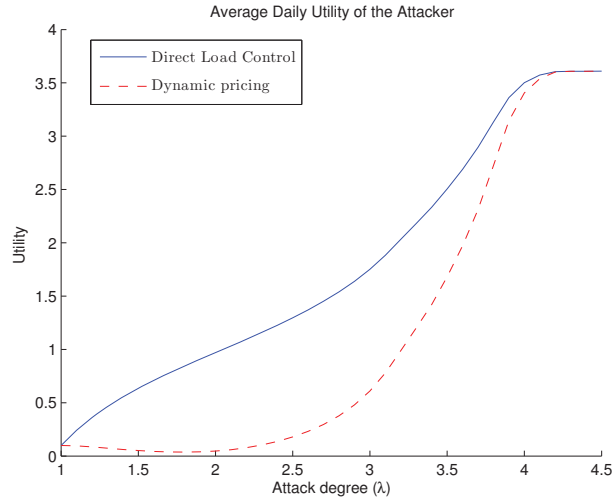


Figure 9: Fraudsters obtain more benefits from attacking DLC systems when compared to dynamic pricing.

important to note that an attacker obtains less utility in a system with dynamic prices when compared to a DLC model.

In contrast, the attack causes lower losses to all consumers with dynamic pricing because the victims receive rewards for their low consumption, as seen in Fig. 8. Furthermore, as γ tends to 1, the population welfare increases because more agents obtain benefits of the attack. Thus, with $\gamma = 1$ the population has no victims and the outcome is optimal in the sense of Pareto.

Similar behavior can be seen when we evaluate the severity of the attack λ in terms of the utility gained by the attacker in Fig. 9 and the average utility of the population in Fig. 10.

Fig. 9 shows that the attack is not optimal for a system with dynamic prices, and consequently, the attacker might do worse that the Pareto optimal outcome for $\lambda < 2.3$. This happens because the attacker is penalized due to its large consumption; however, this penalization is reduced as the other consumers are at their minimum consumption when λ is large enough.

Fig. 10 shows the social welfare under an attack in both DLC and dynamic pricing systems. The attack is less detrimental with a dynamic pricing mechanism because the social planner provides incentives to consumers that reduce their consumption. Fig. 10 also shows the total amount of incentives granted to the population; in particular, the rewards and taxes are not in balance, and consequently, the system requires external subsidies to sustain the DR program. Therefore, these subsidies can be seen as losses to the social planner caused by the attack.

We can conclude that dynamic pricing schemes are more resilient than DLC models because consumers do not have to bear all the looses; the majority of the losses are taken by the social planner (electric utility of DR company). Therefore, in terms of securing the systems, we can say that dynamic pricing schemes are incentive-compatible by placing the burden of the losses to the entity that knows better how to secure these systems, rather than placing the burden of the attack on consumers.

5. MALICIOUS ATTACKER

We now consider an attacker whose goal is to cause damage to the power system, and in order to do that it tries

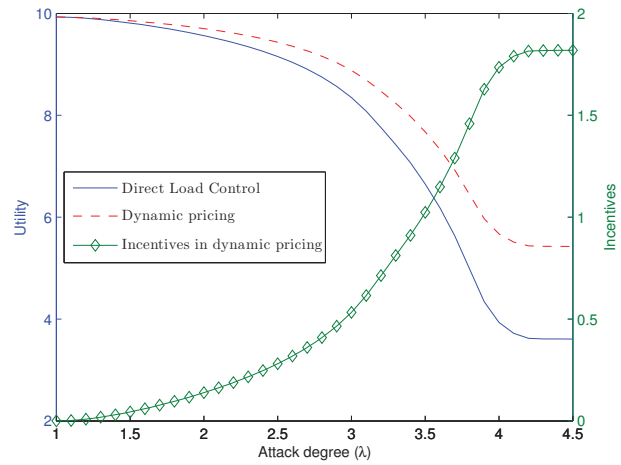


Figure 10: Impact of the attack in the social welfare utility and global incentives as a function of the attack severity λ for both the DLC and dynamic pricing schemes with $\gamma = 0.01$.

to create a sudden (unanticipated) spike in electricity consumption. Even if the distribution system has protection mechanisms, a sudden spike that has not been forecasted will be very difficult to protect against, and it might cause local blackouts (by tripping a distribution fuse or circuit breaker) or kickstarting blackstart generators.

Creating this consequence with a DLC model is straightforward, since in a DLC model the central entity controls electricity consumption signals, the attacker can directly send all electricity consumption signals to their maximum value at the same time instant. Unless consumers somehow override the control commands to increase electricity consumption, this will have a sudden and clear impact.

For the dynamic pricing model the attack strategy is not that easy, as agents make their optimization in an individual, distributed, and rational way.

Following the basic idea behind the attack to DLC models, the most intuitive (but **naive**) attack against dynamic pricing is to select a very low price for electricity (give rewards

via the incentive price signals) for electricity consumption at the time where the highest forecast of demand is set to happen (typically in the evening). Let us called the identified time for the sudden electricity consumption spike t_{attack}.

We assume that the attacker is able to compromise the incentive signal and send the following malicious incentives:

$$I_i^m(\boldsymbol{q}) = \begin{cases} I_i(\boldsymbol{q}^t) + \sigma_1 ||\boldsymbol{q}||_1 & \text{if } t = t_{attack}, \\ I_i(\boldsymbol{q}^t) & \text{otherwise,} \end{cases}$$

where $\sigma_1 > 0$ is a design parameter. This attack creates a sudden increase in the incentives. As a consequence, the consumption of the population will increase at t_{attack}. Again, because this attack takes place in the distributed optimization setting, note that the attack can be implemented without information of the valuation functions by each consumer.

We now show how the previous intuitive (but naive) attack, is not optimal for dynamic pricing schemes. In particular, dynamic pricing models consumers that can defer the consumption of electricity when market conditions are unfavorable but then return to consuming electricity en masse when market conditions change abruptly (in this case this sudden shift in market conditions is created by the malicious attacker).

A strategic attack can be implemented by carefully increasing the prices of electricity in hours previous to the attack (so consumers start deferring electricity consumption to a later time), and then, at the time where the attacker wants to create the spike of consumption, immediately lowering the prices of electricity to their lowest values. This attack causes loads that can be deferred to shift their consumption to a later time, accumulating the need to use electricity until the price is favorable.

The attack is implemented with the following incentives:

$$I_i^m(\boldsymbol{q}) = \begin{cases} I_i(\boldsymbol{q}^t) + \sigma_1 ||\boldsymbol{q}||_1 & \text{if } t = t_{attack}, \\ I_i(\boldsymbol{q}^t) - \sigma_2 ||\boldsymbol{q}||_1 & \text{if } t \in [t_a, t_b], \\ I_i(\boldsymbol{q}^t) & \text{otherwise,} \end{cases}$$

where σ_i are positive real numbers, $[t_a, t_b]$ is the time period in which the attack focuses on reducing the demand, and t_{attack} is the time at which the peak is caused, with $t_a < t_b < t_{attack}$.

In this case, the fitness function (marginal utility) used for population dynamics (to model the transient evolution of the behavior of market participants) of the ith individual is:

$$f_i^t(\boldsymbol{q}) = \begin{cases} \frac{\partial}{\partial q_i^t}\left(\sum_{h=1}^N W_h(\boldsymbol{q})\right) + \sigma_1, & \text{if } t = t_{attack}, \\ \frac{\partial}{\partial q_i^t}\left(\sum_{h=1}^N W_h(\boldsymbol{q})\right) - \sigma_2, & \text{if } t \in [t_a, t_b], \\ \frac{\partial}{\partial q_i^t}\left(\sum_{h=1}^N W_h(\boldsymbol{q})\right), & \text{otherwise,} \end{cases}$$

Note that the parameters σ_1 and σ_2 either increase or decrease the marginal utility associated with a consumption at different times of the day. Thus, the attacker can deceive users by making them believe that it is not convenient to allocate resources (consume electricity) during $[t_a, t_b]$. Also, the attacker can increase the consumption at t_{attack} by spreading the belief that users can increase their utility by using more resources at that time.

Simulations are made with $\sigma_1 = 50$, $\sigma_2 = 100$, $t_a = 0$hrs, $t_b = 17$hrs, $t_{attack} = 20$hrs. In particular, the attack time coincides with the demand peak in the Pareto optimal outcome.

Fig. 11 shows the impact of both the naive and the strategic attack after the attack is launched, that is, during the initial transition period. In particular, the naive attack succeeds in causing an increase of the demand at t_{attack}. However, the main impact of the attack is produced only during t_{attack}. On the other hand, the strategic attack achieves a greater peak by causing demand reduction prior to the attack. Roughly speaking, the strategic attack sets conditions so that the population has more resources to consume when prices are low.

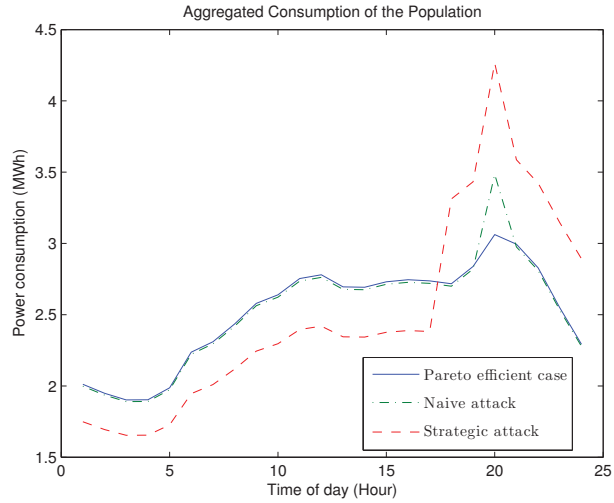


Figure 11: Impact of a malicious attack on the populaiton demand for two different attacks 1) attack on a single hour and 2) coordinated attack on various hours of the day.

6. CONCLUSIONS AND FUTURE WORK

We have introduced two new attack models for DR programs. In contrast to previous work, our attackers are strategic adversaries with clearly defined goals (objective functions), furthermore we proved the optimality of the fraud strategies for attacks against DR with DLC, and proved an optimality property of the fraudster attack against DR with dynamic pricing. We also showed how dynamic pricing is more resilient to attacks than DLC mechanisms.

In addition we introduced population dynamics to model the transient behavior of DR systems and in particular, created a model with consumers that can defer the consumption of electricity based on incentives. We showed how this behavior can be exploited by malicious attackers to cause a sudden spike in electricity load. We introduced two threat scenarios and showed that a strategic attack can perform better than the initial intuitive attacks. In practice we believe that this is the worst possible attack to the power distribution system as utilities will not be able to forecast (and therefore plan contingencies to) these changes orchestrated by attackers.

The main goal of this work was to understand the vulnerability of DR systems to market manipulators with access to the control signals sent by the central authority to consumers. In future work we plan to study mechanisms to improve the security of these systems and to minimize the effects of attacks.

One interesting problem would be to design anomaly detection schemes to detect fraudsters (depending on the parameters γ and λ), and then evaluate them against fraudsters that will try to maximize their gains while avoiding detection.

While our DR models are an improvement over previous work studying the security of DR systems; the accuracy of our analysis for practical applications will still depend on how well our DR models match real deployed systems. As

we continue to deploy trial DR systems around the world we will obtain more data on several properties of the models such as the average elasticity of electricity consumption and how much can incentives control the overall electricity load.

Acknowledgments

This work was supported in part by NIST award 70NANB-14H236 from the U.S. Department of Commerce.

7. REFERENCES

[1] Alexander, M., Agnew, K., Goldberg, M.: New approaches to residential direct load control in california. In: 2008 ACEEE Summer Study on Energy Efficiency in Buildings (2008)

[2] Arslan, G., Marden, J.R., Shamma, J.S.: Autonomous vehicle-target assignment: A game-theoretical formulation. Journal of Dynamic Systems Measurement and Control 129(5), 584 (2007)

[3] Barreto, C., Mojica-Nava, E., Quijano, N.: Design of mechanisms for demand response programs. In: Proceedings of the 2013 IEEE 52nd Annual Conference on Decision and Control (CDC). pp. 1828–1833 (2013)

[4] Barreto, C., Mojica-Nava, E., Quijano, N.: Incentives-based mechanism for efficient demand response programs. arXiv preprint arXiv:1408.5366 (2014)

[5] California Energy Commission: Docket No. 13-IEP-1F: Increasing demand response capabilities in California (2013)

[6] Carlos Barreto: Population dynamics Toolbox (PDToolbox) (2014), `https://github.com/carlobar/PDToolbox_matlab`

[7] Chen, L., Li, N., Low, S.H., Doyle, J.C.: Two market models for demand response in power networks. In: Smart Grid Communications (SmartGridComm), 2010 First IEEE International Conference on. pp. 397–402. IEEE (2010)

[8] Chen, L., Li, N., Low, S.H., Doyle, J.C.: Two market models for demand response in power networks. In: Smart Grid Communications (SmartGridComm), 2010 First IEEE International Conference on. pp. 397–402. IEEE (2010)

[9] Fahrioglu, M., Alvarado, F.L.: Designing cost effective demand management contracts using game theory. In: Power Engineering Society 1999 Winter Meeting, IEEE. vol. 1, pp. 427–432. IEEE (1998)

[10] Federal Energy Regulatory Commission: 2011 assessment of demand response and advanced metering (November 2011)

[11] FERC: FERC Staff Issue Assessment of Demand Response and Advanced Metering (October 2013)

[12] Gellings, C.W.: The smart grid: enabling energy efficiency and demand response. The Fairmont Press, Inc. (2009)

[13] Honebein, P.C., Cammarano, R.F., Boice, C.: Building a social roadmap for the smart grid. The Electricity Journal 24(4), 78–85 (2011)

[14] Huang, L., Walrand, J., Ramchandran, K.: Optimal smart grid tariffs. In: Information Theory and Applications Workshop (ITA), 2012. pp. 212–220. IEEE (2012)

[15] Ibars, C., Navarro, M., Giupponi, L.: Distributed demand management in smart grid with a congestion game. In: Smart Grid Communications (SmartGridComm), 2010 First IEEE International Conference on. pp. 495–500. IEEE (2010)

[16] Johari, R., Tsitsiklis, J.N.: A scalable network resource allocation mechanism with bounded efficiency loss. Selected Areas in Communications, IEEE Journal on 24(5), 992–999 (2006)

[17] Li, N., Chen, L., Low, S.H.: Optimal demand response based on utility maximization in power networks. In: Power and Energy Society General Meeting, 2011 IEEE. pp. 1–8. IEEE (2011)

[18] Liu, Y., Reiter, M.K., Ning, P.: False data injection attacks against state estimation in electric power grids. In: CCS '09: Proceedings of the 16th ACM conference on Computer and communications security. pp. 21–32. ACM, New York, NY, USA (2009)

[19] Liyan, J., Thomas, R.J., Tong, L.: Impacts of malicious data on real-time price of electricity market operations. In: 45th Hawaii International Conference on System Sciences. pp. pp.1907–1914 (January 2012)

[20] Mohsenian-Rad, A., Wong, V., Jatskevich, J., Schober, R., Leon-Garcia, A.: Autonomous demand-side management based on game-theoretic energy consumption scheduling for the future smart grid. IEEE Transactions on Smart Grid 1(3), 320–331 (dec 2010)

[21] Negrete-Pincetic, M., Yoshida, F., Gross, G.: Towards quantifying the impacts of cyber attacks in the competitive electricity market environment. In: 2009 IEEE PowerTech (June 2009)

[22] Nisan, N., Roughgarden, T., Tardos, É., Vazirani, V.V.: Algorithmic Game Theory. Cambridge University Press, 32 Avenue of the Americas, New York, NY 10013-2473, USA (2007)

[23] Roozbehani, M., Rinehart, M., Dahleh, M., Mitter, S., Obradovic, D., Mangesius, H.: Analysis of competitive electricity markets under a new model of real-time retail pricing. In: Energy Market (EEM), 2011 8th International Conference on the European. pp. 250–255 (may 2011)

[24] Roozbehani, M., Dahleh, M., Mitter, S.: Dynamic Pricing and Stabilization of Supply and Demand in Modern Electric Power Grids. In: First IEEE Smart Grid Commnunications Conference (SmartGridComm) (October 2010)

[25] Roozbehani, M., Dahleh, M.A., Mitter, S.K.: Volatility of power grids under real-time pricing. Power Systems, IEEE Transactions on 27(4), 1926–1940 (2012)

[26] Samadi, P., Schober, R., Wong, V.W.: Optimal energy consumption scheduling using mechanism design for the future smart grid. In: Smart Grid Communications (SmartGridComm), 2011 IEEE International Conference on. pp. 369–374. IEEE (2011)

[27] Sandholm, W.H.: Population Games and Evolutionary Dynamics (Economic Learning and Social Evolution). The MIT Press (2011)

[28] Tan, R., Krishna, V.B., Yau, D.K., Kalbarczyk, Z.: Impact of integrity attacks on real-time pricing in smart grids. In: ACM Conference on Computer and Communications Security (CCS 2013) (2013)

[29] Trilliant: Direct load control (2014), `http://trilliantinc.com/solutions/consumer/direct-load-control`

[30] Xie, L., Mo, Y., Sinopoli, B.: False Data Injection Attacks in Electricity Markets. In: First IEEE Smart Grid Commnunications Conference (SmartGridComm) (October 2010)

CPS: Driving Cyber-Physical Systems to Unsafe Operating Conditions by Timing DoS Attacks on Sensor Signals

Marina Krotofil
Hamburg University of Technology
Hamburg, Germany

Alvaro A. Cárdenas
University of Texas in Dallas
Richardson, TX 75080, USA

Bradley Manning
Autolive GmbH
Hamburg, Germany

Jason Larsen
IOActive, Inc.
Seattle, WA 98104, USA

ABSTRACT

DoS attacks on sensor measurements used for industrial control can cause the controller of the process to use *stale data*. If the DoS attack is not timed properly, the use of stale data by the controller will have limited impact on the process; however, if the attacker is able to launch the DoS attack at the correct time, the use of stale data can cause the controller to drive the system to an unsafe state.

Understanding the timing parameters of the physical processes does not only allow an attacker to construct a successful attack but also to maximize its impact (damage to the system). In this paper we use Tennessee Eastman challenge process to study an attacker that has to identify (in real-time) the optimal timing to launch a DoS attack. The choice of time to begin an attack is forward-looking, requiring the attacker to consider each opportunity against the possibility of a better opportunity in the future, and this lends itself to the theory of optimal stopping problems. In particular we study the applicability of the Best Choice Problem (also known as the Secretary Problem), quickest change detection, and statistical process outliers. Our analysis can be used to identify specific sensor measurements that need to be protected, and the time that security or safety teams required to respond to attacks, before they cause major damage.

Keywords

Cyber-physical systems, DoS attacks, optimal stopping problems, CUSUM, Tennessee Eastman process

1. INTRODUCTION

While compromising or disrupting devices or communication channels used to sense or control a physical system is a necessary requirement to attacks aimed at disrupting the physical process, the damage from the attack will be limited if the attacker is unable of manipulating the control system in a way to achieve her desired outcome in the physical world. After all, breaking into a system is not the same as breaking a system. In order to achieve a desired impact on a control system (like Stuxnet [14]), the attacker needs to assess how her attack will perform at the regulatory control level. Launching such an attack requires a different body of knowledge from the one used in IT security. In particular, attackers need to know how the physical process is controlled, and that includes knowledge of failure conditions of the equipment [15], control principles [30], knowledge of process behavior [19], and signal processing, etc.

In this paper we consider an attacker that can read a sensor signal for a given process variable, and then has to decide on a time to launch a DoS attack in order to "freeze" a certain process value above or below a set point in the controller's memory. Most Programmable Logic Controllers (PLC) operate in a scan cycle architecture. During each cycle logic in the PLC uses the last saved input buffers obtained from sensor measurements to issue control command to the actuators. If input buffers are not updated because of the DoS attack, the last measurement received will be the one used by the PLC in each subsequent scan cycle. By doing so the attacker deceives the controller about the current state of the process and can cause compensating reactions which may bring the process into a state desired by the attacker, e.g. an unsafe state.

Typical sensor signals in process control either fluctuate around the set point or follow the dynamic changes in the process. In both cases the process variable exhibits a time series of low and high peaks. In order to move a process into an unsafe state, the attacker should aim to freeze the sensor measurement at one of the optimal values of the process variable (low or high). However, because the attacker has to make the decision in real-time—where she needs to consider each opportunity against the possibility of a better opportunity in the future—the task the assaulter needs to address becomes a sequential decision problem where the attacker is presented with possible candidates at each time step, and has to make a decision immediately whether to launch an attack at this stage or not.

We formulate this challenge as an *optimal stopping time* problem for the attacker. In particular, we formulate the problem as a Best Choice Problem—also known as the Secretary Problem (SP)–in which the adversary is presented

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ACSAC '14, December 08 - 12 2014, New Orleans, LA, USA
Copyright 2014 ACM 978-1-4503-3005-3/14/12
http://dx.doi.org/10.1145/2664243.2664290 ...$15.00.

with a time series of system states obtained by sensors and has to decide on the optimal time to attack based on current and past sensor measurements. In this paper we show that the problem the attacker has to solve is a non-trivial task in many practical situations as sensor measurements can be noisy and have sudden fluctuations. Using a plant-wide chemical control process simulation, we identify different types of sensor signals that can make the selection problem more challenging. We then explore different stopping rules and compare their effectiveness for selecting optimal sensor signal samples. Our results show that different stopping rules will be selected by different types of adversaries: risk-taking attackers (those who prefer to wait for a better opportunity at the risk of waiting for too long to launch an attack) and risk-averse adversaries (those who prefer to attack as soon as a reasonable attack opportunity emerges).

Our new threat assessment model can be used to measure the impact of the DoS attacks on sensor signals and to inform asset owners about the need to prioritize them for protection. In addition, our analysis can be used to identify the time interval (between the beginning of the DoS attack and the response from the security team) before the plant suffers either an emergency shutdown (if safety systems are in place) or a safety accident (if the attacker has disrupted safety systems as well).

2. TIMING AND CYBER-PHYSICAL SECURITY

In this paper we focus on Process Control Systems (PCS) which is a general term used to denote architectures, mechanisms and algorithms which enable processing of physical substances or manufacturing end products. Process industries include assembly lines, water treatment, pharmaceutical, food processing and other industries. In the past few decades plants have undergone tremendous modernization; technology became an enabler of efficiency but also a source of problems. What used to be a panel of relays is now an embedded computer, and what used to be a simple analog sensor is now an IP-enabled smart transmitter [24] with multiple wired and wireless communication modes, a large number of configuration possibilities, and even a web-server so that maintenance staff can calibrate and setup the device without approaching it.

Cyber-abuse in the IT domain do not generally depend on timing aspects. In certain instances such as during race conditions, Time-of-Check to Time-of-Use vulnerabilities, or cross-site scripting attacks that rely on getting access to session cookies before they expire, the attacker needs to make sure that their attack occurs within a tight window of time. In cyber-physical systems, however, timing takes an even more important role as the physical state of the system changes continuously over time, and during the system evolution in time, some states might be more vulnerable to attacks than others. Timing also characterizes the vulnerability of a system; e.g., it may take minutes for a chemical reactor to burst [31], hours to heat a tank of water or burn out a motor, and days to destroy centrifuges [14].Understanding the timing parameters of the physical processes does not only allow an attacker to construct a successful attack but also to maximize its impact.

The dynamic response of a processes variable changing from one state to another can be described with a simple model consisting of process gain, dead time, and time constant. The process gain describes how much the process will

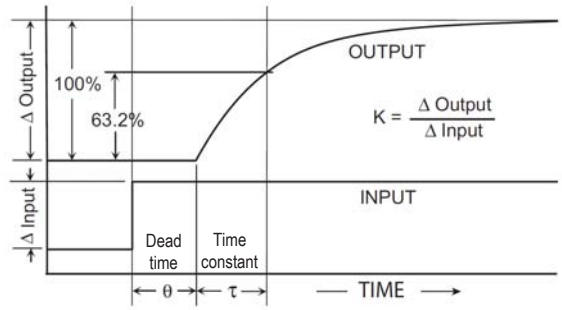


Figure 1: Time constants in process control, based on [30]

respond to a change in controller output, while the dead time and time constant describes how quickly the process will respond (Fig. 1). Precisely, dead time describes how long it takes before a process begins to respond to a change in controller output, and the time constant describes how fast the process responds once it has begun changing. Controlling the processes with large time constants is a challenging task causing operators' stress and fatigue [21]. The described timing parameters are not only important for the design of a control algorithm but also for the attacker to design an effective attack.

2.1 Adversary Model

The adversary's goal is to cause tangible impact on the process. In the physical domain, the attacker can either tamper with the sensor signals or modify the manipulated variables issued by the controller. In this work we limit our study to sensor signals. In particular we assume an attacker who aims to bring the process into an unsafe state by deceiving a controller about the current state of the process and thus forcing it to take harmful compensating actions. To do so the attacker can force the controller to believe that a process variable is below or above its set point. One way to achieve this is to forge the process variable by the means of a false-data injection attack. If the communication channel between a sensor and a controller has integrity protections (e.g., message authentication codes) and the attacker does not have key material, the attacker might opt to jam communication channel to prevent the controller from receiving process measurement updates. In the context of this paper we call this type of attack a DoS attack on a sensor signal.

As a rule, controllers store sensor signals in dedicated memory registers which are updated whenever a new value arrives. During the DoS attack, the input register assigned to storing measurements of a particular sensor will not be overwritten by fresh values. Therefore, the last process value which reached the controller before the attack, will be used by the PLC for the duration of the DoS attack. As a result, the controller will generate control commands based on the last received reading. A DoS attack when compared to a false-data injection or integrity attack is different in that the adversary does not have direct influence on the "attack value". Instead an adversary can take advantage of the timing parameters of an attack, such as the starting time t_a of the attack and the duration T_a.

The impact of the attacks on PCS are sensitive to the specific state of the system, in particular, attacks might only be effective if the process variable is above (or below) a certain threshold. The higher the attack process variable is chosen beyond the threshold, the higher the impact. Moreover, since DoS attacks are easy to detect, one of the goals

of the attacker is to achieve its disruption objective as soon as possible after the attack is launched. Therefore, the attacker should aim at launching a DoS attack at the time the process variable of interest reaches its local maximum or minimum (a more vulnerable state).

The attacker faces the following problem: given a time-series that exhibits a sequence of peaks and valleys of different amplitude, she has to select one of the peaks to launch a DoS attack in real time. If the attacker strikes too soon, she might lose the opportunity of having a higher impact on the system if she had waited longer (i.e., if the process variable would reach higher value later in the future). However, if the attacker waits for too long, the process variable might not reach a more vulnerable state than previously observed, and the adversary might miss the opportunity to cause maximal damage, or even have the implanted attack tools detected before they have the chance to launch the attack.

The problem of selecting a good time to attack can be framed as an *optimal stopping problem*. These class of problems are concerned with the challenge of choosing the time to take a particular action based on sequentially observed random variables in order to maximize an expected payoff. Such class of optimal stopping decision tasks in which the binary decision to either stop or continue the search depends only on relative ranks is modeled as Best Choice Problem, also known as the Secretary Problem [9].

2.2 Susceptibility of Control Equipment to Stale Data Attack

Almost every device in process control equipment needs to deal with stale data but it is rarely documented in open literature. In many environments the bandwidth required to report the current state of every point (field devices) in the system exceeds the capacity of the communications channel. This is particularly prevalent when the data is communicated over long-haul serial communications. Open-source and proprietary protocols both describe themselves as "report by exception" meaning the data is reported to the server only when it has changed significantly. In many cases the amount of change required for the process value to be reported is configurable but the exact rules may still be opaque.

Extensive testing of Industrial Control Systems was performed at the Idaho National Labs and other facilities [6]. During those tests a number of scenarios were found where stale data could be maintained in a system for indefinite periods of time. No general technique is known that is applicable to every device so we describe some of the implementation details that have previously allowed for stale data.

The most common root cause of stale data is when the protocol stack was implemented in a stateless manner. The master maintains a set of timers for the data tied to individual data points or a set of data points assigned to a class. When data is received by exception the timer is restarted. If the data changes often enough, the master never sends a poll request. If the timer expires, the point is placed into a point list to be requested during the next poll. If many cases, the timer is reset when the poll request is sent with the assumption that the point value will be returned in the next reply. In other cases, the timer is only reset upon a successful receipt of the updated data.

The most common finding was if the TCP session was simply maintained without transferring any data, the sensor was never reported as offline. The second most common finding was that the protocol lacked replay protection and old messages could simply be resent. Also more complex attacks have been shown possible. Thus, in many implementations the server listens on both the UDP and TCP ports. Messages received via TCP, UDP, or serial communication were routed to the same processing code without checking which communications mechanism sent the message. An exception report could be sent via the UDP port but it would be merged with the ones received via an existing TCP session. In case the sequence numbers are used to detect missing and retransmitted data, empty acknowledgments with higher sequence numbers could be sent via UDP to increment the internal sequence number of the controller. When a legitimate message arrived at the TCP port, it would be discarded as retransmissions.

Furthermore, a number of examples can be formed where interfering with a data channel is possible but fully viewing and manipulating the data channel is not. The most common case is networking equipment where the attacker only has access to the administrative interface but not to the data plane. For example, in common routers there is a "fast path" where packets are routed in custom designed application-specific integrated circuits (ASIC) and the "slow path" where packets are transferred to the CPU. Access to the administrative interface can give code execution on the device, but only packets transiting the slow path can be observed and manipulated. If all packets are requested by the slow path, the CPU is saturated resulting in a DOS on the communications link. If packets are sampled periodically, they can be examined and with that allowing an attacker to perform a DoS attack at the opportune time.

3. BEST CHOICE AND OPTIMAL STOPPING PROBLEMS

In this section we introduce the theoretical background used in our formulation.

3.1 Best Choice Problem

In the standard version of the Best Choice Problem, a finite and known number n of items (or alternatives) are presented to a decision maker (DM) sequentially, one a time, in a random order. Time is discrete. At any period the DM is able to rank all the items that have been observed in terms of their desirability or quality. For each item inspected the DM must either accept it, in which case the search process is terminated, or reject it and then the next item in the random order is presented and the DM faces the same problem as before. The DM's objective is to maximize the probability of selecting the best item of all the n items available.

The classical Best Choice Problem is formulated in terms of a hiring manager identifying the best secretary (the Secretary Problem) it has interviewed among all applicants. This problem is formulated in terms of six assumptions:

1. There is only one position available.
2. The number of applicants, n, is finite and known to the DM.
3. The n applicants are interviewed sequentially, one at a time, in a random order; consequently, each of the $n!$ orders is equally likely.
4. The DM can rank all the n applicants from best to worst without ties. The decision to either accept or

reject an applicant in a given period is based only on the relative ranks of those applicants interviewed so far.

5. Once rejected, an applicant cannot later be recalled.

6. The DM is satisfied with nothing but the best. (Her payoff is 1, if she chooses the best of the n applicants, and 0 otherwise.)

Note that an applicant is accepted only if it is relatively best among those already observed. A relatively best applicant is called a *candidate*. The stopping rule suggested by the Best Choice Problem theory is the following: do not make any offer to the first n/e candidates (where e is the base of the natural logarithm) and after that, make an offer to the first candidate whose value exceeds the value of all candidates seen so far (or proceed to the last applicant if this never occurs, such case is called non-selection). In other words, the algorithm starts with a *learning phase* in which the DM sees n/e candidates and sets an *aspiration level* equal to the highest value seen in the learning phase. After that, the DM hires the first candidate that exceeds the aspiration level. The main result of the best choice problem states that the optimal stopping rule can select the best candidate with at least $(1/e)$ probability.

It has been recognized that the classical assumptions place more constraints on the observation and selection than would generally apply in practice [12]. Relaxing one or more assumptions for a more realistic formulation of the standard assumptions has attracted attention in the research community. In this paper we consider the classical solution, and a recent result that assumes the order in which the candidates arrive is not completely random, but has a probability distribution satisfying a hazard rate condition [20]. This assumption is commonly used in standard engineering applications and states that given that the value of a candidate is not less than y, the likelihood that it is equal to y increases as y increases. As an example, Gaussian, uniform, and exponential distributions satisfy this property. Under these assumptions it was shown that the learning period falls from n/e to $n/log(n)$, meaning that it is enough to observe a much smaller number of candidates to set the optimal aspiration level and have a similar probability of success. Since the probability of detecting an intrusion increases with time, having a shorter learning phase is beneficial to the attacker.

Fig. 2 illustrates formulation of the Secretary Problem and its solution in the context of an arbitrary sensor signal. We call learning phase as an *observation window* and we refer to the selection phase as an *attack window*. Notice that in this case the maximum selected is lower than the overall maximum in the attack window.

3.2 Peak Detection

In the time series of the physical phenomenon, each time sample X_i is heavily correlated with the next sample X_{i+1}. Thus, if a process variable (e.g. temperature) is increasing, it cannot drop radically in the next time instance. As follows from the SP solution, upon completion of the learning phase the attacker should select the first sample, whose value exceeds aspiration level. By doing so the attacker can miss the opportunity to select an even higher value as in case of an upwards trend where the process measurement will keep increasing until it reaches its local peak. Therefore, in contrast the static choice rule, the attacker may incorporate expectations about the future into her decision process.

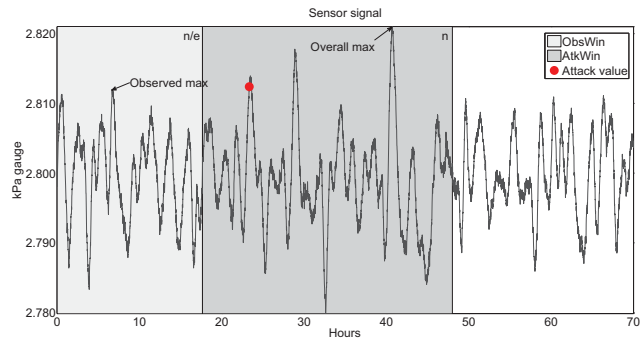


Figure 2: Illustration of the Secretary Problem solution applied to a sensor signal

The problem of identifying a signal peak is exacerbated by the fact that process variables are *noisy* and therefore an upward trend might be followed by a quick drop, followed again by an even higher gain. We propose two algorithms for peak detection.

3.2.1 Forward Looking Search

In our initial approach we add a low-pass filter to the signal to smooth out short-term fluctuations and highlight longer-term trends. In this case the choice between stopping or continuing to search at sample X_i is determined by the difference between the stopping value and the continuation value X_{i+1}. This allows a peak to be identified as soon as a downward trend in a smoothed signal is detected (e.g., three consecutive measurement drops). The optimal smoothing interval depends on the sampling frequency of the signal.

3.2.2 Change Detection

A more sound approach to peak detection is quickest change detection theory. Change detection tests are statistical techniques that allow identification of a possible drift, abrupt, and and sudden changes in data series at an unknown time. Cumulative sum (CUSUM) is one of the most commonly used algorithms for change detection problems. Most optimal stopping rules are parametric and require a priory knowledge about the statistics of the signal and the size of the expected shift. However, in several practical cases the underlying distribution and the magnitude of shifts are both unknown. In such cases the non-parametric scheme (NCUSUM) is a more suitable choice. Let the sensor time series be represented by a series X_i. Our goal is to identify the peak of this time series as soon as possible. To do this we define a new time series S_i. In particular we use two NCUSUM statistics for each signal, S_i^+ to detect a local maximum in the signal, and S_i^- to detect a local minimum. The NCUSUM is initialized with $S_0^+ = 0$ and $S_0^- = 0$ and updates as follows:

$$S_i^+ = \max(0, |X_{i-1} - X_i| + S_{i-1}^+),$$
$$S_i^- = \max(0, |X_i - X_{i-1}| + S_{i-1}^-),$$

An alarm is triggered whenever $S_i > h$, where h is a threshold that can be used as a parameter to control the tradeoff between the rate of false alarms and the delay for detecting the local maximum or minimum of the signal. Although NCUSUM statistic is general and can be applied to any signal, the threshold h has to be learned for specific types of

sensor signals.

3.3 Heuristic Algorithm

Generally speaking, the SP allows an attacker to select the best possible "outlier" sample (due to noise) in the time series. Although SP is based on a sound mathematical theory, one of the disadvantages of the SP optimal solution is the high number of non-selections (the last sample in the time series is taken because the attacker ran out of time). To avoid non-selections we propose an alternative heuristic solution which we call the Outlier Test (OT).

In process control it is not uncommon to assume process measurement noise follows a Gaussian distribution, which is characterized by the mean value μ and standard deviation σ. These parameters can be estimated for each signal (during the observation period), and then, during the attack window, the attacker can use this knowledge to detect outliers. According to the "three-sigma rule" in a normal distribution nearly all values (0,9973%) lie within three standard deviations of the mean. In this case, the attacker will select the first sample in the attack window such that $X_i > \mu + 3\sigma$ or $X_i < \mu - 3\sigma$. Alternatively these thresholds can be lowered to values that may be achieved faster if the attacker has a time deadline. A standard score curve can be used to determine the optimal OT threshold h.

4. SIMULATION SETUP

One of the challenges of cyber-physical security research is the lack of large-scale test beds to allow the study of complex attacks and their effects on physical processes. To mitigate this problem researches can leverage simulation models of realistic industrial plants [7, 2, 5] which have been developed for the process control community to focus on issues important to the industry and to allow comparison of research results. These models can be adapted to study the various aspects of cyber-physical exploitation. In this work we use the full plant-wide control problem proposed by Down and Vogel [7].

4.1 Tennessee Eastman Challenge Process

The Tennessee Eastman (TE) process is a modified model of a real plant-wide industrial process. The authors intentionally omitted certain specific details of the process to protect its proprietary nature. This makes TE problem an excellent case study because the *a priory* information about process is limited and thus allows emulation of the "grey-box" exploitation use cases. For our empirical analysis we use the TE Matlab model developed by Ricker [27]. The plant produces two liquid (*liq*) products from four fresh gaseous (g) reactants involving two irreversible exothermic reactions composed of chemicals A, C, D, E, G and H:

$$\begin{aligned} A(g) + C(g) + D(g) &\rightarrow G(liq), \qquad \text{Product 1,} \\ A(g) + C(g) + E(g) &\rightarrow H(liq), \qquad \text{Product 2.} \end{aligned}$$

The process has five major operation units: the reactor, the product condenser, a vapor-liquid separator, a recycle compressor and a product stripper as shown in Fig. 3. The gaseous reactants and products are not specifically identified. Feed C is not pure and consists of 48.5% A and 51% C. The byproducts and inert gases are purged from the system in the vapor phase using a vapor-liquid separator whereas products G and H exit the stripper base and are separated in a downstream refining section. The plant has 12 valves for manipulation, and in total 41 measurements (with added measurement noise) are involved in process monitoring. The first 22 measurements are continuous, the rest are sampled composition analysis from chromatographs with a delay of 0.1 or 0.25 hours, depending on the process variable. The simulation model control scheme consists of 18 proportional-integral (PI) controllers, 16 process measurements XMEAS{1;2;3;4;5;7;8;9;10;11;12;14;15;17;31;40} and 9 set points which form 8 multivariable control loops and 1 single feedback control loop [17]. The full notation and units of process characteristics including operation constraints can be found in [7]. The default simulation time of a single experiment is 72 hours.

The process description includes a flowsheet, steady-state (without disturbances) material and energy balance as well as the operational constraints of both the optimal steady-state operations and the process shutdown limits. Depending on her goal (economic or physical disruption), the attacker would have to violate these specified constraints. In addition, the original problem statement includes typical set points and load changes, which along with other listed disturbances illustrate different aspects of process operations. In total there are 20 disturbances modes IDV{1-20} and four set point changes. This information is valuable to the attacker as any change in operations causes variations in the process behavior which in turn is visible in the process measurements (sensor signals).

Depending on the noise level and shape of the signal, sensor signals in TE process can be roughly divided into 4 distinct groups (Fig. 4). Type 1 is characterized by a few distinct peaks and a low noise level. Type 2 is distinguished by the multiple noisy signal peaks. Type 3 can be described as a very noisy variation of Type 1 signal. The type 4 signal distinguishes itself by the overall slow signal amplitude change with high amplitude noise. Depending on the state of process, sensor signals can change their properties substantially. For example, A feed flow F_A is of Type 4 in a steady state, of Type 2 under disturbance IDV(11) and of Type 1 under IDV(8).

Initial model does not allow any randomness in the simulations to guarantee the repeatability of the plant operation disturbances. It means that each simulation run produced identical results. In order to obtain statistically significant results we modified the original code by generating a new seed for the random number generator for each run while preserving underlying dynamics of process behavior.

4.2 DoS Attack Model

Let $X_i(t)$ be a measurement of sensor i at time t, where $0 \leq t \leq T$, and T the duration of the simulation; time is discrete. The attack interval T_a is arbitrary and is limited to the simulation run time. In our setting, we simulate manipulated sensor readings X'_i as follows:

$$X'_i(t) = \begin{cases} X_i(t), & \text{for } t \notin T_a \\ X^a_i(t), & \text{for } t \in T_a, \end{cases}$$

where $X^a_i(t)$ is the modified reading (attack value).

During a *DoS attack*, new sensor measurements do not reach the controller. If the attack starts at time t_a, we have:

$$X^a_i(t) = X_i(t_a - 1).$$

where X^a_t is the stale data reading (the last value received by the controller before the DoS attack).

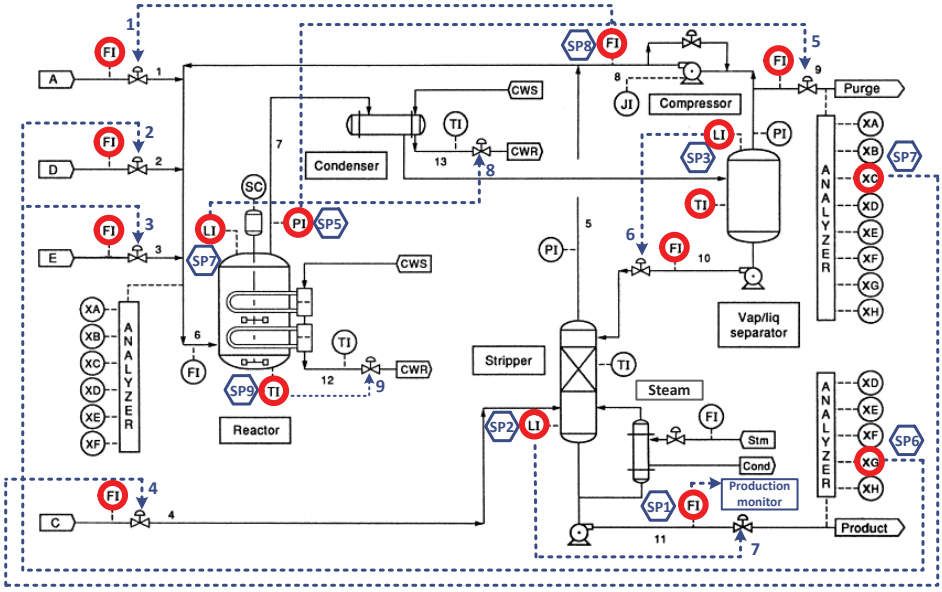
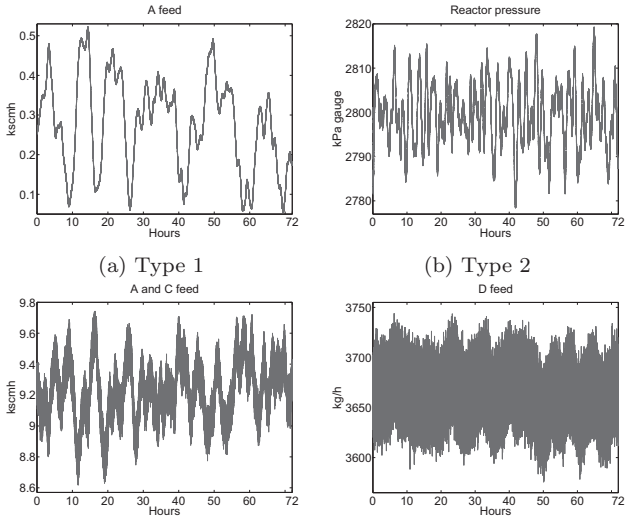


Figure 3: Plantwide chemical process under control–based on [17]



(a) Type 1 (b) Type 2

(c) Type 3 (d) Type 4

Figure 4: Different types of sensor signals in TE process

5. EXPERIMENTAL RESULTS

The TE model execution starts with the predefined base values. The "warm-up" phase of the plant lasts for about two hours and is excluded from our analysis. In the original implementation the process data is downsampled to 100 samples per hour (s/h) before being stored in the Matlab workspace. We conduct our analysis based on the sampling rate of f_s=2000 s/h which is used during the actual real time simulations. We omitted analysis of XMEAS{31;41} as these measurements are not real time. Unless specified differently, the experimental results are conducted with IDV(8) active which stands for the variation in the reactor feed. This disturbance is successfully absorbed by the process and does not impact operations but causes noticeable unpredictable deviations in sensor signals. All statistical results are averaged based on 50 simulation runs. The 95% confidence interval is calculated using Student's t-distribution but due to space limitations are not included into paper.

5.1 Evaluation Metric: Shortest Shutdown Time

In our scenario, we assume an attacker whose goal is to force the physical process into unsafe state and cause its shutdown. To evaluate the result of such an attack we select the Shutdown Time (SDT) as a metric that measures the time that the process is able to maintain safe operating conditions (e.g., maintain the pressure in the reactor tank below $3,000kPa$) after the attack is started. A longer SDT is unfavorable to the attacker as plant operators have more time to take corrective measures to bring the process back into a safe state. In order to be able to compare the results of the proposed approaches, we first obtain reference values of the shortest possible safety time for the attack on each individual sensor. The reference values are selected by looking at the overall lowest (min) and highest (max) process values in each individual simulation run. These attacks would be infeasible in real-time because they would require an attacker to go back in time to select the best value. We refer to these SDTs as *optimum* (Tbl. 1).

As can be seen, the sensitivity of control loops to deception attacks varies greatly with a SDT range from minutes to more than 20 hours. We did not include results on XMEAS{10;11} as no attack on these sensors places system into an unsafe state. Also, only five F_A^{min} attack instances triggered a process shutdown. This is because susceptibility of the process to the attack on A feed depends not only on the attack value but also on the overall balance of A- and C-components. At the same time, attacks P_{reac}^{max} and F_A^{max} do not drive process into an unsafe state. This means an assaulter planning an attack on e.g. reactor pressure should strike only at the minimum peaks.

5.1.1 Without Peak Detection

In this subsection we analyze the attacker's prospects of selecting the highest possible process value in real time ap-

XMEAS		Variable name	Optimum	Secretary, n/e			Secretary, n/log(n)			Outlier Test, $\sigma = 2.0$		
			SDT,h	SDT,h	Error,%	NS,%	SDT,h	Error,%	NS,%	SDT,h	Error,%	NS,%
(1)	min	A-feed	22.22	-	13.96	40	-	42.85	12	-	18.74	70
	max	rate										
(2)	min	D-feed	5.15	5.93	8.52	26	6.05	19.57	0	6.72	42.93	0
	max	rate	3.69	3.85	6.38	20	4.27	13.64	4	6.36	42.15	0
(3)	min	E-feed	4.29	4.46	8.13	38	4.55	17.65	2	5.26	44.87	0
	max	rate	2.83	3.51	7.82	36	3.32	19.43	2	4.44	45.18	0
(4)	min	C-feed	1.05	1.78	12.44	34	1.99	35.80	4	2.07	29.15	12
	max	rate	1.78	2.38	18.83	44	1.93	34.45	8	2.20	28.44	6
(5)	min	Recycle	4.39	5.44	17.12	32	8.73	39.17	2	6.77	28.92	14
	max	flow	9.17	9.90	25.12	40	10.11	42.66	0	10.17	35.44	12
(7)	min	Reactor	8.56	8.2	23.41	34	8.21	39.20	8	8.51	25.93	8
	max	pressure										
(8)	min	Reactor	2.37	2.90	14.57	34	3.31	30.05	2	3.96	44.98	0
	max	level	2.73	3.08	11.13	32	3.40	24.20	0	3.71	37.69	0
(9)	min	Reactor	1.34	1.39	4.06	30	1.45	13.54	8	1.75	45.25	0
	max	temper.	0.65	0.69	3.74	40	0.70	13.56	6	3.71	37.69	0
(12)	min	Separator	5.50	6.92	17.10	36	8.74	40.67	0	8.15	31.18	0
	max	level	3.49	4.28	15.97	30	5.86	41.60	2	5.96	36.10	4
(14)	min	Separator	12.03	12.10	17.50	34	11.88	39.16	0	12.58	34.01	0
	max	underflow	11.58	11.13	20.22	26	11.69	40.70	2	12.15	34.33	2
(15)	min	Stripper	6.39	7.96	20.28	20	11.06	49.43	4	9.70	35.63	14
	max	level	6.35	8.54	23.76	34	13.05	56.36	2	9.71	34.01	10
(17)	min	Stripper	1.86	2.33	4.89	18	3.37	13.91	4	6.75	42.77	0
	max	underflow	1.33	2.36	7.07	24	3.10	19.99	2	7.14	43.30	0

Table 1: Simulation results of the approaches to the Best Choice Problem solution

plying SP approach, without dealing with the fact that sensor measurements are correlated. To begin, the attacker has to decide on two parameters in the context of the secretary problem, namely on the number of samples or alternatives n she is going to consider, and the duration of the learning phase. For simplicity we measure n in hours. In this case, with a time frame of 48 hours, the number of alternatives is equal to $48 \times f_s$. We test both lengths of the learning window n/e and $n/log(n)$. There is no specific requirement to the size of learning window for the Outlier Test. We keep it set to n/e to obtain results comparable with the "classic" secretary approach. Additionally, the Outlier Test requires a decision on the detection threshold. We set it to $\sigma = 2$ which accounts for 95.45% of the data set. We evaluate the approaches based on two metrics: (1) error in selecting the max value in the attack window, in %; (2) number of non-selections (NS) – when no sample is higher than the one observed in the observation window. To evaluate the influence of the error in selecting the highest value we also measure the SDT for each attack instance. The results of the simulations are summarized in Table 1.

Although the classic SP solution with the size of learning window n/e results in the lowest error when selecting the best sample, its shortcoming is a high number of non-selections. This side effect is disadvantageous for the attacker as she either has to choose a clearly suboptimal candidate (last sample in the attack window) or decide to not launch an attack. Cutting the learning window to $n/log(n)$ or applying the Outlier Test substantially reduces the number of non-selections albeit at the cost of selecting a lower attack value and an increased SDT. As the table indicates, most of the control loops are sensitive to the magnitude of the selected attack value.

Another feature of the classic SP solution is at odds with the attacker requirements in that it prescribes a long learning phase before selecting a candidate. The length of the observation window together with a time to selection (TTS) constitute a time to attack (TTA). The longer TTA, the greater chance for the attacker to be detected (also accidentally). Table 2 presents a comparison between the approaches based on the aggregated (averaged) results from Tbl. 1 and results on TTA and TTS. Although OT and SP with $n/log(n)$ algorithms demonstrate roughly similar results in the accuracy of selecting the maximum value and time before making a selection, the latter approach benefits from a drastically smaller time to attack with a very small observation window. It is therefore is more advantageous to the attacker.

	Error	TTS	TTA	NS
SP, n/e	13.78 %	11.52 h	27.17 h	37.77 %
SP, $n/log(n)$	31.26 %	7.63 h	9.61 h	3.36%
OT, $\sigma = 2.0$	36.93 %	5.32 h	20.98 h	3.9 %

Table 2: Aggregated comparison between the approaches

5.1.2 Peak Detection

To take advantage that sensor signals are correlated, the attacker may add a peak detection step into her attack strategy in order to wait if the signal shows a trend. In the forward looking search approach we apply a simple moving average with a smoothing interval $m = 250$ to filter out high frequency noise.

The threshold in the CUSUM algorithm depends on the scale of the change an assaulter would like to detect. Thus, for the noisy signals the threshold should be selected higher than for the low noise signals. We determined optimal thresholds being $h = 0.005$ for signal or Type 1 and 2, and $h =$

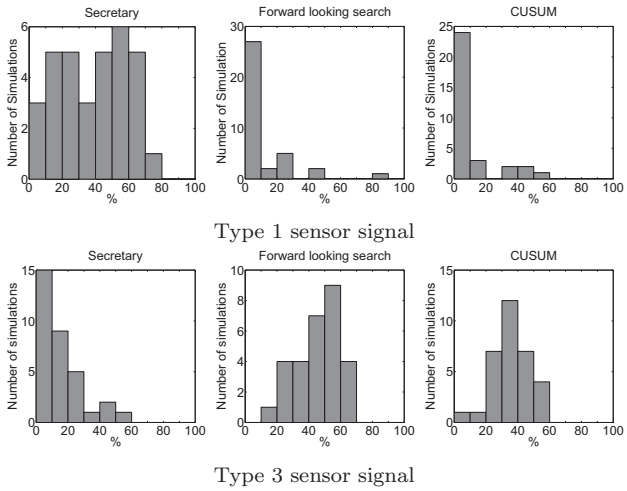


Figure 5: Distribution of the error in selecting highest possible sample, without and with peak detection

0.015 for signal of Type 3. There are no distinct peaks in the signals of Type 4. We compare the peak detection approaches with the classic SP solution as it delivers the most optimal results in selecting the max value (without peak detection). We illustrate the comparison of the results in the form of a histogram which represents the distribution of the error in selecting an optimal sample (Fig. 5).

As can be seen, peak detection approaches deliver superior results in comparison to the SP algorithm for the sensor signal of Type 1 (first row in the figure). This is because in the SP search finishes too early by ignoring the correlations of the signal, thus yielding high error in selecting the max value. At the same time the low noise level allows accurate detection of the signal peak. In contrast, the proposed approaches do not perform well in the case of noisy signal. Thus, noise corrupts the CUSUM statistics resulting in high false alarm rate. Although smoothing reduces the noise in a signal, it introduces a delay of smoothed signal by $(m-1)/2$ samples resulting in the peak detection when the actual signal is already decaying. Vice versa, the SP delivers a lower error while dealing with noisy signals as it sets up the aspiration value based on the highest outlier value in the observation window.

Whereas enhancing SP solution with peak detection may in certain cases substantially increase the attacker's chances in selecting max value, any peak detection algorithm requires to be tuned first to match sensor signal properties.

5.1.3 Attacks in a Steady State

So far we have reported results with disturbance IDV (8) active. However, one of the primary goals of process control is to keep the process as close as possible to its optimal steady state (without disturbances). In the TE process variables in a steady state do not deviate much from the set points and accurately follow the Gaussian distribution. Most of the sensor signals are of Type 4 (with few signals of Type 3). Results from Table 3 shows that relative to each other, all approaches demonstrate performance similar to the use case from 5.1.1 with the exception of the NS statistics for OT. Even with a higher threshold of $\sigma = 3.5$ the Outlier Test delivers diminishingly small number of non-selections. Also, in a steady state the observation window

	Error	TTS	TTA	NS
SP, n/e	3.20 %	2.87 h	28.53 h	37.09 %
SP, $n/log(n)$)	9.09 %	8.39 h	10.37 h	8.64 %
OT, $\sigma = 3.5$	7.82 %	6.39 h	22.02 h	1.5 %

Table 3: Comparison between the approaches, steady state

for the OT can be substantially reduced. For most of the sensor signals it is sufficient to observe only 5 hours of samples in order accurately identify μ and σ. With that the TTA in the Outlier Test becomes comparable to the one of the secretary approach with reduced learning window $n/log(n)$.

Due to the minimal variations in the process measurements all approaches demonstrate a lower error in selecting best possible value. However, for the same reason the selected attack values are also lower. Moreover, in a steady state the process is more resilient to the attacks on certain individual control loops. For instance, it becomes impossible to bring the process into unsafe state with any attack on XMEAS(1) and the SDT for the attacks on XMEAS{5;12;17} increases more than twice. Therefore, it might not be rewarding for the attacker to strike at a steady state.

5.2 Discussion and Future Areas of Research

After obtaining access to the process measurements the attacker faces a number of uncertainties. She neither knows the process variable range nor the sensitivity of the process to the magnitude of the manipulations. What is more important, the adversary has no knowledge about the time constants of individual control loops and specifics of disturbances propagation. It means that the attacker is not certain which attack value to choose and for how long to carry out her assault. The latter is crucial knowledge, for instance for planning concealing activities. Hence, to maximize the impact and minimize attack duration the attacker should try to select the highest or lowest process value possible.

The results of our study shows that the characteristics of the sensor signals even in the same facility are very dissimilar and the attack strategies deliver radically different results when applied to the different types of process measurements at the different plant states and operating modes. Attacking without knowledge about current state of the plant is highly likely to only result in nuisance rather than an actual disruption. In contrast, a knowledgeable attacker may bring the system down in a matter of minutes leaving operators no chance to respond with countermeasures (e.g. [31]). Therefore destructive capabilities in the cyber-physical domain predominantly exist in relation to a specific target and knowledge on process dynamics (such as results from Table 1) are only valid for a specific process control scheme. We would have to conduct a separate analysis for the other control schemes of the TE process, such as [22, 28, 19].

The attacker can do her home work well and design part of the attack in advance; however she will have to tune the attack locally through reconnaissance activities such as changing configuration parameters, manipulating process variables or turning components on and off while observing the system's reaction. From the defense point of view, such short-term "testing" process deviations can be detected by the process-aware anomaly detection solutions [23, 4]. Below we describe further considerations and areas of future research.

5.2.1 Impact of the Sampling Frequency

The sampling rate of the sensor signals changes its noise

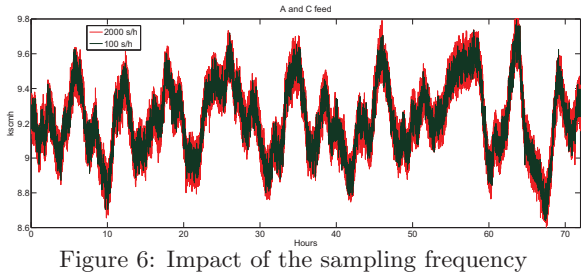


Figure 6: Impact of the sampling frequency

profile. Therefore we also investigated the impact of the f_s. As anticipated, the lower signal sampling rate results in lower error in selecting the best sample due to reduced noise level. However, the selected attack value is also lower if compared to the attack on the signal sampled at a higher rate due to the reduced number of the high amplitude outlier samples (Fig. 6). Depending on the control loop sensitivity, we could observe increase in shutdown time up to 3 hours.

5.2.2 Detection of Plant State Change

Reference value learned in the observation window is only valid for a particular plant state. In practice, the process periodically undergoes through the periods of changes in its operating conditions such as updates of set points, operating modes, production loads, disturbances, etc. The attacker needs to be able to detect such changes quickly in order to adapt her attack strategy to new circumstances. Fig. 7 demonstrates detection of the A/C feed ratio change using CUSUM algorithm. With a threshold $h = 0.001$ such change can be detected in 8.5 min and with $h = 0.005$ in 12.5 min. Once the change is detected, the attacker can either reset her learning phase straight away or wait for some time and see whether process state will keep changing.

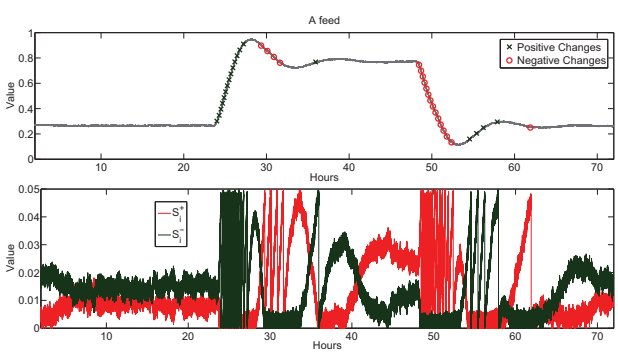
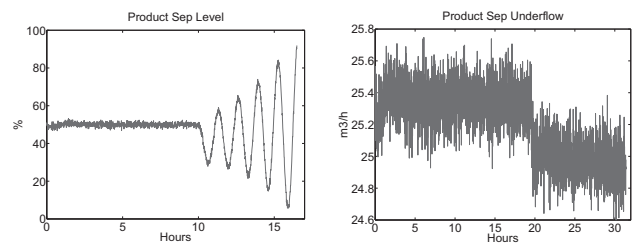


Figure 7: Detection of the plant state change, $h = 0.005$

5.2.3 Chaining Attacks

As was mentioned in 5.1.3, even after executing a successful DoS attack in a steady state it would take a long time to bring the process into unsafe state. In order to achieve shorter SDT, the attacker would need to disturb the process first to cause greater deviation of the process measurements (so that she could select a higher or lower attack value). Causing a plant-wide disturbance might be hard, however the attacker can "chain" two DoS attacks to accomplish her goal. For instance, F_{sep}^{min} attack on a separator underflow causes an oscillation effect on the separator level (Fig. 8a). After 30 min the separator level L_{sep} reaches 30%. The attacker can use CUSUM algorithm to detect the change and launch a DoS attack on L_{sep} when its value reaches its lowest point. In this way the shutdown can be reached in 3.43 hours in comparison to 12.03 hours if the assaulter would execute a direct attack on separator level sensor in a steady state. Similarly, L_{sep}^{min} attack causes an immediate step change of the separator underflow (Fig. 8a) which can be quickly and accurately detected and used for the successive DoS attack.



(a) Oscillation effect of F_{sep}^{min} attack

(b) Step change effect of L_{sep}^{min} attack

Figure 8: Possible steps of chaining attacks

6. RELATED WORK

Securing process control infrastructure and control communication is the first step to safe and secure operations. A large body of literature on Industrial Control Systems (ICS) security looks at intercepting and manipulating the traffic [29], infection by malware [3], access by unauthorized users [18] and addresses the threats by designing ICS-specific defenses such as intrusion detection systems [32, 13], authentication and encryption schemes [8], access control [26], code verification [25] and others.

Physical processes and their particular states are inherently time dependent. The important role of the timing parameters in cyber-physical security was already demonstrated in few academic works. Thus it was shown that process-aware segmentation of the control network increases survivability of the process and extends its time to shutdown [10]. In another work authors studied the effectiveness of remotely executed cyber attacks on a valve in a Boiling Water Power Plant [11]. PLC task scheduling turned to have one of the major impacts on the attack outcome. In [1] authors discuss synchronization as a timing parameter crucial for the stability of the power grid and provide an understanding of the impact of timing uncertainty on the system model accuracy needed to achieve timely situational intelligence. A heuristic triangle approximation algorithm from [16] can be used for peak detection in sensor signals.

7. CONCLUSION

In this paper we introduced the problem of timing DoS attacks based on real-time measurement of process values. We used the TE process to illustrate our approach, but our basic methodology is applicable to any cyber-physical system.

The results of our study shows that the characteristics of the sensor signals even in the same facility are very dissimilar and the attack strategies deliver radically different results when applied to the different types of process measurements. We also showed that applying peak detection algorithms for dealing with correlated time series can potentially improve the performance of the attacker. However noisy signals can render peak detection ineffective.

In general, it is not possible to give definite conclusions regarding which of the proposed approaches is more effec-

tive. All approaches have their own advantages and disadvantages, and their performance largely depends on the type of the signal under analysis and the state of the plant. For instance, NCUSUM can detect changes to the plant state (e.g., in case of the reference variable change) as it is quick and can be easily tuned; however, it was not a significant addition to the problem of detecting peaks in noisy signals.

Overall, the classic secretary approach delivers the best results at the cost of having a long learning phase and a relatively high number of non-selections; however, it consistently delivers good results regardless of the plant state and signal shape. This is particularly useful in the context of black-box exploitation, when the attacker has no a prior knowledge about sensor signals properties. Adversaries that do not want to take the risk of ending the observation period without making a decision may select the SP solution with the reduced learning window $n/log(n)$ or the Outlier Test in order to shorten the time to attack.

In future work we plan studying how attacking multiple sensor signals can affect the system, and to include DoS attacks on control signals.

Acknowledgments

The work of Alvaro A. Cardenas was supported in part by award 70NANB14H236 from U.S. Department of Commerce, National Institute of Standards and Technology.

8. REFERENCES

[1] J. Amelot, D. Anand, T. Nelson, G. Stenbakken, Y.-S. Li-Baboud, and J. Moyne. Towards Timely Intelligence in the Power Grid. In *44th Annual PTTI Meeting*, 2012.

[2] R. Bell and K. Åström. *Dynamic Models for Boiler-Turbine aAlternator Units: Data Logs and Parameter Estimation for a 160 MW Unit*. 1987.

[3] A. Carcano, I. N. Fovino, M. Masera, and A. Trombetta. SCADA malware, a proof of concept. In *CRITIS'08*, pages 211–222, 2008.

[4] A. A. Cárdenas, S. Amin, Z.-S. Lin, Y.-L. Huang, C.-Y. Huang, and S. Sastry. Attacks against process control systems: risk assessment, detection, and response. In *ASIACCS'11*, pages 355–366, 2011.

[5] R. Chen, K. Dave, T. J. McAvoy, and M. Luyben. A Nonlinear Dynamic Model of a Vinyl Acetate Process. *Industrial & Engineering Chemistry Research*, 42(20):4478–4487, 2003.

[6] Control Systems Security Program. Common Cybersecurity Vulnerabilities in Industrial Control Systems. 2011.

[7] J. J. Downs and E. F. Vogel. A plant-wide industrial process control problem. *Computers & Chemical Engineering*, 17(3):245–255, 1993.

[8] I. Fovino, A. Carcano, M. Masera, and A. Trombetta. Design and implementation of a secure modbus protocol. *CIP III*, pages 83–96, 2009.

[9] P. Freeman. The secretary problem and its extensions: A review. *International Statistical Review/Revue Internationale de Statistique*, pages 189–206, 1983.

[10] B. Genge and C. Siaterlis. An Experimental Study on the Impact of Network Segmentation to the Resilience of Physical Processes. In *LNCS*, volume 7289, pages 121–134. 2012.

[11] B. Genge, C. Siaterlis, and M. Hohenadell. Impact of network infrastructure parameters to the effectiveness of cyber attacks against industrial control systems. *IJCCC*, 7(4):673–686, 2012.

[12] J. Gilbert and F. Mosteller. Recognizing the Maximum of a Sequence. *Journal of the American Statistical Assosiation*, pages 35–73, 1966.

[13] D. Hadžiosmanović, D. Bolzoni, and P. Hartel. A log mining approach for process monitoring in SCADA. *IJIS*, 11(4):231–251, 2012.

[14] R. Langner. To kill a centrifuge. Technical report, Langner Communications, 2013.

[15] J. Larsen. Breakage. *Black Hat Federal*, 2008.

[16] J. Larsen. Miniaturization. *Black Hat USA*, 2014.

[17] T. Larsson, K. Hestetun, E. Hovland, and S. Skogestad. Self-optimizing control of a large-scale plant: The Tennessee Eastmann process. *Ind. Eng. Chem. Res.*, 40(22):4488–4901, 2001.

[18] E. Leverett and R. Wightman. Vulnerability Inheritance Programmable Logic Controllers. *GreHack'13*, 2013.

[19] W. L. Luyben, B. D. Tyreus, and M. L. Luyben. *PlantwideProcess Control*. McGraw-Hill, 1998.

[20] M. Mahdian, R. P. McAfee, and D. Pennock. The secretary problem with a hazard rate condition. In *Internet and Network Economics*, pages 708–715. 2008.

[21] S. Mannan. *Lees' Loss Prevention in the Process Industries: Hazard Identification, Assessment and Control*, volume 1. Butterworth Heinemann, 2005.

[22] T. McAvoy and N. Ye. Base control for the Tennessee Eastman problem. *Computers & Chemical Engineering*, 18(5):383 – 413, 1994.

[23] T. McEvoy and S. Wolthusen. A Plant-Wide Industrial Process Control Security Problem. In *CIP V*, volume 367, pages 47–56. 2011.

[24] C. McIntyre. Using Smart Instrumentation. *Plant Engineering: online magazine*, 2011.

[25] S. McLaughlin, D. Pohly, P. McDaniel, and S. Zonouz. A Trusted Safety Verifier for Process Controller Code. In *NDSS 2014*, 2014.

[26] M. Naedele. An Access Control Protocol for Embedded Devices. In *IEEE International Conference on Industrial Informatics*, pages 565–569, 2006.

[27] N. L. Ricker. Tennessee Eastman Challenge Archive. `http://depts.washington.edu/control/LARRY/TE/download.html`. retrieved: May, 2013.

[28] N. L. Ricker and J. Lee. Nonlinear model predictive control of the Tennessee Eastman challenge process. *Comp. & Chem. Engineering*, 19(9):961 – 981, 1995.

[29] J. Rrushi. SCADA protocol vulnerabilities. In *Critical Infrastructure Protection*, volume 7130 of *LNCS*, pages 150–176. 2012.

[30] J. F. Smuts. *Process Control for Practitioners*. OptiControls Inc, 2011.

[31] U.S. Chemical Safety and Hazard Investigation Board. T2 Laboratories Inc. Reactive Chemical Explosion: Final Investigation Report. 2009.

[32] M.-K. Yoon and G. F. Ciocarlie. Communication Pattern Monitoring: Improving the Utility of Anomaly Detection for Industrial Control Systems. In *SENT 2014*, 2014.

Uncovering Network Tarpits with Degreaser

Lance Alt
Naval Postgraduate School
laalt@cmand.org

Robert Beverly
Naval Postgraduate School
rbeverly@nps.edu

Alberto Dainotti
CAIDA, UC San Diego
alberto@caida.org

ABSTRACT

Network tarpits, whereby a single host or appliance can masquerade as many fake hosts on a network and slow network scanners, are a form of defensive cyber-deception. In this work, we develop *degreaser*, an efficient fingerprinting tool to remotely detect tarpits. In addition to validating our tool in a controlled environment, we use *degreaser* to perform an Internet-wide scan. We discover tarpits of non-trivial size in the wild (prefixes as large as /16), and characterize their distribution and behavior. We then show how tarpits pollute existing network measurement surveys that are tarpit-naïve, e.g. Internet census data, and how *degreaser* can improve the accuracy of such surveys. Lastly, our findings suggest several ways in which to advance the realism of current network tarpits, thereby raising the bar on tarpits as an operational security mechanism.

Categories and Subject Descriptors

C.2.0 [**Computer-Communication Networks**]: General—*Security and protection*; C.4 [**Performance of Systems**]: Measurement techniques

Keywords

Tarpits; Internet Census; Sticky Honeypot; Deception

1. INTRODUCTION

Networks face a continual barrage of abusive and malicious attacks. Among available network security defenses is the class of *deceptive* network strategies. The high-level idea is simple: provide to adversaries the illusion of vulnerable targets, or promote an appearance of greater complexity than actually exists. For example, network honeypots [31] have long been used to attract abusive traffic and attacks. Traffic that arrives at the honeypot can then be analyzed in order to build databases of e.g. IP reputation [19, 34], malware signatures [6, 37], and to provide early detection of emerging attacks [25]. Such databases of abusive traffic characteristics can then be leveraged to help mitigate attacks elsewhere in the network.

A more advanced view of deception includes not only providing a believable target, but actively influencing the adversary through deceit [10, 28, 29]. For example, Trassare's work on deceptive network topology [32] seeks to alter the adversary's notion of the attack surface in an effort to change their decision making process, whereas *moving target defense* systems [18] frequently change the IP addresses of resources to increase the complexity of an attack against them.

By extension, network "tarpits" (or "sticky honeypots") seek to slow automated scanning traffic or frustrate and confuse a human adversary [21, 35, 13, 16, 30, 15]. These tarpits can be configured to use inactive IP addresses or address blocks within a network, thereby providing the illusion of a large pool of active hosts. For each fake host, a single tarpit machine can answer all incoming connection requests. Some tarpits answer both to ICMP requests and on all TCP ports – providing the illusion of both fake hosts and services. In addition, by employing TCP flow control mechanisms to hold the connection open without allowing data transfer, the tarpit can both slow and penalize a scanning attacker.

In this work, we investigate the ability to detect network tarpits, and provide empirical evidence of their presence on today's Internet. Our work is motivated by defensive security objectives, where the efficacy of a tarpit is limited by the degree to which the deception is believable; an adversary, whether automated or human, will avoid known tarpits. For instance, an automated scan that can detect tarpits in real-time could skip those tarpits and tarpit networks, improving the scan performance (both in time and accuracy).

We develop an active, on-line probing methodology to efficiently (2 to 6 packets per target) detect a variety of different tarpits in real-time. Our methodology is based on TCP options fingerprints and TCP flow control behavior. By sending a series of specially crafted probe packets, we discern real TCP stacks from tarpit hosts. We synthesize our methodology into a publicly available open source tool [5], *degreaser*, that infers the presence of tarpits among a set of input target networks.

In addition to validating our methodology against known-ground truth tarpits, we utilize *degreaser* to perform an Internet-wide scan. To facilitate large-scale scanning and avoid triggering anomaly detectors, *degreaser* uses permutation scanning [7, 12] to pseudo-randomly iterate through the IP address space when probing. Our real-world Internet scan, which probes at least one address in each /24 network in the Internet, discovers 107 different tarpit subnetworks

This paper is authored by an employee(s) of the United States Government and is in the public domain. Non-exclusive copying or redistribution is allowed, provided that the article citation is given and the authors and agency are clearly identified as its source.
ACSAC '14, December 08–12, 2014, New Orleans, LA, USA
ACM 978-1-4503-3005-3/14/12 ...$15.00
http://dx.doi.org/10.1145/2664243.2664285.

ranging in size from /16 (with up to 2^{16} fake hosts) to /24 (with up to 2^8 fake hosts). As some of these subnetworks have both real hosts and tarpits interleaved, we characterize the occupancy of fake addresses, both as a function of tarpit type and network size. In all, we find over 215,000 active IP addresses that are fake. These provider, customer, and university networks spread across 29 different countries and 77 autonomous systems. While these numbers represent non-trivial portions of address space, they are small relative to the size of the Internet. However, we note that even small blocks of tarpit addresses can greatly slow automated scans as part of their intended capturing behavior.

To better understand how fake hosts and subnetworks are observed in Internet measurement campaigns, we examine publicly available measurement data from an Internet-wide ICMP census [8] and HTTP scans [4]. We find that the networks inferred by *degreaser* as filled with tarpits, appear instead as fully occupied in the census data. Thus, not only are tarpits affecting abusive network scans, they also successfully deceive legitimate measurement surveys. As such surveys are used in policy decisions (e.g. understanding IPv4 address space exhaustion and allocating new addresses), we offer *degreaser* as a means to improve their accuracy.

In general, we demonstrate that tarpit deception is operationally deployed in the Internet. We therefore make four primary contributions:

1. We devise fingerprinting techniques to efficiently detect network tarpits in real-time. We synthesize our approach in *degreaser*, an open source tool that discerns deceptive tarpit hosts and networks (Section 3).
2. We discover tarpits in the wild by running Internet-wide scans using *degreaser*, showing that such security tools are actually employed in real world scenarios and characterizing their deployment (Section 4.1).
3. We empirically examine how tarpits pollute network measurement studies, including the Internet census [2] (Section 4.2). With *degreaser*, we suggest that such measurement surveys can return more accurate results.
4. We suggest improvements to make tarpit deception more realistic (Section 5), thereby raising the bar for this operational security mechanism.

2. BACKGROUND

Cyber-deception has been used to great effect in understanding and mitigating network attacks. For instance, honeypots [31] may attempt to lure an attacker away from a true network resource to some deceptive resource, thereby gaining additional intelligence on the tools, techniques, and procedures (TTP) employed in an attack. Similarly, darknets and network telescopes [24, 35] capture traffic destined to unused blocks of address space, passively providing insight into attack behaviors.

Not only does deceit permit intelligence gathering, it can induce an attacker to expend time evaluating potential attack vectors on fake resources, slows their progress and keeping them in an intelligence gather phase – increasing the likelihood of being discovered [29].

In this work, we restrict our analysis to a particular type of deception: fake hosts of network tarpits. In this section, we review the salient features and operation of network tarpits, as well as prior work in identifying network deception.

2.1 Network Tarpits

Network tarpits were originally conceived in response to aggressive scanning worms, e.g. code-red [23]. Analogous to physical tarpits (which trap animals in sticky tar), network tarpits attempt to hold open remote incoming TCP connections that are likely to be abusive. Once held (or "stuck"), the tarpit both actively prevents data transfer over the connection and discourages the remote end from disconnecting. By preventing such connections from performing any useful work, the tarpit both slows the scanner and consumes the scanner's resources. Penalizing abusive connections, whether from spammers, worms, or other malicious activities, not only makes the target appear less attractive, it helps to slow the global rate of scanning and permits introspection over the behavior of the scanner.

Network tarpitting has been applied at both the transport and application layers. For instance, SMTP tarpits attempt to slow and penalize email spammers once the application-layer detects that the incoming email is likely spam [13, 16]. The tarpit mechanism employed by an email server could be at the SMTP-layer, for instance by taking an arbitrarily long time to respond to incoming SMTP commands.

Other tarpits are able to extend their deceptive operation by co-opting unused addresses on their local subnetwork. Network tarpits such as those in LaBrea [21] and the Netfilter TARPIT plugin [15] can answer and hold connections to multiple IP addresses on a network, including addresses that are not in use. In this fashion, not only do these tarpits slow connections to active machines on the network, they fake the existence of servers for every IP address in network prefix.

2.2 Tarpit Operation

Network tarpits employ three primary mechanisms: 1) a means to determine which IP addresses on a subnetwork are unused and thus may be faked; 2) a strategy to impersonate hosts by responding to TCP, UDP, or ICMP probes destined to fake IP addresses; and 3) a method to hold the TCP connection open.

The IP addresses for which the tarpit responds may be statically configured, or dynamically inferred. When inferred, the tarpit typically acts conservatively and relies on the lack of a layer-2 address resolution protocol (ARP) response as a indication that the IP address in question is not in use by a real host. For example, Figure 1 shows the sequence of packets observed on a subnetwork running an instance of LaBrea [21]. The local router, `10.1.10.1`, has a packet with destination `10.1.10.210` to deliver. Because the network is an Ethernet, it must first determine the layer-2 address of the destination and sends an ARP request. Note that the first three ARP requests in Figure 1 go unanswered and are spaced roughly one second apart. LaBrea promiscuously listens on its network interface and observes these unanswered ARP requests. On the fourth request, LaBrea conservatively infers that there is no true host with the requested IP address and instead responds with its own layer-2 address on behalf of that IP address. In this way, LaBrea takes over responsibility for the IP address, and will do so similarly for other unused addresses.

Next, TCP connections must be held open. A TCP-level tarpit typically listens on all TCP ports and responds to SYN connection initiation with a SYN/ACK. However, the general strategy of the tarpit is to use the 16-bit *window* field in the TCP header [26] to lock the client in a fully

```
06:20:44.848758 ARP, Request who-has 10.1.10.210 tell 10.1.10.1, length 46
06:20:45.953257 ARP, Request who-has 10.1.10.210 tell 10.1.10.1, length 46
06:20:46.962535 ARP, Request who-has 10.1.10.210 tell 10.1.10.1, length 46
06:20:47.970023 ARP, Request who-has 10.1.10.210 tell 10.1.10.1, length 46
06:20:47.970130 ARP, Reply 10.1.10.210 is-at 00:00:0f:ff:ff:ff, length 28
```

Figure 1: LaBrea layer-2 conservative host proxy. On a subnetwork, LaBrea waits for three ARP timeouts before it infers that the host is not present. LaBrea then commandeers the host as part of the tarpit.

```
06:20:47.971276 IP 1.2.3.4.51161 > 10.1.10.210.http: Flags [S], seq 3536100821, win 65535,
                options [mss 1460,nop,wscale 4,nop,nop,TS val 1194569089 ecr 0,sackOK,eol], length 0
06:20:47.971475 IP 10.1.10.210.http > 1.2.3.4.51161: Flags [S.], seq 1457023515, ack 3536100822, win 10, length 0
```

Figure 2: LaBrea layer-3 tarpit behavior. LaBrea replies to incoming TCP connections with an option-less SYN/ACK with a window size of 10. LaBrea then either stops replying or flow controls the connection with ACKs with window size 10 (persistent mode).

established TCP connection, while simultaneously preventing it from transmitting data. In the TCP protocol, the window field implements flow control – providing an indication to the sender of the number of bytes the host is able to accept [26], thus relieving the network of the burden of transmitting packets that can only be dropped by the receiver. The tarpit exploits such mechanism by advertising a small initial window, and then replying to incoming packets with a window size of zero. Per the TCP specification, the remote host will send "zero-window" probes periodically to the tarpit to determine when it can resume sending data. The tarpit will never increase the window, keeping the remote host locked in a persistent connection. In practice, most operating systems at the client will eventually terminate the TCP connection after a certain amount of time has elapsed without an increase in the window size. When the remote host decides to terminate the connect using the normal FIN process, the tarpit ignores these packets forcing the operating system to maintain connection resources until the FIN-WAIT period expires. During the whole connection, the client's socket resources are consumed to keep connection state (while the tarpit instead maintains no state), but the client is unable to perform any useful work.

While the mode of operation described above is known as *persistent mode*, tarpits support also a non-persistent mode in which all the subsequent packets on the connection sent by the client after the initial handshake are simply ignored. Figure 2, shows an example SYN/ACK response packet from LaBrea with a window size of 10 bytes and no TCP options – a behavior we show in §3 to be different than legitimate TCP connections.

Two of the most widely known tarpit applications are LaBrea [21] and the Linux Netfilter TARPIT plugin included in the `xtables-addons` package [15]. In total, we consider the following types of tarpits in this work:

- ***LaBrea:*** LaBrea [21] runs as a user-space application using raw sockets to intercept all packets arriving from the network. TCP responses are crafted without using the operating system network stack, thus requiring no per-connection resources. LaBrea supports hard capturing a specific list of IP addresses, or it can utilize unused IP addresses within a network block using its ARP-timeout mode. In ARP-timeout mode [21], LaBrea will intelligently release IP addresses it responds for if another device joins the network. This allows a subnet to remain "full" even as hosts join and leave the network. LaBrea can be run in either persistent or non-persistent modes. Herein we refer to these as "Labrea-P" and "Labrea-NP" respectively.

- ***Netfilter:*** The `xtables-addons` package [15] includes two deceptive plugins: TARPIT and DELUDE. Netfilter interfaces directly with the operating system, providing additional target firewall rules to which packets can be directed. Netfilter rules are implemented by the user via the `iptables` application. Because the plugins are integrated in to Netfilter, it permits use of the full compliment of firewall rules, for instance permitting more complex decisions than LaBrea when choosing which connections to tarpit. However, Netfilter is more limited when establishing a tarpit over a wide range of addresses, since the network interface must be bound to all IP addresses covered by the tarpit. Additionally, Netfilter does not provide an ARP-timeout mode, requiring all tarpit IP addresses to be specified.

The `xtables-addons` TARPIT plugin holds TCP connections in the same fashion as LaBrea, but does so in a unique what that allows us to distinguish it from LaBrea as we discuss in the next section. In addition, `xtables-addons` includes a DELUDE plugin that replies to incoming SYN connections with a zero-window SYN/ACK, but sends a RST to terminate the connection for all other packets. Delude is designed to fool network scanners looking for open ports. While DELUDE does not explicitly attempt to hold the connection open, it provides deception by responding for non-existent IPs and services. Herein, we refer to the Netfilter TARPIT plugin as "iptables-T" and the Netfilter DELUDE plugin as "iptables-D."

- ***Other:*** Last, we find a class of tarpitting addresses that act as tarpits, but do not behave in a way that is consistent with either LaBrea or the Netfilter plugins as configured by default. The primary distinguishing characteristic of these addresses is that they deterministically respond with a zero-window in the SYN/ACK. We observe that some of these addresses respond once, but then implement a timed blacklist whereby subsequent connections go unanswered for a period of time. While we cannot definitively attribute the variety of behaviors in the group to a particular implementation, they act as tarpits. We therefore herein refer to this group as "Other."

2.3 Detection

Fundamentally, the tarpit detection problem is based on fingerprinting. Both active and passive network stack fingerprinting has been employed with large success to iden-

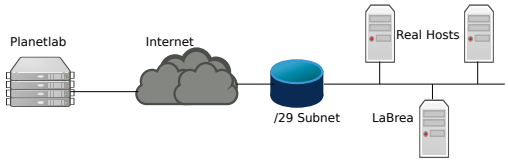


Figure 3: Ground-truth testing on LaBrea network in a /29 subnetwork.

tify operating systems [36, 22]. These methods rely on implementation-specific differences as exhibited by various TCP/IP fields, for example IPID, TTL, options, etc. However, these existing fingerprinting tools fail to identify tarpits. More complex physical device fingerprinting [20] techniques similarly fail due to the lack of TCP timestamp option support in tarpits.

Instead, this work seeks to use additional characteristics to identify tarpits by eliciting tarpit-specific behavior via active probing. To the best of our knowledge, *degreaser* is the first tool to reliably infer the presence of tarpits.

3. METHODOLOGY

In this section, we seek to identify traffic characteristics of network tarpits. We empirically analyze various traffic properties of real production networks as compared to tarpits, and describe features that we find to not provide sufficient power to discriminate tarpits. Then, by running known tarpits in a controlled environment, we develop a set of discriminating traffic characteristics as the basis for *degreaser*. From these characteristics, we detail *degreaser's* inference algorithm. Last, we describe the permutation scanning used by *degreaser* to perform Internet-wide probing.

To verify our hypotheses, we establish a ground-truth LaBrea network on the Internet. As shown in Figure 3, we install LaBrea on a /29 subnetwork and use PlanetLab [9] to probe from multiple vantage points the entire /24 aggregate to which the /29 belongs. We scan the /24 network by attempting to establish TCP connections to each IP address in the subnet and capture the packets for further analysis.

3.1 Non-Features

We first describe features that, intuitively, might provide an indication of a tarpit host. These features, however, proved unreliable or unfeasible.

- ***Layer-2 Address:*** An observer with access to the same network segment as LaBrea can trivially discern traffic from fake hosts by examining the link-layer addresses. LaBrea uses a specific, hardcoded, Ethernet MAC address (`00:00:0f:ff:ff:ff`) for all of its responses regardless of its physical network adapter address.

In contrast to a fixed address, the Netfilter plugin uses the network adapter's MAC address for all packets it generates. However, in normal operation, a single interface may have multiple assigned IP addresses. Thus, the same MAC address across multiple IP addresses is only a weak indicator of a Netfilter tarpit. Since a tarpit would typically be used to combat threats outside the local network, our efforts instead focus on remotely identifiable characteristics, i.e. those that are discernible from outside the network segment containing the tarpit host.

- ***Active IPs in a Subnet:*** Intuitively, we might expect high-occupancy subnets to be good indicators of possible tarpits. To this end, we initially investigated using a hitlist of probable tarpits as inferred from the /24 subnets with more than 240 responding web hosts in the `scans.io` [4] survey. However, we found this to be a poor selection criterion as many of these subnets belong to well known Content Distribution Networks and web hosting services. We did however, make use of other aspects of the data in [4] when building *degreaser* as detailed in §3.3.

- ***Open TCP Ports:*** An IP address that answers for all TCP ports is indicative of a tarpit. However, ascertaining the set of open ports requires 2^{16} probes per host. Even a search for hosts with more than a particular threshold of open ports imparts an exponential impact on the total number of probes required to scan the Internet. Further, some tarpits answer only to one or a small number of ports. We therefore use open TCP ports only as a last test to disambiguate instances of possible tarpits.

- ***Response Time:*** As described in §2.2, LaBrea can use ARP-timeout mode where it waits for multiple ARP requests to go unanswered before using an IP address. The remote host will therefore observe a delayed response time between sending the SYN packet and receiving the SYN/ACK. Intuitively, the default timeout period of three seconds provides a discriminating characteristic to identify a possible LaBrea host. However, this ARP-induced behavior is unreliable as it only occurs when the router's ARP cache is not already populated. Given the large amount of typical network scanning traffic and noise, we could not reliably use such criterion to detect our ARP-based tarpit.

More importantly, in §4 we show that LaBrea typically operates in hard-capture mode, where it does not utilize ARP and does not introduce extra delay. The Netfilter tarpit instead, does not provide an ARP-timeout mode and is not susceptible to exploitation of this characteristic.

3.2 Discriminating Characteristics

We identify two characteristics which, taken together, enable reliable remote detection, via active probing, of fake hosts created by tarpits: TCP window size and options.

3.2.1 TCP Window Size

Fundamental to the tarpit-like behavior of both LaBrea and the Linux Netfilter TARPIT plugin is TCP flow control (§2.2). We observe that LaBrea and the Netfilter plugin return by default a TCP window size of 10 and 5 respectively. While LaBrea's window size is configurable, Netfilter's value is hard-coded into the source.

Given these known window values, we first examine the more general distribution of window values (scaled as required by any TCP window scaling option) as observed in two different traffic captures and summarized in Table 1:

1. ***Equinix:*** One minute of traffic from a 10Gbps link within the San Jose Equinix exchange point on December 19, 2013 as anonymized and made available by CAIDA [1]. This trace contains approximately 5.4M flows, 31M packets, and 24Gbytes of traffic (average of 456Mbps). 94.8% of the traffic is TCP.
2. ***Campus:*** One hour of traffic from our academic department's border router. This trace contains approximately 1.2M flows, 48M packets, and 34Gbytes of traffic (average of 9Mbps). 94.7% of the traffic is TCP.

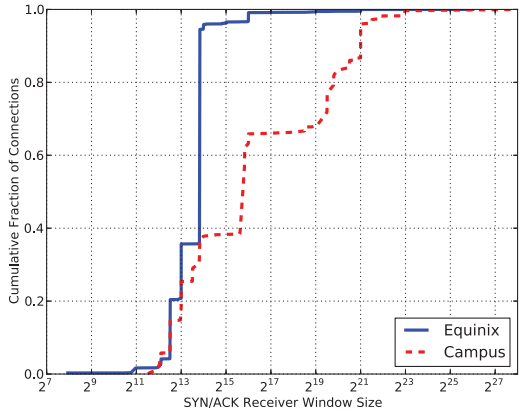


Figure 4: Distribution of receiver window size observed in campus and Equinix traces. Over 99.7% of connections have an initial window size > 512 bytes.

Table 1: TCP option and window sizes of two traces. Since < 0.5% of connections use no TCP options, this is a good indicator of tarpit hosts.

Trace	Length	Pkts	Flows	Min Non-Zero Window Size	No TCP Opts
Equinix	60s	31M	5.4M	246	0.5%
Campus	3660s	48M	1.2M	2,920	0.0%

As shown in Figure 4, ∼99.7% of TCP connections used window sizes greater than 512 bytes. Only 401 (0.3%) experienced a zero window, while none saw the characteristic tarpit windows of either 5 or 10. Thus, by default, *degreaser* looks for TCP connections with initial window sizes less than 20 as the first indicator that a host is a tarpit.

3.2.2 TCP Options

The TCP protocol enables hosts to negotiate additional functionality in the form of TCP options. Typically, these are negotiated by the operating system during connection establishment and are transparent to application layer programs. We again examine the traffic traces to gain intuition over the distribution of TCP options. As shown in Table 1, none of the connections in our campus trace and only 0.5% of connections in the Equinix trace contained no TCP options.

Since both LaBrea and the Netfilter plugin generate TCP packets without the assistance of the operating system network stack, any TCP options must be negotiated by the tarpit application directly. In practice, both LaBrea and the Netfilter plugin ignore all TCP options and return an empty options list. We exploit this behavior by noting any TCP connections with options set (other than MSS as we describe later), and disregard those connections as non-tarpitting.

3.3 Transport versus Application Response

Based on two additional observations, we identified a list of likely tarpits in the Internet for both manual investigation and probing by *degreaser*: (i) tarpits tend to fill the subnet in which they operate; (ii) even if they accept TCP connections, they do not generate any application-layer response. We exploited these two properties in conjunction with data we extracted from the logs of the HTTP scans

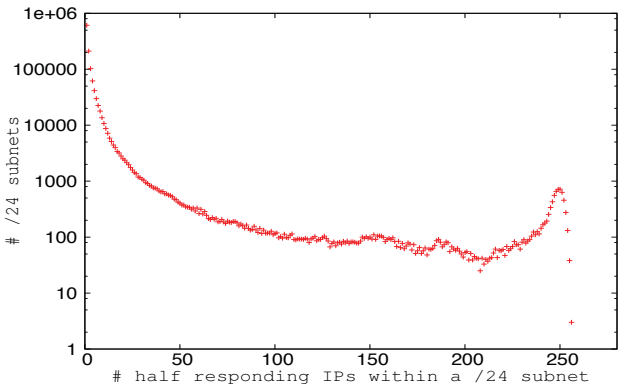


Figure 5: Distribution of /24 subnets with at least 1 IP that accepts TCP port 80 connections but does not generate application-layer HTTP responses (*half-responding* IP). Data from Project Sonar's HTTP scan in April 2014 [4].

conducted by Project Sonar [4]. These scans send regular HTTP GET requests to all IPv4 hosts found listening on port 80/TCP, logging their HTTP response (we call such a host *fully responding*) or `null` when no response is returned (*half responding*). Even if half-responding hosts are not necessarily tarpits (e.g., they may simply be running an application listening on port 80/TCP which is not an HTTP server, such as Skype), our intuition is that when they cover a large percentage of the IP addresses within a subnetwork, they are likely to be fake (e.g., tarpits or some form of firewall). Figure 5, shows the distribution of the /24 subnets that – according to the logs of Project Sonar's HTTP scan from April 2014 – we found hosting between 1 and 256 half-responding hosts (1,577,791 /24 subnets). Through manual active probing, and by comparing logs of subsequent (monthly) HTTP scans available at [4], we found that: (i) there is a certain probability of encountering, or to erroneously infer, a half-responding host; this probability generates the main mode visible in the distribution: a non-linear decay as the number of half-responding hosts within the same subnet increases; each time we manually verified a host which was the only one half-responding within its /24 subnet, we found it was not behaving as a tarpit but rather running an actual service; (ii) however, another mode in the distribution breaks the exponential decay trend roughly around 200 half-responding hosts per subnet; manual verification of several of these cases showed a behavior which either indicates a tarpit filling the network or an unconventional behavior. By winnowing our targets down to these likely tarpits, we were able to tune *degreaser*'s inference algorithm, as described next.

3.4 Degreaser

Based on our detection criteria, we build a publicly available open source tool, *degreaser*, to automatically detect tarpitting hosts and networks. *Degreaser* runs on a standard Linux host using raw sockets and supports multi-threaded scanning. *Degreaser* is designed to scan a list of subnets and classify each responding IP. When tarpit-like behavior is detected, *degreaser* can determine which of the two most popular tarpit applications, LaBrea or the Netfilter plugin, is being used.

Algorithm 1 Degrease(Dst)

```
SYNACK ← SendSYN(Dst)
W ← Window(SYNACK)
if (Options(SYNACK) \MSS = ∅) and (W < 20) then
    ACKResponse ← SendACK(Dst)
    if ACKResponse = RST then
        return(delude)
    else if Window(ACKResponse) = 0 then
        return(iptables-T)
    if W = 0 then
        FINResponse ← SendFIN(Dst)
        if FinResponse = ∅ then
            return(other)
    else
        DataResponse ← SendData(Dst, size = W − 1)
        if DataResponse = ∅ then
            ZeroWinResp ← SendZeroWinProbe(Dst)
            if ZeroWinResp = ∅ then
                return(labrea-NP)
            else
                return(labrea-P)
return(real)
```

3.4.1 Scanning Algorithm

Because *degreaser* circumvents the host TCP stack by sending raw IP packets, we take care to prevent the operating system from receiving unexpected replies. *Degreaser* uses ports outside the ephemeral port range and integrates with the Linux firewall to block incoming packets to that port range. This ensures that the host's operating system does not receive probe responses and send RSTs.

The *degreaser* pseudo-code is given in Algorithm 1. The scan is initiated by sending a TCP SYN packet to the remote host. The packet is a standard SYN that contains common TCP options (MSS, WSOPT, SACK, and TSOPT) [17]. A response timeout of five seconds is used for all outgoing packets and if no response is received within the timeout, the host is marked as not responding. If a SYN/ACK is received, *degreaser* classifies a host as non-tarpitting if the receive window size is greater than the 20 byte threshold determined in 3.2.1. Similarly, if the received SYN/ACK contains any TCP options it classifies the host as non-tarpitting since neither LaBrea nor Netfilter include TCP options in their replies. During testing however, we observe that some paths proactively add the MSS option for all TCP connections when none is present. Previous work has shown that middleboxes are known to add the MSS option[11, 14], therefore *degreaser* can selectively ignore the presence of MSS.

This simple algorithm is sufficient to classify the nodes in our test network as either tarpits or real hosts with perfect accuracy and no false positives or negatives. However, it is possible that a legitimate host would advertise a small window and also not include any TCP options. In addition, we wish to distinguish between the variety of different tarpits defined in §2.2. The remainder of the algorithm provides this functionality.

Next, *degreaser* sends an ACK to complete the three-way handshake. If the response to this ACK is a RST, we infer an iptables using delude. However, if the response contains a window of zero, we infer an iptables tarpit.

Typically, however, we do not expect nor receive a response to the ACK packet. In this case, and when the SYN/ACK window was zero, we send a FIN packet to elicit a response from a valid host. If we receive no response to our FIN, we infer that the target is an unknown tarpit.

Otherwise, if the SYN/ACK window was non-zero, we wish to distinguish between a real-host with a small advertised window and the two types of LaBrea tarpits. We transmit a data packet with a payload size one byte smaller than the advertised receiver window from the remote host. A legitimate host will respond in one of two ways. If the legitimate host is still busy and has not drained its receive buffer, it will send an ACK with the receive window decreased by the amount of data we send. Alternatively, a legitimate host will increase its window if the buffer pressure is relieved. In contrast, LaBrea will not respond. However, in persistent mode, LaBrea will respond to zero window probes. We therefore distinguish between persistent and non-persistent LaBrea by sending a zero window probe and observing whether we receive a response.

Thus, *degreaser* requires only one TCP connection and a maximum of four packets per probed IP address, and avoids non-deterministic network behavior such as response time measurements.

3.4.2 Random Permutation Scanning

To facilitate large-scale network scanning and avoid triggering anomaly detectors, *degreaser* includes the ability to pseudo-randomly scan large network subnets using a cryptographic cipher. We use the RC5 block cipher with a 32-bit block size to create a pseudo-random permutation over a much smaller domain. *Degreaser* automatically switches between either a prefix-cipher or cycle-walking cipher, as described in [7], depending on the number of addresses to scan. Multiple disjoint subnets are combined in to a single continuous domain to provide pseudo-random scanning across the entire scan range.

4. FINDING TARPITS IN THE WILD

In our quest to find live tarpit hosts on the Internet, we first searched on the web for organizations that publicly admit to using tarpits. Our search revealed only one company that indicates using a tarpit. In fact, their website provides a publicly viewable statistics page that shows a list of all the IP addresses captured by their three tarpitting hosts. However, we sought to better understand the wider operational deployment of unadvertised (and previously unknown) tarpitting on the Internet.

4.1 Probing the Internet

Realizing that as a network defense mechanism, very few organizations would provide detailed information revealing tarpit hosts they are running, we scanned the Internet for tarpits. We used *degreaser* with the pseudo-random permutation scanning described in §3.4.2 to scan approximately 20 million IP addresses in May, 2014. Using permutation scanning allowed us to scan at least one host in all of the $\approx 14.5M$ routed /24 subnets over the course of 30 days. Out of the $\approx$ 20 million addresses probed, *degreaser* discovered 1,451 IP addresses exhibiting tarpit-like behavior (either LaBrea or the Netfilter TARPIT plugin). We manually verified a random sample of these hosts, and found that they did indeed exhibiting tarpit-like behavior, confirming that our detection methodology works correctly.

From these seed tarpit IP addresses, we used *degreaser* to perform an exhaustive scan of each /24 subnet containing

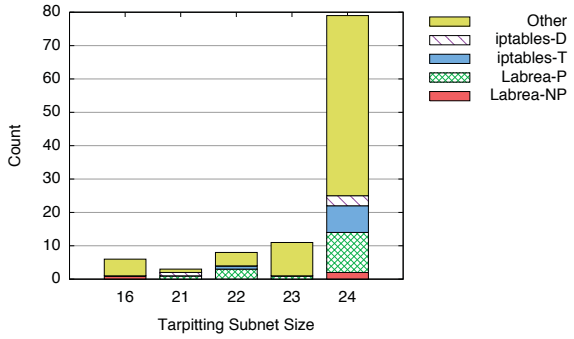


Figure 6: Distribution of tarpitting subnets based on their inferred subnet size.

one of the 1,451 tarpit IPs. Often, the majority of these subnets are completely full of fake tarpit IP addresses. Several subnets however were mixed, having tarpits intermingled with real hosts and non-responding IPs. We more completely characterize the tarpits in the next section.

We then expanded *degreaser*'s search to adjacent subnets to determine if the tarpitting /24 belonged to a larger aggregate tarpitting subnetwork. Overall, we found several larger subnets (up to /16 blocks) filled with tarpits and totaling over 215,000 fake hosts.

4.2 Characterization

From our Internet-wide scan, we assimilate a list of 107 different tarpit subnetworks. These networks are spread across 29 different countries and 77 autonomous systems, indicating that multiple independent organizations are using network tarpits. Additionally, the subnet ownership was diverse, with 51 university subnets, 36 provider subnets, 19 customer subnets, and 1 government subnet exhibiting tarpit-like behavior. This non-trivial presence of operational tarpits in the Internet speaks to one aspect of cyber deception currently used in real networks today. Our survey allows us to understand more about their properties and to validate some of our reasoning.

Figure 6 shows a breakdown of the various tarpit subnet sizes, as a function of tarpit type. Of note are the existence of six large /16 tarpit networks, where a /16 has a total of 2^{16} possible IP addresses. The Netfilter plugins, delude in particular, are the least prevalent of the tarpits we discover. LaBrea in persistent mode is more commonly observed than non-persistent mode, however the "other" tarpit type is most common for all of the subnet sizes. Recall that "other" are tarpits that advertise zero-window and behave like a flow-controlling tarpit, but do not use the default LaBrea or iptables configuration (with a TCP window of either 3, 5, or 10). These other tarpits may be LaBrea in non-standard configurations, or another class of tarpit software or device.

Among our tarpitting subnets, we examine the distribution of latencies we observe during a complete enumeration of all addresses within those subnets. Figure 7 depicts a subset of all tarpit subnets, where each subnet is represented by an inter-quartile boxplot and obfuscated identifier. While the inter-quartile range is small, there are significant outliers. However, none of the addresses exhibit a latency larger than one second, indicating non-ARP based operation.

Next, we examine the occupancy of the tarpitting subnets. Recall that all of the addresses of some subnets are tarpitting, while other subnets are a mix of real and tarpitting IPs. Figure 8(a) shows the cumulative fraction of tarpit networks versus their occupancy for each of the fix tarpit types. We observe a variety of occupancy's, with persistent LaBrea being the most highly occupied.

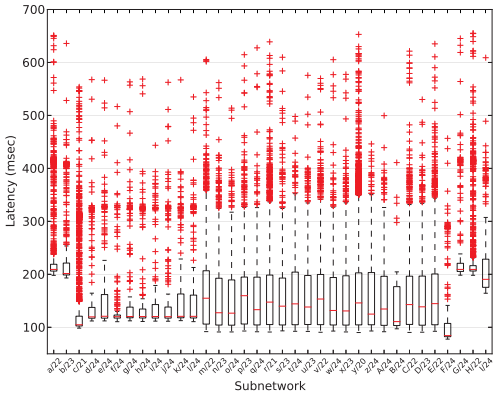


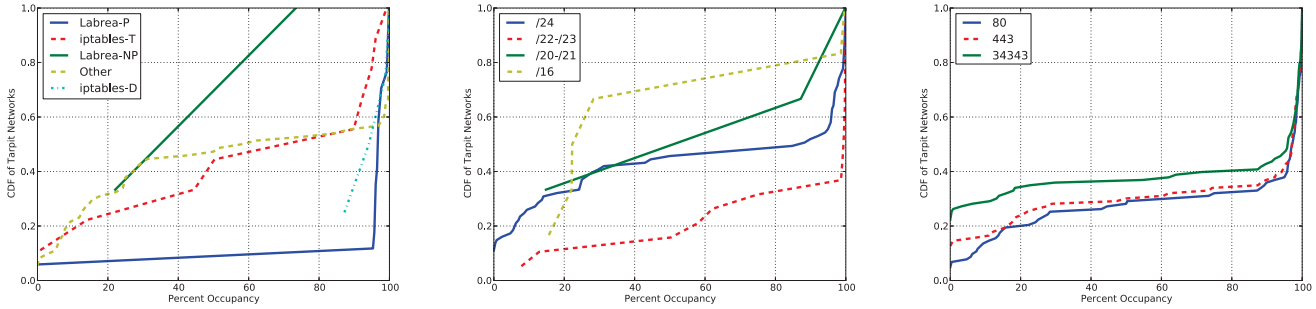
Figure 7: Latencies from probing addresses within a subset of our discovered Internet tarpits. Each tarpit is represented by an inter-quartile boxplot and obfuscated identifier. While the inter-quartile range is small, the outliers are significant.

Figure 8(b) again shows the cumulative fraction of tarpit networks versus their occupancy, but broken down by the subnet size. Approximately half of all /24's have an occupancy of 95% or greater, while more than 60% of the /22's and /23's have an occupancy of 95% or more. The occupancy's of the six /16's vary more widely; two of the /16's are fully occupied (more than 99%), while the other four are between 15-30% occupied.

Next, we note that many tarpits answer all TCP ports. To better understand the port-specific behavior of the tarpits we discover, we probe all addresses within each tarpit subnet for TCP ports 80, 443, and 34343. TCP port 34343 is not assigned to any service, and therefore would not typically be expected to respond. Figure 8(c) shows the cumulative fraction of tarpit networks versus their occupancy as a function of port number. Unsurprisingly, we observe that by probing port 80 we find a higher occupancy than port 34343. Of note however, is that the difference between port 80 and 34343 is relatively small, suggesting that most of the tarpits we find answer for all ports.

Finally, with a substantial list of subnets running network tarpits, we sought out ground truth to further confirm that our detection methodology is accurate. Since we were unable to directly find confirmation on any of the subnet owner's websites, we utilized Whois[3] records to make email contact. After waiting over two weeks, we had only received responses from two of the organizations we queried. One organization's response was the creation of a "trouble ticket," for which we never received further information. The second organization that responded was helpful and confirmed that they indeed were running LaBrea on the subnets in question.

The lack of ground truth and unresponsiveness of organizations suspected of running tarpits makes determining *degreaser*'s false-positive rate difficult to calculate. Lab testing resulted in 100% accuracy, however, due to the numerous configurable options in existing tarpit software and the pos-

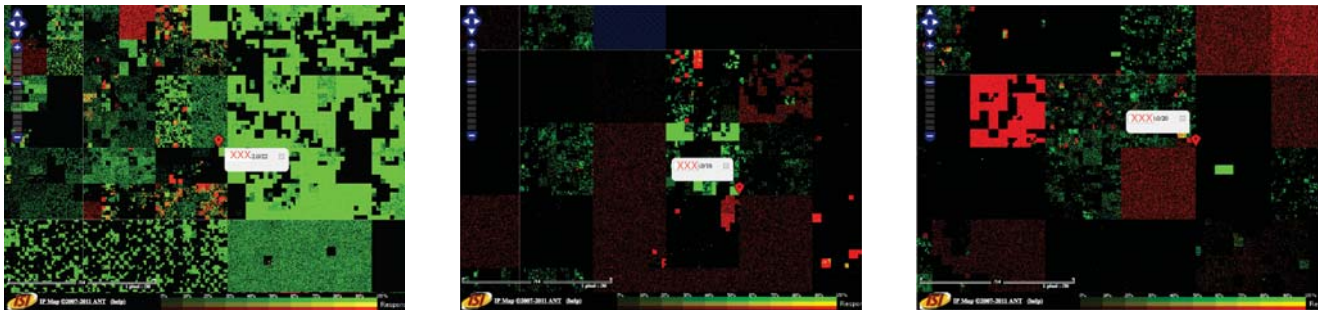


(a) Distribution of inferred tarpits as a function of their occupancy and tarpit type.

(b) Distribution of inferred tarpits as a function of their occupancy and network size.

(c) Distributing of inferred tarpits as a function of their occupancy and TCP port.

Figure 8: Inferred tarpit occupancy (fraction of addresses within network prefix acting as a tarpit).



(a) A fake /22 tarpit subnetwork in an otherwise high-occupancy region.

(b) 58 of the 256 /24 subnetworks within this large /16 aggregate are fake (23%).

(c) The only live addresses appearing in the highlighted /20 subnetwork are fake.

Figure 9: Visualizations of tarpits polluting the Internet address space. Data from USC/LANDER internet_address_census_it58w-20140122 [33], visualized with their IPv4 browser [27]. The red arrow anchor and white window label points to the fake subnetwork.

sibility of tarpits not based on the stock LaBrea software or the Netfilter TARPIT plugin, we can not claim perfect accuracy in the wild. We have not publicly disclosed our list of suspected tarpits due to security concerns, but we do encourage organizations or researchers interested in our work to contact the authors for access to our results.

4.3 Effect on Internet Scans

With several identified tarpitting subnets we explored how these tarpits were reflected across several different Internet scans [4, 2]. For scans that utilized ICMP-based approaches such as `ping`, we find these subnets appearing as fully occupied with responding hosts.

For example, Figure 9 shows screenshots of three sections of the IP address space as viewed from the ISI ANT Internet census browser [2]. The census browser visualizes subnet utilization by laying out subnets on a Hilbert Curve and then using a heatmap where increasingly bright green corresponds to proportionally higher occupancy. Dark regions indicate regions inferred to be empty, while red indicates negative replies. Figure 9(a) shows one of the /22 tarpit subnets we discover, within a region of otherwise high-occupancy. Figure 9(b) highlights a /16 aggregate where all of the green corresponds to the 58 of 256 tarpit /24 subnetworks within the larger aggregate. Last, Figure 9(c) shows a region of green surrounded by black – this green is a fake /20 tarpitting subnet within a larger aggregate that does not respond otherwise.

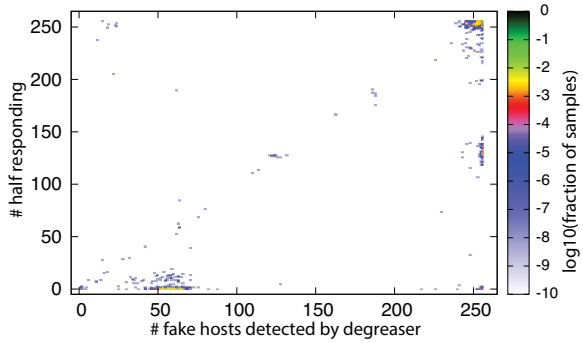


Figure 10: Density relationship of /24's: inferred tarpit occupancy vs. Project Sonar half-responding count.

Next, we examine the relation between the data we extracted from the logs of Project Sonar's HTTP scan from April 2014 [4] and our inferred tarpit subnets. To facilitate comparison, we broke each tarpit network into its respective /24's. For each /24, Figure 10 evaluates the number of half-responding IP addresses from the HTTP scan (introduced in §3.3) versus the number of tarpit IP addresses we infer.

We observe two dominant characteristics of the density plot. First, we see close agreement between the inferred oc-

cupancy's of many of the /24's between the two methods; these appear along the diagonal and also the large cluster in the upper right indicative of fully occupied tarpits. A small number of /24's with few tarpits but many half-responding addresses is present. We postulate that these addresses belong to tarpits that implement various forms of temporal timeouts and other non-deterministic behavior; §6 contains further discussion of these.

A second cluster of /24's with 40-70 tarpit addresses and few half-responding addresses raises suspicion that these are addresses missed by the Project Sonar probing, perhaps due to timeout thresholds. Similarly, a cluster of fully occupied tarpit /24's with 100-150 half-responding addresses also suggests at differences resulting from different probing methodologies. In general, we discover more addresses within our tarpits than are present in the Project Sonar's HTTP scan logs. In future work, we plan to more rigorously investigate these per-/24 discrepancies.

5. BUILDING A BETTER TARPIT

Network deception has proven to be an effective tool to thwart attackers. Our research has shown that in their current state, tarpit applications are relatively easy to detect and could easily be integrated into malicious scanning tools. Below we present a few recommendations to improve network tarpits to make them less easily distinguishable.

- ***TCP Options:*** Fingerprinting using TCP options can be overcome by simply to supporting TCP options. In Section 3.2.2 we showed that the relatively low number of TCP connections that do not use options is a key indicator of tarpit activity. An improved tarpit would respond to an incoming SYN with similar TCP options. Appending TCP options could be done easily with minimal performance degradation and still maintains the advantage of not require the tarpit to maintain per connection state.

- ***Window Obfuscation:*** Overcoming the TCP window size characteristic is a much more difficult task since setting a small window size is fundamental to operation of existing tarpit applications. The Netfilter TARPIT plugin takes advantage of shrinking the window during the three-way handshake. The SYN/ACK contains the hard-coded 5 byte window, but once the client sends the final ACK, the Netfilter plugin immediately responds with another ACK, this time shrinking the window to zero. This behavior, however, is easily detectable since it occurs immediately after the connection establishment is complete. An improved tarpit could take this concept and delay the window shrinking until later in the connection.

The improved tarpit would send its SYN/ACK with a sufficiently large window. The client would in turn attempt to send a data packet. The tarpit would examine the data packet and send an ACK, but only for part of the data and reduce the window by an amount larger than the size of the ACK. The client would attempt retransmission of the "lost" data and the process would continue until eventually the window is reduced to zero. Unlike supporting TCP options, this method requires the tarpit to maintain a receive window and thus inducing per connection state. The per connection state is minimal since the tarpit does not need to keep any of the partial data packets it has received. Additionally, once a zero window has been sent, all per connection state information can be discarded.

By shrinking the window over the course of several data packets, we continue to lure the client in to the trap, making immediate tarpit detection more difficult. Of note, the TCP standard specifically classifies "shrinking the window" as strongly discouraged behavior, however, for robustness requires conforming implementations to handle this behavior. While violating the recommendations of the standard, this robustness requirements assures that our improved tarpit would continue to effectively trap hosts.

- ***TCP Retransmissions:*** One final improvement is in exploiting the nature of TCP retransmission to improve the stickiness of the tarpit. Since most TCP implementations will attempt retransmission at least three times before closing the connection, an improved tarpit would discard the first two packets, and wait for the third before responding. This incurs the cost of having to remember unacknowledged packets for each TCP connection, but can effectively slow the connection by several RTTs. The space overhead of remembering packets could be reduced by only storing a hash of the TCP header under the assumption that the host will not change the header during a retransmit.

6. CONCLUSIONS

In this work, we build *degreaser*, a tool to infer the presence of fake tarpitting hosts and networks. By probing at least one address in each /24 network in the Internet, *degreaser* uncovers more than 100 different tarpit subnetworks. Notably six of the tarpits are /16's with two of the /16's using all of their addresses to tarpit. Overall, we find over 215,000 active IP addresses that are fake in 29 countries and 77 autonomous systems.

While the size and extent of tarpits we discover on today's Internet is small relative to the entire Internet, we are pleased that *degreaser* is able to discover these needles in a haystack. However, even small blocks of tarpit addresses can greatly slow automated scans as part of their intended capturing behavior.

Further, our results emphasize that cyber-deception is real and requires additional research attention. At present, it is unknown whether the deception we observe is security or policy related – for instance an attempt to influence address allocation policies. In general, it is an open question as to whether the use of deception is becoming more popular. We thus plan to run *degreaser* periodically in order to perform longitudinal study.

While we are confident that our classification of LaBrea and Netfilter-based tarpits are accurate, our scans reveal several other behaviors that are inconsistent with either type tarpit or real hosts. We encounter non-tarpitting hosts that accept TCP connections on well known ports, ACK data packets, but provide no application layer response (such as a HTTP Bad Request). Some of these hosts eventually terminate the connection using FINs, others do not. More exotic behaviors include networks that accept our connection attempts but after scanning several hosts, suddenly stop responding. A second attempt scanning the same network results in no successful connections, while scans from a different origin network are successful. The combined use of deception and temporal blacklisting warrants future study.

The effectiveness of tarpits is difficult to measure against the ever increasing range of attacks plaguing the Internet. This work has shown the simplicity in detecting tarpits and

we can only assume that as malicious scanning tools evolve, they will become more resilient to the effects of tarpits. Furthermore, we believe that our detection methodology could easily be incorporated into operating system TCP stacks in order to automatically skip tarpits, providing immunity to all network applications (abusive or otherwise).

Due to the negative effect these tarpitting subnets have on a variety of legitimate network scans, we suggest more explicit distribution of known-tarpits among whitehat communities in conjunction with using techniques developed in *degreaser*. Further, we presented avenues for future research into making tarpits more resilient to detecting and more believable to adversaries. We thus hope this work serves to raise awareness of a particular form of network deception that is popular in the wild, and its present-day implications.

Acknowledgments

We thank Mark Gondree, kc claffy, John Heidemann, and our reviewers for feedback. Nathan Desso helped with early experiments and coined the degreaser moniker. This work supported in part by the Department of Homeland Security (DHS) under contract N66001-2250-58231 and by U.S. NSF grant CNS-1228994. Views and conclusions are those of the authors and should not be interpreted as representing the official policies or position of the U.S. government or DHS.

7. REFERENCES

[1] The CAIDA UCSD Anonymized Internet Traces, 2013. `http://www.caida.org/data/passive/passive_2013_dataset.xml`.

[2] ANT Censuses of the Internet Address Space, 2014. `http://www.isi.edu/ant/address/`.

[3] IANA WHOIS Service, 2014. `http://www.iana.org/whois`.

[4] Internet-Wide Scan Data Repository, 2014. `https://scans.io/study/sonar.http`.

[5] L. Alt. Degreaser git respository, 2014. `https://github.com/lancealt/degreaser`.

[6] P. Baecher, M. Koetter, T. Holz, M. Dornseif, and F. Freiling. The nepenthes platform: An efficient approach to collect malware. In *Recent Advances in Intrusion Detection*, pages 165–184. Springer, 2006.

[7] J. Black and P. Rogaway. Ciphers with arbitrary finite domains. In *Topics in Cryptology–CT-RSA*, pages 114–130. Springer, 2002.

[8] X. Cai and J. Heidemann. Understanding block-level address usage in the visible internet. In *Proceedings of ACM SIGCOMM*, pages 99–110, 2010.

[9] B. Chun, D. Culler, T. Roscoe, A. Bavier, L. Peterson, M. Wawrzoniak, and M. Bowman. Planetlab: an overlay testbed for broad-coverage services. *ACM SIGCOMM Computer Communication Review*, 33(3):3–12, 2003.

[10] F. Cohen. A note on the role of deception in information protection. *Computers & Security*, 17(6):483–506, 1998.

[11] R. Craven, R. Beverly, and M. Allman. A Middlebox-Cooperative TCP for a non End-to-End Internet. In *Proceedings of ACM SIGCOMM*, pages 151–162, 2014.

[12] Z. Durumeric, E. Wustrow, and J. A. Halderman. Zmap: Fast internet-wide scanning and its security applications. In *Proceedings of the 22nd USENIX Security*, 2013.

[13] T. Eggendorfer. Reducing spam to 20% of its original value with a SMTP tar pit simulator. In *MIT Spam Conference*, 2007.

[14] M. Honda, Y. Nishida, C. Raiciu, A. Greenhalgh, M. Handley, and H. Tokuda. Is it still possible to extend tcp? In *Proceedings of the ACM SIGCOMM Internet Measurement Conference*, pages 181–194, 2011.

[15] A. Hopkins. iptables TARPIT target, 2014. `http://xtables-addons.sourceforce.net`.

[16] T. Hunter, P. Terry, and A. Judge. Distributed tarpitting: Impeding spam across multiple servers. In *LISA*, volume 3, pages 223–236, 2003.

[17] V. Jacobson, R. Braden, and D. Borman. TCP Extensions for High Performance. RFC 1323, May 1992.

[18] J. H. Jafarian, E. Al-Shaer, and Q. Duan. Openflow random host mutation: transparent moving target defense using software defined networking. In *Proceedings of Hot topics in software defined networks*, pages 127–132, 2012.

[19] J. Jung and E. Sit. An empirical study of spam traffic and the use of dns black lists. In *Proceedings of the ACM SIGCOMM Conference on Internet Measurement*, 2004.

[20] T. Kohno, A. Broido, and K. C. Claffy. Remote physical device fingerprinting. *Dependable and Secure Computing, IEEE Transactions on*, 2(2):93–108, 2005.

[21] T. Liston. Labrea, 2003. `http://labrea.sourceforge.net/labrea.1.txt`.

[22] G. F. Lyon. *Nmap Network Scanning: The Official Nmap Project Guide to Network Discovery and Security Scanning*. Insecure, 2009.

[23] D. Moore, C. Shannon, and J. Brown. Code-Red: a case study on the spread and victims of an Internet worm. In *Internet Measurement Workshop*, pages 273–284, Nov 2002.

[24] D. Moore, C. Shannon, G. Voelker, and S. Savage. Network Telescopes. Technical report, Cooperative Association for Internet Data Analysis (CAIDA), Jul 2004.

[25] G. Portokalidis, A. Slowinska, and H. Bos. Argos: an emulator for fingerprinting zero-day attacks for advertised honeypots with automatic signature generation. In *ACM SIGOPS*, volume 40, pages 15–27, 2006.

[26] J. Postel. Transmission Control Protocol. RFC 793, Sept. 1981.

[27] Y. Pradkin and J. Heidemann. Browsing the internet address space. Web site `http://www.isi.edu/ant/address/browse/`, Feb. 2008.

[28] N. C. Rowe. The ethics of deception in cyberspace. *Handbook of Research on Technoethics*, 2008.

[29] N. C. Rowe and H. C. Goh. Thwarting cyber-attack reconnaissance with inconsistency and deception. In *Information Assurance and Security Workshop*, 2007.

[30] C. Ruvalcaba. Smart IDS - Hybrid Labrea Tarpit. 2009. `http://www.sans.org/reading-room/whitepapers/detection/smart-ids-hybrid-labrea-tarpit-33254`.

[31] L. Spitzner. The honeynet project: Trapping the hackers. *IEEE Security & Privacy*, 1(2):15–23, 2003.

[32] S. Trassare, R. Beverly, and D. Alderson. A technique for network topology deception. In *MILCOM*, Nov. 2013.

[33] USC/LANDER Project. Internet ipv4 address space census. PREDICT ID `USC-LANDER/internet_address_survey_it58w-20140122`, Jan. 2014.

[34] A. G. West, A. J. Aviv, J. Chang, and I. Lee. Spam mitigation using spatio-temporal reputations from blacklist history. In *Proceedings of the 26th Annual Computer Security Applications Conference*, pages 161–170, 2010.

[35] V. Yegneswaran, P. Barford, and D. Plonka. On the design and use of internet sinks for network abuse monitoring. In *Recent Advances in Intrusion Detection*, pages 146–165, 2004.

[36] M. Zalewski. Passive OS fingerprinting tool, 2012. `http://lcamtuf.coredump.cx/p0f3/`.

[37] J. Zhuge, T. Holz, X. Han, C. Song, and W. Zou. Collecting autonomous spreading malware using high-interaction honeypots. In *Information and Communications Security*, pages 438–451. Springer, 2007.

Network Dialog Minimization and Network Dialog Diffing: Two Novel Primitives for Network Security Applications

M. Zubair Rafique
iMinds-DistriNet, KU Leuven
3001 Leuven, Belgium
zubair.rafique@cs.kuleuven.be

Juan Caballero
IMDEA Software Institute
Madrid, Spain
juan.caballero@imdea.org

Christophe Huygens
iMinds-DistriNet, KU Leuven
3001 Leuven, Belgium
christophe.huygens@cs.kuleuven.be

Wouter Joosen
iMinds-DistriNet, KU Leuven
3001 Leuven, Belgium
wouter.joosen@cs.kuleuven.be

ABSTRACT

In this work, we present two fundamental primitives for network security: network dialog minimization and network dialog diffing. Network dialog minimization (NDM) simplifies an original dialog with respect to a goal, so that the minimized dialog when replayed still achieves the goal, but requires minimal network communication, achieving significant time and bandwidth savings. We present network delta debugging, the first technique to solve NDM. Network dialog diffing compares two dialogs, aligns them, and identifies their common and different parts. We propose a novel dialog diffing technique that aligns two dialogs by finding a mapping that maximizes similarity.

We have applied our techniques to 5 applications. We apply our dialog minimization approach for: building drive-by download milkers for 9 exploit kits, integrating them in a infrastructure that has collected over 14,000 malware samples running from a single machine; efficiently measuring the percentage of popular sites that allow cookie replay, finding that 31% do not destroy the server-side state when a user logs out and that 17% provide cookies that live over a month; simplifying a cumbersome user interface, saving our institution 3 hours of time per year and employee; and finding a new vulnerability in a SIP server. We apply our dialog diffing approach for clustering benign (F-Measure = 100%) and malicious (F-Measure = 87.6%) dialogs.

Categories and Subject Descriptors

C.2.2 [**Computer Systems Organization**]: Network Protocols;
D.4.6 [**Operating Systems**]: Security and Protection

General Terms

Security

Keywords

Network dialog minimization, network dialog diffing, network security, network delta debugging.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ACSAC '14, December 08 - 12 2014, New Orleans, LA, USA.
Copyright 2014 ACM 978-1-4503-3005-3/14/12
http://dx.doi.org/10.1145/2664243.2664261 ...$15.00.

1. INTRODUCTION

A network dialog comprises the exchange of network traffic between peers to achieve a goal, which can be benign such as visiting your favorite website or downloading a file from a remote server, but also malicious such as exploiting a networked application. For example, a simple goal like visiting the root page of Amazon.com produces a complex network dialog that involves multiple remote servers and requires establishing as many as 41 TCP connections and exchanging up to 330 HTTP messages, used among others to download the HTML content, figures from CDNs, perform analytics, and display advertisements from ad networks.

Two fundamental primitives for network security are replaying a network dialog previously captured in a network trace and comparing network dialogs. Dialog replay enables applications such as testing which server versions are vulnerable to a exploit [11], evaluating intrusion detection systems [18], building interactive honeypots [23], and active botnet infiltration [5, 8]. And, comparing network dialogs enables clustering malware samples based on their network behavior [38] and building C&C signatures [16, 31].

Many tools have been proposed to replay traffic from a trace [7, 18, 28, 35] and prior work proposes techniques for replaying a network dialog regardless of the protocols involved [11, 27]. However, a network trace may contain much traffic unrelated to the replay goal. For example, a trace obtained at the network's boundary may capture an attack on an internal server that an analyst wants to replay. But, it likely contains much traffic (e.g., Gigabytes) from other hosts not attacked, and traffic from the attacked server unrelated to the attack, e.g., because it happens before or after the attack. Before replaying the attack an analyst may want to remove all unrelated traffic to better understand the attack, and also for efficiency since the attack traffic may be a small part of the full trace and the attack may need to be replayed many times (e.g., against different server versions).

In this work, we introduce the problems of *network dialog minimization* (NDM) and *network dialog diffing*. Network dialog minimization is the problem of given an *original dialog* that satisfies a goal, producing a *minimized dialog* comprising the smallest subset of the original dialog that when replayed still achieves the same goal as the original dialog. In essence, the minimized dialog provides a shortcut to the goal, removing all connections and messages in the original dialog unrelated to the goal. A minimized dialog enables understanding what parts of a dialog really matter for the goal e.g., determining which messages and fields are really required to exploit a network server without expensive code analysis. Furthermore, it provides huge reductions (up to 71 times) in time and band-

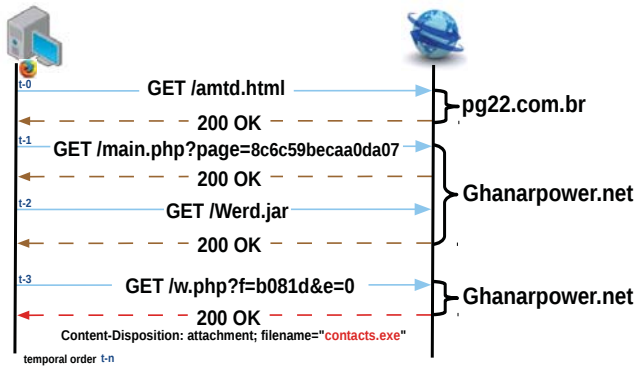


Figure 1: A drive-by download network dialog.

width to achieve the goal, which is fundamental when replaying a dialog repeatedly over time, e.g., in active botnet infiltration [5, 8].

In this work, we propose *network delta debugging* the first technique to automatically minimize a network dialog. Our insight is that NDM is a generalization of the problem of minimizing an input that crashes a program, for which delta debugging is a well-known solution [39]. Our network delta debugging technique generalizes delta debugging so that it can be applied to remote applications, allows application-specific goals beyond crashing an application, and takes advantage of the hierarchical structure of a network dialog.

Network dialog diffing is the problem of given two dialogs, identifying how similar they are, how to align them, and how to identify their common and different parts. In this work we propose a novel technique for dialog diffing, which aligns two network dialogs by finding a mapping between the dialogs that maximizes similarity. Once aligned, it identifies which elements (e.g., connections, messages) have been inserted, removed, and modified. We also define a dialog similarity metric that captures how similar two dialogs are.

We have applied our dialog minimization and diffing techniques to 5 applications. First, we apply NDM to build drive-by download milkers, which periodically replay a minimized drive-by download dialog to collect the malware an exploit server distributes over time. We have built milkers for 9 exploit kits and have integrated them into an infrastructure that has collected over 14,000 malware samples. A milker achieves a 34x reduction in replay time and their use enables reducing our malware collection infrastructure from three to a single host. Second, we apply NDM for measuring the current status of well-known cookie replay issues in popular websites [1, 4, 14, 17, 30]. The use of NDM achieves a 71x reduction in replay time, saving over 20 hours of processing each day. Our measurements show that despite years of research on the topic 31% of the top 100 Alexa domains do not destroy the server-side state when a user clicks the logout link. Thus, the user remains effectively logged in. For 17% of all Alexa top 100 sites their cookie can be replayed for more than a month, creating a very large window for session hijacking. Third, we use NDM for vulnerability analysis, finding a new vulnerability in the OpenSBC SIP server. Fourth, we apply NDM for simplifying an unnecessarily cumbersome proprietary web interface to move up and down a windows sunblind in an automated building, converting it to a simple command-line tool. The simplified interface saves up to 3 hours of time a year per employee, and $10,000 that the building vendor would charge for modifying the proprietary interface. Finally, we evaluate our dialog diffing approach for clustering similar dialogs. Our dialog clustering achieves 100% precision and 100% recall on 60 benign dialogs from the top 30 Alexa sites, and 100% precision with 78% recall when clustering 91 malware dialogs from 6 families.

Contributions:

- We introduce the problem of network dialog minimization (NDM) and present network delta debugging, the first technique to solve NDM.
- We propose a novel dialog diffing approach that finds a mapping that maximizes similarity between two network dialogs, and then identifies their common and different elements. We define a dialog similarity metric that leverages the alignment.
- We present a novel encoding of a network dialog as a tree, which captures the dialog's hierarchical structure of connections, messages, and fields.
- We apply our dialog minimization and diffing techniques to 5 applications: building drive-by download milkers, cookie expiration validation, simplifying user interfaces, vulnerability analysis, and dialog clustering.

2. OVERVIEW

A *network dialog* comprises the network traffic exchanged between peers for achieving a goal such as sending an email or installing malware in a victim's host through a drive-by download. At the core of this work is the intuition that a network dialog can be represented as a *network dialog tree* that captures the hierarchical structure of the network dialog, which comprises connections, messages, and fields in messages.

A network dialog tree enables automatically operating on a network dialog, e.g., replaying the network dialog, minimizing it, or comparing two dialogs. Figure 1 shows a simple drive-by download dialog where a victim visits a compromised site (`pg22.com.br`) that redirects the victim to an exploit server at `Ghanarpower.net`, which checks the browser and Java plugin version (GET /main.php), redirects the victim's browser to download a Java exploit (GET /Werd.jar), and if exploitation succeeds redirects the browser to download a malware sample (GET /w.php). Figure 2 shows the corresponding dialog tree, where the first level below the root corresponds to connections, the second level to messages in those connections, and deeper levels to the hierarchical field structure of each message. We detail the network dialog tree in Section 3.

The remainder of this section presents our novel operations on the network dialog tree (Section 2.1) and describes applications of those operations (Section 2.2).

2.1 Network Dialog Tree Operations

We propose two novel operations on network dialog trees: dialog minimization and dialog diffing.

Dialog minimization. One important use of network dialog trees is to automatically replay a previously captured network dialog. The network dialog can be periodically replayed to the same entities involved in the dialog or to different ones (e.g., by replacing their IP addresses / domain names). We take dialog replay one step further and introduce the problem of *network dialog minimization* (NDM), which is the process of given an *original dialog* that satisfies a *goal*, automatically producing a *minimized dialog* that, when replayed, achieves the same goal as the original dialog but with minimum traffic, i.e., removing all connections and messages not needed to achieve the goal. Further removal of elements from the minimized dialog causes the goal not to be reached.

For example, Figure 3 presents the minimized dialog for the drive-by download original dialog in Figure 1, where the goal is to download the malware sample being distributed. The minimized

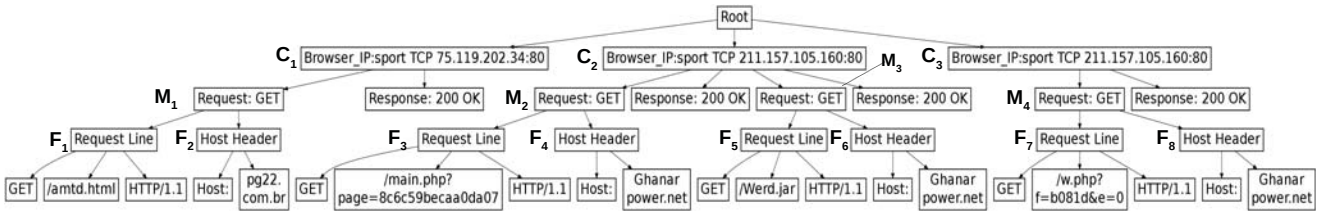


Figure 2: Network dialog tree for the dialog in Figure 1. For brevity, response nodes are not expanded.

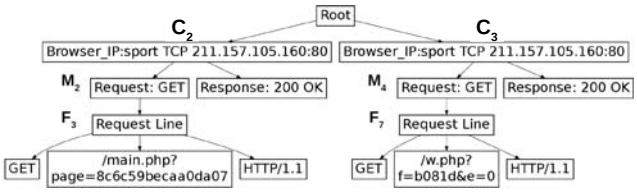


Figure 3: Minimized drive-by download dialog.

dialog shows that it is possible to download the malware by simply replaying two of the four requests in the original dialog.

Dialog diffing. Two dialogs that achieve exactly the same goal but are taken at different points in time, or from different end hosts, may intuitively be similar but may include a large number of divergences due to non-determinism. For example, a visit to Microsoft's MSN.com root page produces a complex network dialog with 82 connections connections and 177 requests. Visiting MSN.com again one day later produces a similar dialog but with 99 connections and 222 requests instead.

Network dialog diffing is the problem of given two network dialogs automatically identifying how similar they are, which parts are the same across them, and which parts different (i.e., connections and messages added, removed, or modified). Solving the dialog diffing problem implies solving the dialog similarity and dialog alignment problems as well. Our approach first defines similarity metrics between messages, connections, and dialogs. Then, it uses the similarity results to align both dialogs. Once aligned, common and different parts between both dialogs are output.

2.2 Dialog Tree Applications

This section introduces the 5 applications we use to demonstrate our techniques.

Building drive-by download milkers. A drive-by-download milker is a lightweight program that periodically downloads a malware sample from an exploit server. Its goal is similar to a honeyclient, which given a URL, visits it, gets exploited, and downloads the malware sample being distributed. But, a drive-by download milker does not go through the exploitation process and thus does not require running a heavyweight VM with a vulnerable browser; it simply replays a minimized drive-by download dialog. A drive-by download milker takes as input a minimized drive-by download dialog for a specific exploit kit (e.g., RedKit or CoolExploit), a set of exploit server IP addresses, and a time value specifying how often to replay the dialog. At each iteration it downloads a malware sample from each exploit server.

Drive-by download milkers need to be used in combination with honeyclients since it is not always possible to replay a drive-by download dialog. However, they achieve large performance savings. To combine them, we first run a honeyclient on a URL obtained from an external feed. If the URL leads to exploitation, the honeyclient outputs an original dialog, which our infrastructure immediately tries to minimize using our dialog minimization technique. If minimization succeeds, then it builds a signature on the URL of the exploit server so that next time a URL for the same exploit kit is seen a milker is used instead of a honeyclient. Our infrastructure periodically visits each exploit server every few hours and from different IP addresses. Thus, using a milker instead of a honeyclient quickly adds up in performance gains.

We have used this combination of drive-by download milkers and honeyclients to collect over 14,000 malware samples during one year from a single machine. The malware collection results and their analysis have been introduced in prior work [26]. However, that work did not present the network dialog minimization technique at the core of our milkers. In addition, our prior work only built milkers for two exploit kits (BlackHole 1.x and Phoenix), while this work builds milkers for 9 (Section 6.1).

Cookie expiration validation. Most web applications generate an authentication token upon successful validation of a user's credentials (e.g., username and password) and embed it into a cookie that is stored in the user's browser. The user login is typically protected by HTTPS. However, a significant fraction of web applications (37% of the top 100 Alexa sites) use plain HTTP for subsequent connections because of performance and compatibility issues. Since the unencrypted cookie is included in following requests to the server, a network attacker (e.g., on a public WiFi) can capture the cookie and include it in its own connections to hijack the session, impersonating the user [1,4,13,14,17,30]. This enables the adversary to access and change the user's sensitive information and to steal the account by changing the user's password.

To mitigate cookie replay attacks, the web server should expire the authentication token in the cookie after some time forcing the user to re-authenticate afterwards (thus minimizing the time window for an adversary to launch a session-hijacking attack). Here, it is not enough to delete the cookie in the user's browser when she clicks on a logout link. The web application also needs to remove the authentication token on the server-side, otherwise the attacker can still replay it, even if the user has (apparently) logged out.

While these problems have been known for over a decade [14, 30], it is necessary to understand if web applications are addressing it and how fast. However, manually checking cookie expiration times for many web applications is cumbersome because each application's cookie has a different format and may be encrypted. The most general way to test cookie expiration times is to replay the cookie periodically until the server stops accepting it. For this, we use our dialog minimization technique on a valid authenticated dialog for each application under test and automatically replay the minimized dialog at fixed intervals.

We have applied our approach to the top 100 Alexa sites that do not use full HTTPS communication, measuring how many sites allow cookie replay and for how long (Section 6.2). Our measurements show that 31% of the top 100 Alexa domains still do not destroy the server-side state when a user clicks the logout link and that 17% have cookies the live over a month, creating a large opportunity window for session hijacking.

Simplifying user interfaces. Web applications often require the user to go through several manual steps (e.g., traversing multiple webpages, filling forms) to reach an interesting state. Network dialog minimization can be used to produce a shortcut to that state that removes unnecessary steps and which can be automatically replayed. This simplifies application access, e.g., converting a web based application into a command line tool. Simplified user interfaces save user time and also provide benefits for disabled users [21]. For example, one of our institutions recently moved to a brand new fully automated building. This building has no manual switches; tasks like turning on the lights, moving sunblinds, and setting the temperature of an office can only be performed through a web application that interfaces with a building automation system (BAS). A simple task like moving the sunblind down when the sun hits the computer screen, requires accessing three different webpages (login, office selection, value setting) through a cumbersome interface. Unfortunately, the web application and the BAS interface are proprietary and undocumented. Any changes to them requires expensive work by the BAS manufacturer. We have applied our dialog minimization technique to generate minimized dialogs for moving a sunblind up and down, and a command line tool that replays them (Section 6.3). Employees can now perform these tasks more efficiently saving 3 hours per person and year.

Vulnerability analysis. Replaying an attack on a vulnerable networked application is a useful capability that enables testing the vulnerability and determining which versions are vulnerable [11]. In addition, minimizing a network attack dialog enables understanding the conditions leading to exploitation without expensive code inspection, and the minimized dialog can be used as an exploit signature [12]. Furthermore, minimization is fundamental to isolate the real attack when the input is a network trace (e.g., captured at an IDS) that contains an attack on a vulnerability but also much other traffic unrelated to the attack. We have applied our network dialog minimization technique to minimize an attack on the OpenSBC SIP server (Section 6.4). The minimized dialog still exploits a patched version of the OpenSBC server unveiling a previously unknown vulnerability, which has been acknowledged by the author and assigned an OSVDB identifier [29].

Dialog clustering. Our dialog diffing approach defines a similarity metric between network dialogs that enables clustering multiple similar dialogs at once. We have evaluated dialog clustering on 60 benign and 91 malicious dialogs (Section 6.5). The benign dialogs correspond to visiting the top 30 Alexa sites twice, spacing the visits one day apart. Our clustering groups with perfect accuracy and recall the dialogs from the same site in the same cluster, despite the many changes between both dialogs due to non-determinism. The malicious dialogs are obtained by running malware, capturing their C&C communication. Here, the clustering achieves 100% accuracy and 78% recall. The perfect accuracy enables integrating our clustering as a first step into existing signature generation tools [31, 33].

3. NETWORK DIALOG TREE

A network dialog may comprise multiple connections, each with multiple messages on both dialog directions, and each message with its own field hierarchy. The network dialog tree captures this hierarchical structure. The nodes of the tree correspond to connections, messages, or fields depending on the depth in the tree. An edge from a parent node to a child node implies that the child belongs to the parent, e.g., that a message belongs to a specific connection or a field to a specific message. Nodes at the same depth and with the same parent (e.g., all messages in a connection) respect a temporal ordering in which the beginning of a node (e.g., the first packet in a message or the first byte in a field) appeared on the network earlier than the start of the node on its right.

The root represents the complete dialog and is annotated with the list of peers involved in the dialog (at least two but possibly more), each peer characterized by an IP address, a DNS domain, or both.

Nodes at depth 1 correspond to connections. Each connection node has 7 attributes: source and destination peers, source and destination ports, transport protocol, application protocol, and the timestamp of the first packet. We build UDP "connection" nodes by grouping all UDP packets between two peers until no traffic is seen between them for a predefined window of time. The source peer corresponds to the initiator of the connection. Connections are ordered using the timestamp of their first packet.

Nodes at depth 2 correspond to the messages in the connection. Each message node is annotated with the peer that sends the message, which captures its direction. For UDP connections, each packet is considered a separate message. Messages in the same connection are ordered using the timestamp of their first packet.

The nodes at depth ≥ 3 correspond to fields in the message. Each field node is annotated with the range of bytes it occupies in the message and fields can only have completely disjoint or completely enclosed ranges (i.e., fields cannot partially overlap). Field nodes can be *hierarchical fields* if they have children or *leaf fields* otherwise. The children of a hierarchical field cover disjoint parts of the parent's range. For example, every HTTP GET message in Figure 2 contains a Request-line field that in turn contains 3 other fields: the method, the URI, and the protocol version. Field nodes are ordered using the start offset of the field in the message it belongs to (offset zero is the first byte).

Node dependencies. A network dialog tree also captures field dependencies, e.g., a field may be the length or checksum of another field in the same message. Dependencies can also happen across messages, e.g., the value of a Cookie field received from a web server in an HTTP response may be sent in a later HTTP request to the same server. There can also be dependencies across connections, e.g., the Referer and Location HTTP headers indicate that a request was originated by a redirection in a previous response.

Protocol knowledge. The network dialog tree is a general abstraction that can be used to represent any network dialog regardless of the protocols involved. However, only the construction of the first level, i.e., extracting the connections nodes, is protocol-independent. Extracting the message and field levels requires knowledge about the application protocol used in the connection. We assume the availability of the grammar of each protocol involved. When the grammar is not available, prior work has shown how to recover it from an implementation of the protocol [6, 10, 24, 37]. In this paper, we focus on HTTP and SIP dialogs since these are the only protocols needed by our applications.

4. DIALOG MINIMIZATION

Minimizing a network dialog is an intuitive process: create a *test dialog* by removing some part of the original dialog, replay the test dialog, and check if the goal is still achieved. If the test *passes* (i.e., the goal is still achieved) the part removed is not needed to achieve the goal and thus should not be in the minimized dialog. If the test *fails* (i.e., goal not achieved), it is needed and should be kept in the minimized dialog. The main goals are that the minimized dialog should not contain unnecessary parts and that the number of tests to produce the minimized dialog should be as small as possible.

Our insight is that network dialog minimization (NDM) is a generalized version of the problem of minimizing an input that crashes

a program, for which *delta debugging* is a well known solution [39]. The differences are that (1) NDM deals with remote networked applications, which we may not control, rather than a local program; (2) NDM's goal is not necessarily crashing the remote application, but more generally to drive it to some externally visible state; and (3) NDM operates on sequences of connections and messages rather than a file or a single network packet. We detail how we address these differences next.

Reset button. The first difference is that in input minimization the analyst controls the program for which the input is being minimized. More specifically, the analyst can restart the program before each test. Thus, the tests are independent because the program starts each test in the same initial state. However, NDM deals with a remote server which we may not control because it may belong to another party, e.g., an exploit server. This is problematic because when we perform a test, the remote server may be left in a state different from its initial state. Thus, the next test starts in a different server state and the tests are not independent. Without independent tests, the minimization may produce an incorrect result. For example, the minimized drive-by download dialog in Figure 3 shows that messages M_2 and M_4 are both needed to download the malware. If a test consisted only of message M_2 and the next test only of message M_4, the second test would reach the goal thanks to the server state set by the first test, which would be incorrectly interpreted as only M_4 being needed to reach the goal.

The solution to this problem is to define an application-specific *reset button* that the minimization process can use to reset the remote server to its initial state before each test. We propose one reset button implementation, applicable to many network server types, but other implementations are possible.

The intuition behind our reset button is that a network server handles multiple simultaneous clients by keeping a separate state machine for each. Thus, if the replayer uses a previously unused IP address for every test, then at the start of a new test every endpoint will be in its initial state. For each level of the tree with n nodes, the minimization may generate up to $n^2 + 3n$ tests, so that our reset button may require a large IP address pool. In addition, the remote application may limit the geographical location of IP addresses it serves. To identify this case, the minimization first conducts a geographical distribution test (GDT) where it replays the full original dialog from one IP address in each region. If some regions are blocked, the reset button is modified to utilize only IP addresses in the allowed regions.

To implement the reset button we use a commercial Virtual Private Network (VPN) that offers exit points in more than 50 countries, each with a pool of IP addresses, totaling more than 45,000 IPs. Each test changes the VPN observable IP. In our applications we never run out of IP addresses. For very large trees, NDM supports limiting the minimization to a certain tree depth.

Goal function. The second difference is that NDM minimizes the original dialog with respect to a goal that may not be crashing the remote network application. For example, in the drive-by download dialog in Figure 1 the goal is downloading a malware binary from the exploit server. NDM takes as input a boolean *goal function* that given the responses received from the peers during the test, returns true if the goal was achieved in the test, false otherwise. The implementation of the goal function is dialog-specific. In some applications it is enough to check the received responses, e.g., whether a response from an exploit server contains an executable. Other goal functions may be more complex, e.g., for vulnerability analysis it may check if the application crashed through local monitoring or by sending a probe to see if the application still responds.

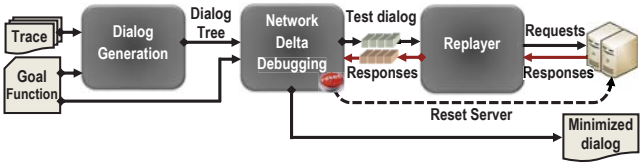


Figure 4: Network dialog minimization architecture.

Dialog tree. The third difference is that NDM does not deal with minimizing a single input, but rather a sequence of network connections and messages. Our insight is that by encoding a network dialog as a tree we can use a hierarchical version of delta debugging to minimize the dialog [25]. Capturing the hierarchical structure of the network dialog enables applying delta debugging (ddmin) at each level of the tree. Intuitively, this greatly speeds up minimization because only properly-formatted inputs are produced and because if a test removes the subtree rooted at one node and the test passes, then the complete subtree requires no more tests.

Architecture overview. Figure 4 shows the architecture of NDM, which comprises 3 modules: *dialog generation*, *network delta debugging*, and *replayer*. The dialog generation module runs once. It takes as input a network trace and the goal function. It builds a network dialog tree from the traffic in the trace and uses the goal function to remove any connections and messages after the goal was reached, since they are unrelated to the goal. The core of NDM is the network delta debugging module, which takes as input the original dialog tree, the goal function, and the reset button. It implements an iterative process. At each iteration it produces a test network dialog by removing some elements from the original dialog. The test dialog is passed to the replayer module, which replays it to the remote peers and passes the responses back to the network delta debugging module. The responses are checked with the goal function to determine if the test passed or failed, and a new test dialog is produced for the next iteration. This iterative process eventually outputs the minimized dialog.

Network delta debugging. Network delta debugging applies ddmin at each level of the network dialog tree from the root to the leaves, using the goal function to determine if a test passed and the reset button to restart the server state after each test. In a nutshell, ddmin executes the following 3 steps:

1. Reduce to subset: Split the current configuration into n partitions. Test each partition. If a partition reaches the goal, treat it as the current configuration and go to 1.
2. Reduce to complement: Test the complement of each partition. If any reaches the goal, treat it as the current configuration and go to 1.
3. Increase granularity: Split the current configuration into $2n$ partitions, where n is the current number of partitions, and go to 1. If the configuration cannot be split further, the algorithm outputs the current configuration as the minimized dialog.

Figure 5 details the minimization process for the dialog tree in Figure 2. First, it applies ddmin to the connections (level 1). Here, ddmin splits the connections into 2 subsets and tests each subset separately (tests §1–2). The first test contains connection C_1 (i.e., all messages in C_1) and fails to reach the goal. The second test contains $\{C_2, C_3\}$ and this time the goal is reached. The current configuration is set to $\{C_2\}, \{C_3\}$; the subtree rooted at C_1 will no longer be tested. Next, it reduces the current configuration into 2 subsets, testing C_2 and C_3 individually (§3–4), both failing to reach the goal. Since the complements have been tested and the granularity cannot be increased, level 1 minimization outputs C_2 and C_3. Next, ddmin is applied to the messages (level 2) of the remaining

	Test cases			Goal	
	Level 1				**Starting NDM**
1	C_1	.	.	no	**Testing {C_1}, {C_2, C_3}**
2	.	C_2	C_3	yes	**Reduce to {C_2, C_3}**
3	.	C_2	.	no	**Testing {C_2}, {C_3}**
4	.	.	C_3	no	
Res.	.	C_2	C_3	done	**Level 1 minimized. Removing C_1 and its children from dialog tree.**
	Level 2				
5	M_2	.	.	no	**Testing {M_2}, {M_3, M_4}**
6	.	M_3	M_4	no	**Increase granularity**
7		M_3	.	no	**Testing {M_3}, {M_4}**
8	.	.	M_4	no	
9	M_2	.	M_4	yes	**Testing complements, Reduce to {M_2, M_4}**
10	M_2	.	.	no	**Testing {M_2}, {M_4}**
11	.	.	M_4	no	
Res.	M_2	.	M_4	done	**Level 2 minimized. Removing M_3 and its children from dialog tree.**

Figure 5: Minimization of the dialog in Figure 2.

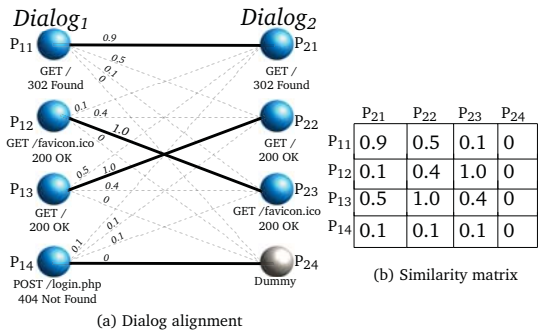
connections. The messages are split into 2 subsets, both failing to reach the goal (§5–6). Here, ddmin increases the granularity to $2n$, halving the message subsets. Thus, M_3 and M_4 are tested separately, both failing (§7–8). Note that M_2 had already been tested in §5. Next, the complement of M_3 is tested reaching the goal (§9), so the current configuration is reduced to $\{M_2\}, \{M_4\}$, but both fail (§10–11). At this point, level 2 minimization finishes determining that messages M_2 in C_2 and M_4 in C_3 are required to reach the goal. We leave level 3 minimization as an exercise to the reader. Figure 3 shows the resulting minimized dialog.

5. DIALOG DIFFING

Our dialog diffing approach first computes similarity scores between messages in both dialogs. Then, it aligns the dialogs by finding an optimal mapping between their messages. Once aligned, common and different parts are output and the dialog similarity score is computed. The two dialogs are compared on their structure and message content. Similar dialogs are identified despite changes in endpoints (domains, IPs), message reordering, and different number of connections and messages in both dialogs. Endpoints often change with malware since C&C domains and IPs are frequently updated, and also with benign dialogs if CDNs or load balancers are involved. Many HTTP dialogs contain asynchronous connections producing alternative message and connection orderings, which do not affect the dialog semantics. Furthermore, while different number of messages make the dialogs less similar, it is still possible to flag similarity if the spurious messages are few.

RRP similarity. Rather than comparing individual messages, our approach compares HTTP request-response pairs (RRPs). RRP similarity uses 11 features:

1. Request type: same HTTP method in both requests.
2. URL path: same path in both requests' URL.
3. URL filename: same filename in both requests' URL.
4. URL parameters: Jaccard index of the sets of parameter names (without values) in the URLs of both requests.
5. Referer: same value of the Referer header in both requests.
6. Response type: same response code and response status in both responses.



	P_{21}	P_{22}	P_{23}	P_{24}
P_{11}	0.9	0.5	0.1	0
P_{12}	0.1	0.4	1.0	0
P_{13}	0.5	1.0	0.4	0
P_{14}	0.1	0.1	0.1	0

Figure 6: Example of dialog alignment (a) and similarity matrix (b) for two dialogs with 4 and 3 messages respectively.

7. Server header: same value in both responses.
8. Location header: same value in both responses.
9. Content-Type header: same value in both responses.
10. Content length: computes $1 - \frac{||L_1 - L_2||}{max(L_1,\ L_2)}$ where L_1, L_2 are the content lengths for both responses (after joining chunks and decompressing, if needed).
11. Response headers: Jaccard index between the set of response headers (without values).

Features 1–3, 5–9 are boolean, while 4, 10, and 11 output a value in [0,1]. RRP similarity simply computes the average of the RRP features: $RRPsim(RRP_1, RRP_2) = \frac{1}{11} \sum_i f_i(RRP_1, RRP_2)$.

Message alignment. The first step to align the messages in both dialogs is building an $N \times N$ matrix, where N is the maximum number of nodes in either dialog, and a row captures the similarity between an RRP in dialog 1 and all RRPs in dialog 2. If a dialog has less messages than the other, it is padded with dummy RRPs, and the similarity of any RRP with a dummy RRP is zero. This matrix can be represented as a complete bipartite graph $G = (V_1 \cup V_2, E)$ where $v_i \in V_1$ corresponds to an RRP in dialog 1 and $v_j \in V_2$ to an RRP in dialog 2. The bipartite graph is complete because there is an edge between each RRP in one dialog and all RRPs in the other dialog ($v_i \in V_1, v_j \in V_2$) weighted by their RRP similarity score. Figure 6 shows the bipartite graph (a) and similarity matrix (b) for two dialogs with 4 and 3 messages respectively, the latter padded with one dummy RRP ($N = 4$).

Our insight is that aligning the messages in both dialogs corresponds to the *assignment problem* in graph theory, which finds the mapping in a weighted bipartite graph that maximizes the total weight of the mapping. The Hungarian algorithm [22] solves this problem in polynomial time. The solid lines in Figure 6a correspond to the alignment output by the Hungarian algorithm.

The reader may realize that our approach aligns messages without considering connection structure. This is to handle misaligned connections, e.g., a connection with 4 messages in one dialog may align with 2 connections with 2 messages each in the other dialog.

Dialog diffing. For the diffing, RRPs aligned with similarity 1.0 are considered identical. RRPs aligned with high similarity ($s \geq 0.7$) are considered changed. RRPs aligned with low similarity ($s < 0.7$) are considered new. Threshold selection is explained in Section 6.5. In Figure 6, the diffing outputs two identical RRPs, one changed, and one new.

Dialog similarity. The dialog similarity is computed by summing the weights of all alignment edges and dividing by the maximum number of nodes in a dialog:

$$sim(D_1, D_2) = \frac{1}{N} \sum_{i=1}^{N} w_i \quad (1)$$

In Figure 6, $sim(D_1, D_2) = 2.9/4 = 0.725$.

Dialog Generation					Network Delta Debuging							Milking Time	
Exploit Kit	Pre-filtering Nodes	Pre-filtering C:M:F	Filtered C:M:F	Filtered IPs	L1 C:M:F	L2 C:M:F	L3 C:M:F	Tree Nodes	IPs used	GDT Pref.	Time (sec.)	Honeyclient (sec.)	Milker (sec.)
BlackHole 1.x	73	6:6:60	5:5:50	2	2:2:22	2:2:22*	2:2:6	11	33	✓	157.0	53.5	11.1
CoolExploit	646	18:58:569	5:5:49	2	1:1:7	1:1:7*	1:1:3	6	15	✗	42.5	118.8	1.2
CritiXPack	192	4:19:168	2:7:62	2	1:4:33	1:1:7	1:1:3	6	17	✗	49.0	46.8	1.3
Eleonore	936	12:76:848	8:66:736	2	1:1:8	1:1:8*	1:1:4	7	27	✗	215.8	238.0	3.3
Phoenix	132	12:12:107	7:7:73	1	1:1:7	1:1:7*	1:1:3	6	15	✗	24.2	20.8	2.2
ProPack	137	10:12:114	6:6:57	2	1:1:7	1:1:7*	1:1:3	6	15	✗	37.3	94.1	0.4
RedKit	154	8:17:128	2:6:57	1	2:6:57	2:2:19	2:2:10	15	71	✓	250.4	25.9	0.5
Serenity	54	5:5:43	5:5:43	1	2:2:15	2:2:15*	2:2:6	11	28	✗	79.7	96.2	0.8
Unknown	79	5:7:66	5:7:66	2	1:2:14	1:1:7	1:1:3	6	18	✗	51.0	29.6	0.4

Table 1: Summary of network dialog minimization of drive-by downloads dialogs.

6. EVALUATION

In this section we evaluate our dialog minimization and dialog diffing techniques for 5 different applications.

6.1 Building Drive-by Download Milkers

Our drive-by download malware collection infrastructure runs a honeyclient on any suspicious URL received from external feeds. If the honeyclient is exploited, it outputs a network trace. The infrastructure then applies NDM on the honeyclient dialog, to check if a milker can replace the honeyclient for increased performance.

Goal. The goal function checks if the body of a HTTP response contains an executable file (after joining chunks and decompressing, if needed) since in the drive-by download dialogs examined the binary was not encrypted. If the exploit server drops more than one executable, the goal function can be set to output true after the first executable is seen, or after a specified number of binaries is received using a conjunction of constraints.

Results. Table 1 details the dialog minimization for the 9 exploit kits for which we built milkers. The left part of Table 1 summarizes the honeyclient dialogs (responses not included). It shows the number of nodes in the dialog tree built from the honeyclient network trace; the split of those nodes into connections, messages, and fields (`C:M:F`); the remaining nodes when messages and connections after the goal are filtered (e.g., C&C communication after exploitation); and the number of endpoints (IPs) in the filtered dialog. Filtering reduces the number of messages by a factor of 1.8 on average and up to a factor of 11 (CoolExploit). The filtered dialogs comprise 2–8 connections, 5–66 messages, and 1–2 peers.

The middle part of Table 1 summarizes the 3-level minimization on the filtered dialogs. It shows the connections, messages, and fields left after each minimization level (L1–L3); the number of nodes in the minimized dialog; the number of IPs (i.e., tests) used in the minimization; whether the exploit kit distributed samples geographically; and the minimization runtime. For 6 kits L2 minimization is not run (marked with asterisk) because connections remaining after L1 minimization contain at most one request each. The minimized dialogs for 6 kits contain a single request, i.e., the malware can be directly downloaded. For BlackHole 1.x and RedKit two messages are left, indicating that the first request sets server state. For Serenity, 2 connections are required because it drops 2 malware samples.

L3 minimization results show that the majority of HTTP headers are not needed to download the malware. For 7 kits, only the Request-Line (HTTP method, URI, and protocol version) is required. In addition, Redkit requires Referer and User-agent headers in both requests and Eleonore the Host header. This shows that NDM reveals which fields are checked by the server. The 3-level minimization reduces the nodes in the original tree by a factor of 39 on average, and up to a factor of 162 for Eleonore. The geographical distribution test (GDT) shows that 7 kits drop the malware to visitors from any country. BlackHole and Redkit did not serve malware for IPs in China. After GDT, only 27 IPs (i.e., tests) were needed on average to minimize a dialog. The Time column shows the minimization runtime in seconds, when run on a 32-bit 3.3GHz host with 4 cores and 4GB RAM. On average, 3-level minimization took 1.7 minutes, the slowest being RedKit with 4.2 minutes. Tree size is the dominant runtime factor, but latency and whether the server keeps state (i.e., L2 minimization needed) also matter.

There were 2 other exploit kits (BlackHole 2.x and Neutrino) for which minimization failed due to replay protection through dynamically generated URLs. For these exploit kits we used honeyclients.

Savings. The right side of Table 1 shows the milking time when using a honeyclient and a milker. Using a milker is 34 times faster than using a honeyclient. These values only include the time for network communication. The overall savings are larger because the honeyclient requires an additional 10 seconds to load the snapshot and transfer the URL, and runs for 4 minutes after initialization to allow exploitation to complete. Furthermore, each VM uses 512 MB, so the number of VMs that can simultaneously run on a host is memory constrained. In practice, milkers enabled us to reduce our milking infrastructure from 3 hosts down to a single host that runs all milkers and honeyclients.

6.2 Cookie Expiration Validation

In this experiment, we measure how many of the top 100 Alexa websites are affected by cookie replay attacks, leading to session hijacking, and the expiration time of their cookies. As preparation, we create a user account in each of the websites. Then, we obtain the original network dialog by logging into the website using those credentials, accessing the user profile, and logging out.

Goal. To determine if the user logged in successfully, the goal function checks if a server response includes the username string; all evaluated sites include it in the web page sent to the user after successful login.

Minimization and replay. 2-level minimization is applied to each dialog. Then, the minimized dialog is automatically replayed every 30 minutes, each time checking if the goal is reached. If a replay does not reach the goal, the replayer outputs the cookie lifetime.

Results. Table 2 summarizes the cookie replay experiments. Of the top 100 Alexa websites, 53 are not vulnerable because they either use full HTTPS (42) or have no user login (11). For other 10 we were unable to create a user account due to stricter requirements, e.g., having a cell phone from a specific region. Cookies can be replayed for 37 of the top 100 Alexa sites[1]. All 37 sites destroy the

[1]Some of those (e.g., `youtube.com`, `linkedin.com`) can be accessed fully through HTTPS, but by default use HTTP.

	Cookie Replays						No Replay		
Total Websites	Replay login	Replay remote	Replay logout	Duration > 2 days	Duration > 1 week	Duration > 1 month	Comp. HTTPS	No login	Others
100	37	36	31	27	22	17	42	11	10

Table 2: Summary of cookie replay results for Alexa Top 100.

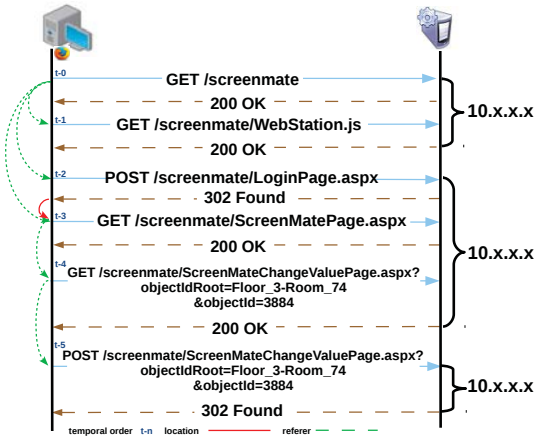


Figure 7: Building Automation Dialog.

client-side cookie when the user clicks the logout link. However, only 6 sites destroy the server state after the user clicks the logout link. For the other 31 sites, cookies can still be replayed after the user believes he has logged out. One site (`vk.com`) does not allow cookie replay from an IP address different from the one used to login. Of the 37 sites for which replay works, for 27 the cookie still replays after 2 days, for 22 after a week, and for 17 after a month.

Our experiment flags two issues. First, 31% of popular sites do not destroy the server-side state when a user clicks the logout link. Thus, the user is effectively still logged in. In addition, for 17% of the sites the cookie replays for more than a month, creating a very large window in which session hijacking is possible.

On average a minimized dialog replays in 0.6 seconds compared to 42.6 seconds for the original, a 71 times reduction. Each site is visited 48 times a day, so minimization saves over 20 hours of processing each day.

6.3 Simplifying User Interfaces

In this experiment we simplify the cumbersome proprietary web interface to move a sunblind up/down in our brand new automated building, into a simpler command line tool. To begin, we capture two original dialogs corresponding to a user going through the web interface to move the sunblind down and up, respectively. Figure 7 shows the sunblind-up dialog, which requires 6 HTTP request-response pairs during which the user visits the service (an additional 2 unsuccessful requests are not included), logs in with a username and password; selects the office, sunblind item, and event type; and finally rises the sunblind.

Goal. The goal function checks whether the server redirects the user to a webpage that indicates that the command was successful.

Minimization. The minimized dialog comprises a single POST request to the web service, with the request line, the session cookie, and a payload defining the sunblind-up event. The replayer is able to replay this dialog over time without updating the cookie value, which indicates that the cookie does not have any expiration time.

We have built a 10-line C wrapper for the replayer that takes a parameter to indicate if the sunblind-up or sunblind-down dialog should be replayed. On average, a user moves the blind twice a day and each use of our tool saves 22 seconds. This amounts to significant savings of 3 hours per person each year. More importantly, we save $10,000 that the building vendor would charge for simplifying their proprietary web interface.

6.4 Vulnerability Analysis

Dialog minimization and replay is a useful capability in vulnerability analysis. In this experiment, we use our dialog minimization technique to replay a previously published attack on the OpenSBC SIP server [32]. Surprisingly, the minimization finds another, previously unknown, attack on the OpenSBC server.

To obtain the original dialog we deploy a test OpenSBC server, configured in UDP stateful proxying mode without authentication, and use SIPp [34] to simulate both a SIP user agent client (UAC) and a SIP user agent server (UAS). Then, we use the available attack tool against the test server, producing an attack trace.

Goal. To check server availability after each test, the goal function sends a benign request to the server, verifying it still responds.

New attack. The 3-level minimization outputs that a single 74-byte SIP INVITE request is needed to crash the server (Figure 8). However, the minimized input does not contain the `Via` header, whose flawed processing created the original vulnerability. Replaying the minimized input against a version of OpenSBC that patches the original vulnerability, still crashes the server. We have reported the new vulnerability to the OpenSBC author, who has confirmed that it corresponds to a null-pointer dereference in the parser. The vulnerability has been assigned an OSVDB identifier [29]. This example illustrates that beyond performance savings when replaying, the minimized dialog is also useful for capturing the essential constraints to reach and exercise a vulnerability, without expensive code analysis. It can also be used as an exploit signature.

6.5 Dialog Clustering

This experiment describes the use of our dialog similarity metrics for clustering benign and malicious dialogs.

Datasets. Our benign dialog dataset contains 60 dialogs obtained by visiting twice the root page of each Alexa Top 30 website, one day apart. Each pair of dialogs achieves the same goal but contains changes introduced by non-determinism and the time difference such as message reordering, additional messages, and small content modifications. Four sites simply redirect a user to the HTTPS version, which produces short dialogs with 1–2 RRPs. On average each benign dialog comprises 27.8 connections and 66.3 messages, the largest being a dialog for `163.com` with 177 connections and 355 messages. These numbers highlight how complex web application dialogs can be.

The malicious dialog dataset comprises 91 dialogs obtained by executing 91 labeled malware binaries from 6 families (`cleaman`, `qakbot.ae`, `malagent`, `spyeye`, `zbot`, and `zeroaccess`), once each, in a contained environment that allows only HTTP C&C traffic to reach the Internet. On average, each malware family dialog comprises 2.0 connections and 2.6 messages, the largest being a dialog for `spyeye` with 4.5 connections and 6.0 messages.

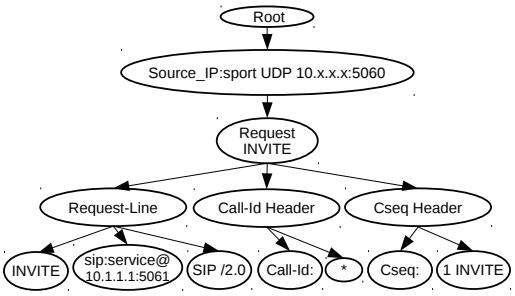


Figure 8: Minimized SIP INVITE request revealing previously unknown vulnerability on the OpenSBC SIP server.

Similarity matrix. The first step is building the similarity matrix by computing the similarity between each pair of dialogs. The average similarity between two dialogs from the same website is 0.96 and the minimum 0.80, which shows that our metric correctly captures their similarity despite changes due to non-determinism. The average similarity for dialogs from different sites is 0.16 and the maximum is 0.73 (between `facebook` and `twitter` which have only 1 RRP), which shows that the dialog similarity metric properly differentiates between similar and different dialogs. For the malicious dialogs the average similarity between dialogs from the same family is 0.74 and the minimum 0.41. The average similarity for dialogs from different families is 0.25 and the maximum is 0.73 (between `zeroaccess` and `zbot`).

Clustering algorithms. We use two different clustering algorithms: partitioning around mediods (PAM) [20] and aggressive. PAM takes as input the similarity matrix and the number k of clusters to output, so we run it with different k values, selecting the one which maximizes the silhouette width, an internal measure of clustering quality [20]. Our aggressive clustering starts with zero clusters and iterates on the list of dialogs. For each dialog, it checks if its similarity is larger than 0.8 with any dialog already in a cluster. If the comparison holds only for dialogs in the same cluster, it adds the dialog to that cluster. If it holds for dialogs in multiple clusters, it merges those clusters and adds the current dialog to the merged cluster. Otherwise, it creates a new cluster for it.

To select the 0.8 threshold we measured the clustering accuracy on a subset of the dataset for each threshold value between [0,1] with a step of 0.1 [15]. A threshold of 0.8 achieved best results.

Clustering results. Table 3 shows the clustering results for benign and malicious dialogs using both algorithms. For the benign dialogs, both algorithms output 30 clusters, each with the 2 dialogs for the same website, achieving perfect precision and accuracy. For the malware dialogs, the precision is perfect but the recall is 64.8%-78.0% respectively. This is because the same malware family is split into multiple clusters. There are two main reasons for this. First, zbot is a malware kit and zeroaccess an affiliate program. In both cases owners/affiliates may configure the malware differently, which creates differences captured by our metric. In addition, our metric fails to detect similarity for short dialogs with highly polymorphic requests. This is why signature generation tools use traffic clustering only as a first step before analyzing other information like content and endpoints [31, 33]. Given its perfect accuracy, our dialog clustering could be integrated as a first step in those tools.

Dataset	Algor.	Clusters	Precision	Recall	F-Measure
Alexa	PAM	30	100%	100%	100%
Malware	PAM	10	100%	64.8%	78.6%
Alexa	Agg.	30	100%	100%	100%
Malware	Agg.	12	100%	78.0%	87.6%

Table 3: Dialog clustering results.

7. RELATED WORK

Traffic replay. A number of tools have been proposed to replay traffic from a network trace [7, 18, 28, 35]. NDM differs from traffic replay in that the aim is not cloning the original dialog on a new setting, but rather to deliberately produce a different, minimized, dialog that achieves the same goal with minimal traffic. Other works automatically identify and adjust state-dependant fields, e.g., cookies and IP addresses, in the traffic to be replayed, regardless of its protocol, using network-based [11] and dynamic binary analysis [27] approaches. Our goal is not replaying any protocol, but rather minimizing a dialog or comparing two dialogs. The minimization achieves significant performance savings and enables understanding the minimum set of constraints to achieve a goal, while dialog diffing is a very different problem. Our approach assumes the protocol grammar is available, which is true in all our applications. If unknown, protocol reverse-engineering techniques can be used to recover its grammar [6, 10, 24, 37], state-machine [9], and application session structure [19]. Also related is ShieldGen [12], which generates vulnerability data patches by modifying an attack and replaying it to an oracle. A key difference with this work is that NDM operates on a complete dialog rather than a packet.

Drive-by download malware collection. Honeyclients are a popular approach for collecting malware distributed through drive-by downloads [36]. We develop drive-by download milkers that periodically replay a previous dialog for the same goal. Drive-by download milkers are more lightweight but less flexible than honeyclients, being specific to an exploit kit. Our drive-by download infrastructure [26] uses a combination of honeyclients and drive-by download milkers to achieve both flexibility and efficiency.

Cookie replay. Session hijacking through cookie replay has been known for more than a decade [14, 30]. Attacks on WiFi have been demonstrated [17] and tools are available to exploit it [1, 4]. It has also been mentioned that cookie replay still works after a user logs out [4, 17]. A large number of techniques have also been proposed to prevent session hijacking attacks [2, 3, 13, 14, 30]. Our work does not discover any new attacks. Instead, we show how dialog minimization and replaying tools provide an efficient and convenient way to automate the periodic visiting of a large number of sites to determine for how long cookie replays works, measuring the current state of these issues.

Traffic clustering. A number of works perform malware clustering based on their traffic [16, 31, 33, 38]. Most of these works cluster traffic at the packet level [16, 31, 33]. Most related is Nemean [38], which generates semantics-aware network signatures, clustering similar connections and sessions using request/response types and byte distribution features. Our work is different in that we propose a dialog alignment technique that finds a mapping between dialogs that maximizes similarity.

8. CONCLUSION

In this work, we have introduced the problems of network dialog minimization and network dialog diffing. We have proposed network delta debugging, the first technique to solve network dialog minimization. We have also proposed a novel technique for network dialog diffing, which aligns two dialogs by finding a mapping between them that maximizes similarity.

We have demonstrated our techniques for 5 applications: building drive-by download milkers for 9 exploit kits, measuring cookie replay on popular websites, simplifying cumbersome proprietary web interfaces, vulnerability analysis, and clustering benign and malicious dialogs.

9. ACKNOWLEDGMENTS.

This research was started while M. Zubair Rafique was an intern at the IMDEA Software Institute. This research was partially funded by the Research Fund KU Leuven, by the EU FP7 project NESSoS (FP7-ICT No.256980), and by the Prevention of and Fight against Crime Programme of the European Union (B-CCENTRE). This research was also partially supported by the Spanish Government through Grant TIN2012-39391-C04-01 and a Juan de la Cierva Fellowship for Juan Caballero. All opinions, findings, and conclusions or recommendations expressed herein are those of the authors and do not necessarily reflect the views of the sponsors.

10. REFERENCES

[1] Firesheep. `http://codebutler.com/firesheep`.

[2] B. Adida. Sessionlock: Securing web sessions against eavesdropping. In *International World Wide Web Conference*, 2008.

[3] A. Bittau, M. Hamburg, M. Handley, D. Mazieres, and D. Boneh. The case for ubiquitous transport-level encryption. In *USENIX Security Symposium*, 2010.

[4] S. Bowne. Cookie re-use in office 365 and other web services, 2013. `http://samsclass.info/123/proj10/cookie-reuse.htm#steps`.

[5] J. Caballero, C. Grier, C. Kreibich, and V. Paxson. Measuring pay-per-install: The commoditization of malware distribution. In *USENIX Security Symposium*, 2011.

[6] J. Caballero, H. Yin, Z. Liang, and D. Song. Polyglot: Automatic extraction of protocol message format using dynamic binary analysis. In *ACM Conference on Computer and Communications Security*, 2007.

[7] Y.-C. Cheng, U. Hölzle, N. Cardwell, S. Savage, and G. M. Voelker. Monkey see, monkey do: A tool for tcp tracing and replaying. In *USENIX Annual Technical Conference*, 2004.

[8] C. Y. Cho, J. Caballero, C. Grier, V. Paxson, and D. Song. Insights from the inside: A view of botnet management from infiltration. In *USENIX Workshop on Large-Scale Exploits and Emergent Threats*, 2010.

[9] P. M. Comparetti, G. Wondracek, C. Kruegel, and E. Kirda. Prospex: Protocol specification extraction. In *IEEE Symposium on Security and Privacy*, 2009.

[10] W. Cui, J. Kannan, and H. J. Wang. Discoverer: Automatic protocol description generation from network traces. In *USENIX Security Symposium*, 2007.

[11] W. Cui, V. Paxson, N. C. Weaver, and R. H. Katz. Protocol-independent adaptive replay of application dialog. In *Network and Distributed System Security Symposium*, 2006.

[12] W. Cui, M. Peinado, H. J. Wang, and M. Locasto. shieldgen: Automatic data patch generation for unknown vulnerabilities with informed probing. In *IEEE Symposium on Security and Privacy*, 2007.

[13] I. Dacosta, S. Chakradeo, P. Traynor, and M. Ahamad. One-time cookies: Preventing session hijacking attacks with disposable credentials. *ACM Transactions on Internet Technology*, 12(1), 2012.

[14] K. Fu, E. Sit, K. Smith, and N. Feamster. Dos and don'ts of client authentication on the web. In *USENIX Security Symposium*, 2001.

[15] R. Gras, E. Suzuki, F. Guillet, and F. Spagnolo. *Statistical Implicative Analysis*. Springer, 2008.

[16] G. Gu, R. Perdisci, J. Zhang, and W. Lee. Botminer: Clustering analysis of network traffic for protocol and structure independent botnet detection. In *USENIX Security Symposium*, 2008.

[17] Hamster, 2007. `http://blog.erratasec.com/2007/08/sidejacking-with-hamster_05.html#.UwdUz4ZDuKk`.

[18] S.-S. Hong and S. F. Wu. On interactive internet traffic replay. In *International Symposium on Recent Advances in Intrusion Detection*, 2006.

[19] J. Kannan, J. Jung, V. Paxson, and C. E. Koksal. Semi-automated discovery of application session structure. In *Internet Measurement Conference*, 2006.

[20] L. Kaufman and P. J. Rousseeuw. *Finding Groups In Data: An Introduction To Cluster Analysis*, volume 344. John Wiley & Sons, 2009.

[21] S. Keates. Designing user interfaces for ordinary users in extraordinary circumstances: A keyboard-only web-based application for use in airports. *Universal Access in the Information Society*, 12(2), 2013.

[22] H. W. Kuhn. The hungarian method for the assignment problem. *Naval Research Logistics Quarterly*, 2(1-2), 1955.

[23] C. Leita, K. Mermoud, and M. Dacier. scriptgen: An automated script generation tool for honeyd. In *Annual Computer Security Applications Conference*, 2005.

[24] Z. Lin, X. Jiang, D. Xu, and X. Zhang. Automatic protocol format reverse engineering through context-aware monitored execution. In *Network and Distributed System Security Symposium*, 2008.

[25] G. Misherghi and Z. Su. Hdd: Hierarchical delta debugging. In *International Conference on Software Engineering*, 2006.

[26] A. Nappa, M. Z. Rafique, and J. Caballero. Driving in the cloud: An analysis of drive-by download operations and abuse reporting. In *SIG SIDAR Conference on Detection of Intrusions and Malware & Vulnerability Assessment*, 2013.

[27] J. Newsome, D. Brumley, J. Franklin, and D. Song. Replayer: Automatic protocol replay by binary analysis. In *ACM Conference on Computer and Communications Security*, 2006.

[28] Ostinato. `http://goo.gl/oo5rmn`.

[29] OSVDB. opensipstack opensbc.exe null pointer dereference remote dos, 2012. `http://osvdb.org/86607`.

[30] J. S. Park and R. Sandhu. Secure cookies on the web. *Internet Computing, IEEE*, 4(4), 2000.

[31] R. Perdisci, W. Lee, and N. Feamster. Behavioral clustering of http-based malware and signature generation using malicious network traces. In *Symposium on Networked System Design and Implementation*, 2010.

[32] M. Z. Rafique, M. A. Akbar, and M. Farooq. Evaluating dos attacks against sip-based voip systems. In *Globecom*, 2009.

[33] M. Z. Rafique and J. Caballero. firma: malware clustering and network signature generation with mixed network behaviors. In *International Symposium on Recent Advances in Intrusion Detection*, 2013.

[34] SIPp. `http://sipp.sourceforge.net/`.

[35] Tcpreplay. `http://tcpreplay.synfin.net/`.

[36] Y.-M. Wang, D. Beck, X. Jiang, R. Roussev, C. Verbowski, S. Chen, and S. King. Automated web patrol with strider honeymonkeys: Finding web sites that exploit browser vulnerabilities. In *Network and Distributed System Security Symposium*, 2006.

[37] G. Wondracek, P. M. Comparetti, C. Kruegel, and E. Kirda. Automatic network protocol analysis. In *Network and Distributed System Security Symposium*, 2008.

[38] V. Yegneswaran, J. T. Giffin, P. Barford, and S. Jha. An architecture for generating semantics-aware signatures. In *USENIX Security Symposium*, 2005.

[39] A. Zeller and R. Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Trans. Softw. Eng.*, 28(2), 2002.

Lightweight Authentication of Freshness in Outsourced Key-Value Stores

Yuzhe Tang
Syracuse University
Syracuse, NY USA
ytang100@syr.edu

Ting Wang
IBM Research Center
Yorktown Heights, NY USA
tingwang@us.ibm.com

Ling Liu
Georgia Tech
Atlanta, GA USA
ling.liu@cc.gatech.edu

Xin Hu
IBM Research Center
Yorktown Heights, NY USA
huxin@us.ibm.com

Jiyong Jang
IBM Research Center
Yorktown Heights, NY USA
jjang@us.ibm.com

ABSTRACT

Data outsourcing offers cost-effective computing power to manage massive data streams and reliable access to data. Data owners can forward their data to clouds, and the clouds provide data mirroring, backup, and online access services to end users. However, outsourcing data to untrusted clouds requires data authenticity and query integrity to remain in the control of the data owners and users.

In this paper, we address the authenticated data-outsourcing problem specifically for multi-version key-value data that is subject to continuous updates under the constraints of data integrity, data authenticity, and "freshness" (i.e., ensuring that the value returned for a key is the latest version). We detail this problem and propose INCBM-TREE, a novel construct delivering freshness and authenticity.

Compared to existing work, we provide a solution that offers (i) lightweight signing and verification on massive data update streams for data owners and users (e.g., allowing for small memory footprint and CPU usage for a low-budget IT department), (ii) immediate authentication of data freshness, (iii) support of authentication in the presence of both real-time and historical data accesses. Extensive benchmark evaluations demonstrate that INCBM-TREE achieves higher throughput (in an order of magnitude) for data stream authentication than existing work. For data owners and end users that have limited computing power, INCBM-TREE can be a practical solution to authenticate the freshness of outsourced data while reaping the benefits of broadly available cloud services.

1. INTRODUCTION

In the big data era, data sources generate data of large variety, volume, and at a high arrival rate. Such intensive data streams are widely observed in system logs, network monitoring logs, social application logs, and many others. In order to efficiently digest the large data streams, which can be beyond a regular data owner's computing capability, outsourcing data and computation to clouds becomes a promising approach. Clouds can provide sufficient storage and computing capabilities with the help of large data centers and scalable networked software. By delegating processing, storing, and query serving of data streams to a third-party service, the data outsourcing paradigm not only relieves a data owner from the cumbersome management work but also saves significant operational cost.

For example, stock exchange service providers, social networking companies, and network monitoring companies can benefit from outsourcing their streaming data to clouds. In a stock exchange market, stock buyers and sellers make deals based on the changing price. To identify stock market trends, stock buyers may frequently consult exchange providers about historical and real-time stock prices. With a large number of stocks, the footprint of the stock price data would easily grow out of a regular company's computing capability or its IT budget. Moreover, as there are more and more stock brokers in the market, it requires huge computing power to serve such a large customer base. Another example is a social networking website where the stream of social application events arrive at a high rate, which can easily exceed the limit of the server capability of the operating company. Big data streams can be also observed in a network monitoring scenario where a company monitors its real-time network traffic.

Despite its advantages, data outsourcing causes issues of trust, because the cloud, being operated by a third-party entity, is not fully trustworthy. A cloud company could deliver incomplete query results to save computation cost or even maliciously manipulate data for financial incentives, e.g., to gain an unfair advantage by colluding with a data user competing with the rest. Therefore, it is imperative for a data owner to protect data authenticity and freshness when outsourcing its data to a third-party cloud.

It is crucial to assure *temporal freshness* of data, i.e., obtain proofs that the server does not omit the latest data nor return out-of-date data. Especially when the value of the data is subject to continuous updates, it is not sufficient to guarantee only the *correctness* of data because a data user expects to obtain the "freshest" data. For example, in the online stock exchange, a broker is interested in the latest price of a stock.

While there are different types of data authenticity (e.g. basic data integrity, completeness for range queries [17, 16], etc) in various outsourced systems, data freshness is particularly challenging to authenticate. To break the data freshness, malicious clouds can simply return stale but properly signed data versions. The problem of freshness authentication can be complicated by the presence of 1) big data and 2) accessibility to historical data, as observed in many cloud applications. In order to ensure the users of data freshness, it is critical for the owner to remember the latest data

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.
ACSAC '14, December 08 - 12 2014, New Orleans, LA, USA
Copyright 2014 ACM 978-1-4503-3005-3/14/12 $15.00
http://dx.doi.org/10.1145/2664243.2664244 .

Table 1: Comparing INCBM-TREE with prior work

Outsourced systems	Approaches	Big data	Historical access	Real-time verification	Efficient signing
Data stores	CloudProof [26], SPORC [12]	+	–	–	+
	Iris [29], Athos [13]	+	–	+	–
Streams	CADS [25], ProofInfused [17]	–	–	+	+
Highly dynamic stores	INCBM-TREE	+	+	+	+

acknowledged by the cloud; in the case of big data, such "latest snapshot" of the whole dataset may be simply too big to maintain locally, making it impractical for the owner to remember. On the other hand, accessing historical data (i.e. not only the latest snapshot) is required by many workloads; for example, predictive analysis needs to look at the history to predict the future, or consistency control protocols need to know the latest version at a historical time. Authenticating freshness in the presence of historical data access could even increase the size of local state to maintain on the owner side, because it needs to maintain not only the snapshot but also the full update history of the dataset. In this perspective, we summarize the existing work in Table 1 where symbol +/– denotes that an approach support/does not support a feature. In the table, we consider two additional features for freshness authentication: 3) real-time detection and 4) data write efficiency. Existing audit-based approaches [26, 12, 19] rely on an offline process to detect the violation; such approaches can not guarantee the freshness in real time. Existing real-time detection approaches, such as Iris [29], require an up-to-date local state (e.g. root hash of a remote Merkle tree) on the owner side, which can be expensive to maintain (i.e. each data update requires to read a partial Merkle tree from the cloud and then to locally update the root).

In this paper, we propose a novel authentication framework for multi-version key-value data stores. Our freshness authentication is based on the owner's promises to frequently sign the updates within certain time interval; such assumption is practical in the context of intensive data updates. In this framework, we formalize the problem of freshness authentication to be a non-membership test; for instance, the freshness of a data record updated 5 minutes ago can be authenticated by the non-membership fact that there were no updates of the data in the last 5 minutes. We formally describe our problem and authentication framework in §2 and §3, and introduce a novel construct INCBM-TREE in §4 to address the non-membership test problem. An INCBM-TREE uses a Bloom filter [7] for freshness authentication while enabling lightweight signing and optimized verification. Conceptually, an INCBM-TREE is a Merkle Hash tree (MHT) [21] that embeds a hierarchy of Bloom filters (BFs). In an INCBM-TREE, an MHT signs and protects authenticity of data streams along with their associated BFs, whereas BFs are used in non-membership tests for verifying version freshness. Furthermore, we design INCBM-TREE in such a way that it can be incrementally constructed and maintained so that signing data stream can be done efficiently without reading the historical data from the cloud.

In summary, our main contributions are as follows.

- To the best of our knowledge, we are the first to solve the problem of *efficiently outsourcing multi-version key-value stores with verifiable version freshness*, enabling devices with limited computation capabilities to leverage cloud-based data management while ensuring the freshness and authenticity of the outsourced data.
- We propose a novel construct INCBM-TREE to authenticate version freshness, which dramatically reduces freshness verification overhead and signs the data stream efficiently with small memory footprint.
- We evaluate the implementation of INCBM-TREE and our results confirm that it applies to generic key-value stores, offering more throughput (in an order of magnitude) for data stream authentication than existing work.

2. PROBLEM FORMULATION

2.1 System Model

Our system model, illustrated in Figure 1, considers a data outsourcing scenario; it involves a single data owner (e.g., a small enterprise) and multiple data users (e.g., employees or customers of the enterprise), who are bridged through a public cloud.

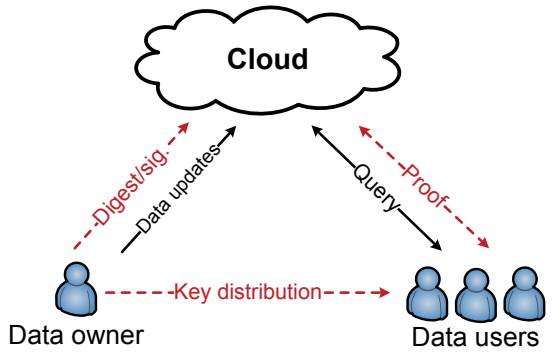


Figure 1: System model and authentication framework

Specifically, in this scenario the basic data flow (as black solid arrows in Figure 1) starts from the data source [1], which generates the big data stream. The semantics of the data stream is that each stream unit is an update to a record in the outsourced dataset in the cloud. The data updates are uploaded to the cloud and applied there.

The data users issue queries against the outsourced data in the cloud. In this work, we assume an SaaS/PaaS cloud model (i.e. Software/Platform-as-a-Service); the cloud provides a managed storage service accessible through the Put/Get API, such as the Amazon S3 service and many other key-value stores [9, 2, 1].[2] With the API, we consider a key-value data model where each object has a unique key k, and multiple overwriting values $\{v\}$, each of which is associated with a unique timestamp t. The cloud storage API is formally described as follows. Note that the timestamp t_q is optional in Get; when absent, it means that Get needs to return the newest version as of now (i.e. the query time).

$$\mathsf{Put}(k, v, t) \rightarrow \text{ACK} \quad (1)$$

$$\mathsf{Get}(k[, t_q]) \rightarrow \langle v, t \rangle \quad (2)$$

[1]We use "data owner" and "data source" interchangeably.

[2]This paper focuses on simple data access, while the support and optimization for advanced queries (e.g. aggregations) can be future work of this paper.

2.2 Threat Model

We assume the minimal trust in our system that data users only trust the data owner.

The public cloud, administrated in a third-party domain, is untrusted. In our model, the cloud can do anything with the outsourced data at its proposal. To break data freshness, the cloud can launch a replay attack [18] to return properly signed but stale data when processing the Get queries. The incentives of mounting such attacks could be many, such as to save computation costs. While the malicious behavior could occur, our authentication framework (as will be described below) guarantees that such behavior can not occur without being noticed by the data users.

Based on our query and threat models, there are specifically two desirable security properties to be guaranteed.

DEFINITION 2.1. *Given a query* Get$(k[,t_q])$*, the result returned from the cloud, say* $\langle v,t\rangle$*, is*

- fresh *if and only if* $\langle v,t\rangle$ *is the latest version updated before* t_q *and*
- correct *if and only if* $\langle v,t\rangle$ *is indeed a version that belongs to the key* k *and was originally submitted by the data owner.*

Table 2: Notations

k: key	v: value
t: timestamp	B: record size
S: memory size	b: KOMT batch size
q: memory ratio for KOMT	l: INCBM-TREE depth
E: error rate for non-membership test	
E_b: bound for E	r: reporting time interval

3. AUTHENTICATION FRAMEWORK

We have described the basic data flow in our system model, and in this section we describe an extra data flow for authenticating this basic flow. The key-value data stream, after being generated from the data source, is locally digested and then signed. As shown by the red dashed arrow in Figure 1, the digest and signatures are uploaded to the cloud. After issuing a query request and receiving the result from the cloud, the data user needs proofs to verify the result authenticity; such proofs are constructed by the cloud. To make the users be able to verify, we assume a key-distribution channel from the owner to the users.

3.1 Flexibility

In our framework, the data source is in the enterprise domain, while the data users can be flexible, either inside or outside the domain. For example, if a data user is an employee of the enterprise, s/he is inside the domain, and if the user is an enterprise customer, s/he can be outside the domain. For the users inside the enterprise domain, the authentication framework may use symmetric key to sign and verify the data for performance concerns; such keys are shared securely between the data source and users in the domain. If the data needs to be shared with the users outside the enterprise domain, then the framework can use the public key infrastructure to authenticate the data and to distribute the public key to the users. The users outside the enterprise domain may maliciously collaborate with the public cloud to forge a proof for compromised data. It is thus necessary to use the public keys which gives users only the ability to verify the data, but not the one to sign the data.

3.2 Freshness Authentication and Non-Membership Test

Our framework primarily focuses on the freshness authentication[3]. Given the design of Get API, it is required to authenticate the result freshness for now or for a historical time t_q. In particular, for freshness as of now, the challenge comes from that the time "now" is constantly evolving; the freshness of every record needs to be constantly reported, even when there is no update on the record. Our framework supports to authenticate relaxed real-time freshness at a time interval of r minutes (e.g. $r = 1$ or $r = 0.1$). We assume that the owner promises to report the authenticated data updates every r minutes. This is a reasonable assumption, and it does not incur dramatically huge overhead onto the owner; in our system, the owner (e.g. an enterprise) is ingesting a high-rate data stream and unlike some other offline owners [12] it needs to be $24/7$ online anyway. In addition, we assume that the physical time on all participating parties (including the data owner, the cloud and data users) is synchronized. Such synchronization can be realized by a reliable trusted time service[4]. Under the promise and assumption, the user can easily authenticate data freshness relaxed by a r-minute interval; that is, the absence of a proper freshness proof for data more than r minutes ago directly means the misbehavior of the cloud. The cause of such misbehavior could be various; the cloud may be unreliable and (unintentionally) lose the data outsourced by the owner, or the cloud may deliberately neglect the fresh data for economic reasons.[5]

The above mechanism of promised frequent reporting provides sufficient conditions to support that a behaving cloud *can* prove the data freshness. In the following, we briefly describe *how* data freshness is verified. Proving the freshness of data $\langle v,t\rangle$ as of time t_q (or now) is equivalent to saying that no newer version $\langle v',t'\rangle$ exists such that $t' \in (t,t_q]$. Formally, given that the owner's data stream is signed in time intervals, it needs to authenticate two facts:

- **Membership of** $\langle k,v,t\rangle$: The outsourced data that is reported in a time interval covering the queried time t indeed contains the key-value data of key k.
- **Non-membership of** $\langle k,v',t'\rangle$: The outsourced data whose time interval falls in $(t,t_q]$ (i.e. the interval's older time bound postdates t and newer time bound predates or equals t_q) does not contain any key-value data of key k.

It is thus essential to design a digest structure that enables both the membership and non-membership tests, which is described in the next section.

4. INCBM-TREE BASED DIGEST

In this section, we describe our digest. We will first overview the digest, then specifically describe the proposed INCBM-TREE, and then the materialization of the digest in our authentication framework. After introducing the mechanism, we will describe policies to optimize the system performance. At the end, we will describe the system implementation and deployment.

4.1 Multi-Level Digest

To enable the (non)-membership test, a baseline digest is to use the Key-ordered Merkle Hash Tree, or KOMT [21, 17, 16, 29, 13].

[3]Correctness can be easily authenticated using the standard techniques, such as MAC or digital signatures [16].

[4]http://tf.nist.gov/tf-cgi/servers.cgi

[5]We consider the use of a strongly consistent data store in the cloud; that is, the data, once acknowledged being successfully written to the cloud, is made immediately available to the subsequent queries. Real-world data stores of strong consistency include BigTable [9], HBase [2], and Cassandra [1] under certain configuration.

Given a batch of b key-value records, the KoMT digest is constructed by sorting the records based on data key and then building an MHT on top of the ordered list. The KoMT is not well suited for our scenario of outsourcing intensive data streams, due to performance concerns. Specifically, KoMT's performance depends on the size of the data batch, b. With one setting of b, KoMT may achieve efficiency in either signing or verification, but not both (e.g. for a small b, there will be a large number of batches and digests which increases the number of non-membership tests for verification; or for a large b, the owner may need to hold a huge local state which grows out of its memory space).

In this paper, we propose a multi-level digest for efficient key-value data authentication. Our digest is a hierarchical structure of four levels (from the bottom level to the top): 1) a first-level KoMT which is built for each (small) time interval r and serves for the purposes of real-time reporting, 2) a second-level KoMT with batch size b configured based on available memory size, 3) IncBM-Tree that is built on top of the second-level KoMT; it uses Bloom filters to index non-membership information across KoMT's, 4) the chain of root hashes of the previously digested IncBM-Trees. In the following, we specifically describe the structure of proposed IncBM-Tree.

Design goals. IncBM-Tree's design considers the presence of both historical and real-time data access. In this context, its design goals are: (1) to minimize the amount of data stored on the data-owner side for lightweight data signing, (2) to reduce the proof size for efficient verification of data freshness on the data-user side,

4.2 IncBM-Tree Structure

The basic idea behind the IncBM-Tree is that a Bloom filter can summarize a data set in a space-efficient way and thus facilitates the non-membership test. The structure of an IncBM-Tree is illustrated in Figure 2. Comparing to the traditional Merkle tree, each tree node in the IncBM-Tree maintains not only a hash digest but also a digest value that summarizes the key set in the subtree rooted at the node. For a key set of the subtree, the digest includes a Bloom filter and a value range. For instance, a leaf node 6 maintains a Bloom filter BF_6 summarizing key 1 and key 12 under the node and a hash digest h_6. Given node 3 which is the parent of two leaf nodes (node 6 and node 7), its digest is a Bloom filter of union of its children nodes' Bloom filters, namely $BF_3 = BF_6 \cup BF_7$. Considering numeric data, the range digest is simply with lower bound 1 and upper bound 23, namely $R_3 = [1, 23]$. It comes from merging the ranges from its two children, that is, $[1, 12] \cup [15, 23]$ (note that there are no data in $[12, 15]$). In total, the digest of the tree node is the hash value of concatenation of all its children's hashes, the range digest, and the Bloom filters, that is, $h_3 = H(h_6 \| h_7 \| BF_3 \| R_3)$. Note that in an IncBM-Tree, Bloom filters at different levels are of the same length. Formally, the IncBM-Tree uses the following constructs.

$$\begin{aligned} R(\text{node}) &= R(\text{left_child}) \cup R(\text{right_child}) \\ BF(\text{node}) &= BF(\text{left_child}) \cup BF(\text{right_child}) \\ h(\text{node}) &= H(h(\text{left_child}) \| h(\text{right_child}) \| BF(\text{node}) \| R(\text{node})) \end{aligned} \quad (3)$$

Security property

THEOREM 4.1. *The* IncBM-Tree *root node can authenticate any bloom filter in the tree structure.*

PROOF. The proof of security is based on the infeasibility of finding two different Bloom filters BF_1 and BF_2 such that $H(\ldots BF_1 \| \ldots) = H(\ldots BF_2 \| \ldots)$. If this is feasible, then it is easy to find two values, $v_1 = \ldots BF_1 \| \ldots$ and $v_2 = \ldots BF_2 \| \ldots$, such that $H(v_1) = H(v_2)$, which contradicts the fact that H is a collision resistant secure hash. □

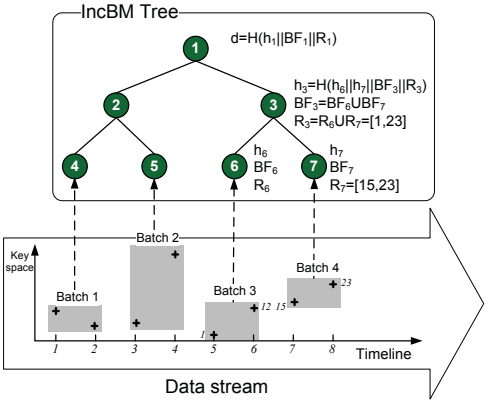


Figure 2: IncBM-Tree structure

4.3 Digest Materialization in the Framework

The digest structure is materialized in our authentication framework at two places: 1) The data owner locally maintains a partial IncBM-Tree as its local state for the signing purpose, 2) The cloud maintains a full copy of the IncBM-Tree for the query verification. In the following, we will follow the authentication data flow to describe the functioning of the materialized IncBM-Tree, that is, the owner-side digest construction and the cloud-side proof construction.

4.3.1 Digest Construction at Owner

The data owner digests a data update stream by maintaining a data batch in memory. Based on the buffered data batch, KoMT is constructed at two levels of time granularity, that is, every r minutes (for level-1 KoMT) and whenever the available memory space is consumed (for level-2 KoMT). At either time point, the owner would sort the buffered key-value data based on key and on the ordered list builds an MHT. The produced root hash for level-1 KoMT is reported to the cloud, while that for level 2 KoMT is added to IncBM-Tree as part of a new leaf node.

For IncBM-Tree construction, upon adding each new KoMT root, the owner maintains the Bloom filter of the current data batch. The maintenance of IncBM-Tree is incremental as illustrated in Algorithm 1; the newly created leaf node in IncBM-Tree would trigger a series of merging actions – If it finds that there are two nodes at the same level, those two nodes will be merged into one node and the new node will be promoted one level up. The process would construct a growing IncBM-Tree; it stops growing when it produces a single node and the error rate of Bloom filter of this node exceeds an error bound denoted by E_b. After a root of the IncBM-Tree is generated, it is chained with previously signed IncBM-Tree roots (i.e. chained in order of time). The signed IncBM-Tree root is then uploaded to the cloud. Note that a key-value record is digested and signed twice, one by level-1 KoMT (for real-time freshness authentication) and the other by the IncBM-Tree (for efficient freshness authentication to historical big data).

An example: We consider $b = 2$, which means KoMT is built every 2 records. At time point 2, a new KoMT is built based on data batch 1 of the first two records. The KoMT's root node is signed and uploaded to the cloud. The root node is also included in a new leaf node of the IncBM-Tree, that is, node 4. Then at time point 4, a similar KoMT is built on the next two records. After

it builds a new KoMT and uploads its root node, a new leaf of IncBM-Tree is created and then merged with node 4 to node 2. Similarly, at time point 6 and 8, two new KoMT's are built based on the new data batches and two new leaves are inserted into the IncBM-Tree. Particularly, at time point 6 it generates a new leaf node 6, and at time point 8 it generates another leaf which triggers the merging of node 6 and node 2 to node 1. At this point of time, only node 1 is maintained in the local state of the owner.

In general, during the construction of the IncBM-Tree, only the "frontier" part of the tree is maintained locally. Figure 3 illustrates a snapshot of the local state maintained by the data owner. The owner's local state consists of the full data copy of current batch for constructing two-level KoMT's, the frontier part of IncBM-Tree and a serials of previous signed roots of IncBM-Tree's.

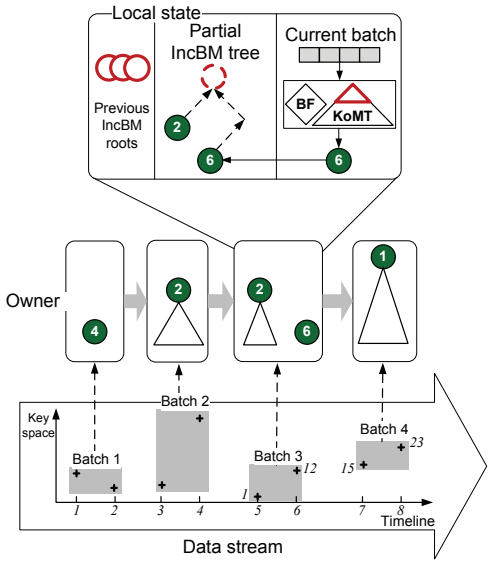


Figure 3: Incremental IncBM-Tree construction on key-value stream: the red shapes are digests that are signed and uploaded to the cloud.

Algorithm 1 IncBuild(Key-value batch s)

```
currentNode←keyOrderedMerkleDigest(s)
l ← 0
node← removeTreeNode(l)
loop node≠NULL
    currentNode← merge(currentNode, node)
    l ← l + 1
    node← removeTreeNode(l)
end loop
insertTreeNode(currentNode, l)
if E >= E_b then
    uploadIncBM-TreeToCloud()
end if
```

4.3.2 Proof Construction and Verification

To see how a proof is constructed and verified, we need to first take a look at the data and digest maintained by the cloud. At any point of time, the cloud maintains data of three kinds: 1) data without any digest; such data must be uploaded to the cloud within the last r minutes, given our real-time reporting mechanism, 2) data with a digest being level-1 KoMT, and 3) data with two digests, being level-1 KoMT and IncBM-Tree. While data 1) can not be authenticated due to lack of digest and signatures, data 2) can be authenticated by the traditional MHT's authentication path [17]. We assume the size of data 2) is small enough to fit into memory and freshness authentication can be efficiently performed. In the following, we hence focus on describing the mechanism for verifying freshness regarding data 3).

For data 3), the IncBM-Tree is used to construct proofs to verify a result of Get query. More specifically, the authenticated Bloom filters can be used to efficiently test the (non)-membership of a particular data key, as required for authenticating the data freshness. Following the example in Figure 2, to prove the freshness of key 98 at time point 8, it suffices to return only two nodes, that is, node 5 and node 3. Because Bloom filter BF_3 can test the non-membership of key 98 in time interval $[5, 8]$ (i.e. node 3), and digest h_3 can be used to verify authenticity of BF_3.

In reality, a Bloom filter can have error in its (non)-membership test. This implies that when a key k is not in a set, a Bloom filter may falsely claim the membership that k is in the set. In this case, our strategy is to go down IncBM-Tree by one level. For instance, when BF_3 can not verify the non-membership of key k, we use the two children's Bloom filters, BF_6 and BF_7, which collectively verify the non-membership. By using multiple lower-level Bloom filters, the chance of correctly verifying the non-membership becomes higher. In an extreme case where all the internal nodes fail to confirm the non-membership, it resorts to the KoMT beneath the IncBM-Tree to test the non-membership or membership. Traditionally, the KoMT can verify the non-membership by returning an authentication path that covers the queried key[17, 16]. By this means, it can always guarantee the error-free verification of data freshness.

4.3.3 Cost Estimation

We first define the error rate of non-membership test, $E(l)$, which is the probability that a non-membership test can fail using a BF at tree level l. We will deduce the mathematical formula of $E(l)$ later.

Recall that our digest construct enforces that $E(l = *) < E_b$. Based on this fact we can estimate the upper bounds of traversing the IncBM-Tree. Specifically, in the traversal, descending one level down only happens when a parent node fails to test non-membership, which occurs with probability $E(l)$. Since $E(l) < E_b$, we can have the fact that descending N levels down occurs with a probability smaller than E_b^N.

Theorem 4.2. *In an IncBM-Tree where the error rates of any Bloom filters at different levels are bounded by E_b, the extra cost X is expected to be the following.*

$$X = \frac{1 - E_b}{1 - 2E_b} + (2E_b)^l \big(E_b \log b - 2E_b \cdot \frac{1 - E_b}{1 - 2E_b}\big) \quad (4)$$

Here, cost X is defined to be the number of tree nodes for error-free verification.

Proof. Suppose the expected cost of a tree node at level l ($l = 0$ for leaf nodes) is X_l. Considering the traversal process of non-membership test, there are two cases: 1) The tree node's Bloom filter can correctly answer the non-membership query, and 2) The tree node can not. For the first case, it occurs with probability $1 - E_b$, since a Bloom filter's error rate is E_b. When the query cost is 1 for a single node, its contribution to the overall expected cost is $(1 - E_b) \cdot 1$. For the second case, it occurs with a Bloom filter's error rate E_b. And the query evaluation needs to descend into the tree node's direct children at level $l - 1$. The cost should be equal to the sum of expected costs at all the children nodes. Suppose each tree node has 2 children, the contribution of the second case to the

overall expected cost is $E_b \cdot 2X_{l-1}$. Overall, we have the following and drive a closed-form for the expected cost.

$$\begin{aligned} X_l &= 2E_b \cdot X_{l-1} + (1-E_b) \\ &= (2E_b)^2 \cdot X_{l-2} + (1-E_b)(1+2E_b) \\ &\ldots \\ &= (2E_b)^l \cdot X_0 + (1-E_b)[1+2E_b+(2E_b)^2+\cdots+(2E_b)^{l-1}] \\ &= \frac{1-E_b}{1-2E_b} + (2E_b)^l\big(X_0 - \frac{1-E_b}{1-2E_b}\big) \\ &= \frac{1-E_b}{1-2E_b} + (2E_b)^l\big(E_b \log b - 2E_b \cdot \frac{1-E_b}{1-2E_b}\big) \end{aligned}$$

The last step is due to that $X_0 = E_b \log b + (1-E_b)$. □

The implication of this theorem is that if we can bound the error rate of all BFs in the INCBM-TREE, we can also bound the cost of non-membership test largely based on the value of E_b. In practice, we set $E_b < 0.5$, then we can have $X_l \approx \frac{1-E_b}{1-2E_b}$ for a reasonably large l.

4.4 Effective System Configuration

In this part, we analyze our authentication framework and give suggestions on configurations to make the framework effective.

We start by deducing the formula of non-membership error rate $E(l)$. For a BF at level l in the INCBM-TREE, the node contains no more than $b \cdot 2^l$ key-value records. The BF's error rate (for membership test) is $E' = (1 - e^{-\frac{b \cdot 2^l}{m}})$ (here we use a single hash function)[23]. For key-value data, we consider a key domain of cardinality 2^{B_k}; that is, B_k is the necessary number of bits for storing a key. Thus, the error rate for non-membership test is the ratio of the likelihood that the error case occurs (i.e. $E' \cdot b2^l/(1-E')$) to the likelihood that a non-membership test happens (i.e. $E' \cdot b2^l/(1-E') + 2^{B_k} - b2^l$).

$$\begin{aligned} E(l) &= \frac{E' \cdot b2^l/(1-E')}{E' \cdot b2^l/(1-E') + 2^{B_k} - b2^l} \qquad (5) \\ &\approx (e^{\frac{b}{m}2^l} - 1) \cdot b \cdot 2^{l-B_k} \\ &= (e^{\frac{q}{1-q}\frac{l}{B}2^l} - 1) \cdot \frac{S}{B} q \cdot 2^{l-B_k} < \frac{1}{2} \end{aligned}$$

Here, we used approximation that $2^B_k \gg 1$ (e.g. $B_k = 8000$ for a one-KB key). In the last step, we used system parameter q which is the ratio of memory allocated to host KOMT (with the rest of memory to host INCBM-TREE); thus, we can apply $b = Sq/B$ and $m \cdot l = S(1-q)$ in the equation.

From the equation, we can see that 1) $E(l)$ is a monotonic increasing function to variable l; thus the root node would have the largest error rate among all tree nodes and it suffices to control the root in order to bound the error rates of all nodes in INCBM-TREE. 2) E is very sensitive to increasing the value of l. Such E value is a major factor that limits the INCBM-TREE from growing large and deep.

Setting the depth for INCBM-TREE (i.e. l) is an art. While a deep INCBM-TREE is good for increasing the chance to have a hit on the INCBM-TREE during the proof construction, it may hurt the verification efficiency and contribute to a large proof size when it misses. A rule of thumb is to set l to be a medium value; in our experiment setting, $l = 7$ works well in performance.

4.5 System Implementation

We implemented a proof-of-concept prototype for our authentication framework. We choose HBase [2] to be the substrate system, because 1) HBase is a representative key-value store, which is widely used for many cloud storage services, 2) HBase is optimized toward the write performance; it is designed by a log-structured merge tree [24], which lends itself to persisting the high-rate data stream,3) HBase attains strong consistency, a feature that is required for the freshness authentication by our framework.

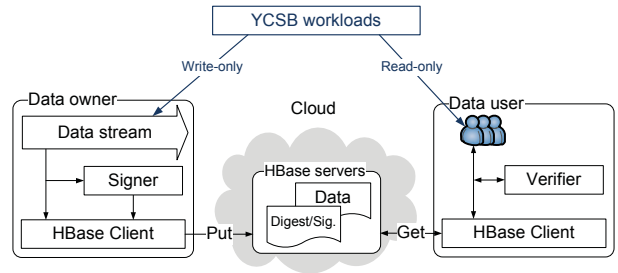


Figure 4: Prototype on HBase and experimental setup

The basic data flow in the HBase system is following. The data source installs an HBase client, and through the Put API submits the key-value data to the cloud. The data is then stored in a (base) table in the HBase server cluster. A data user installs the HBase client and through an HBase Get API requests to retrieve the outsourced data in the cloud.

To hook our authentication data flow, we made several system modifications and configurations. First, we attached the signer component to the Put path at the owner side; this is done by instrumenting the HBase client library; we emit every key-value data in the Put path to the signer component, which then builds the digest as was previously described. The digest is submitted to the cloud (more specifically to the "digest table" in the cloud as described below) by the HBase Put call. Note that we only submit the signed root node to the cloud, and leave everything else (e.g. the digest structure in the INCBM-TREE) local; such digests can be recomputed from the outsourced raw data in the cloud. Second, the HBase server in the cloud maintains two tables, one for the base data and the other for the meta-data including the digests and signatures. The digest table is sharded based on the generation time. Third, for proof construction, a baseline implementation is to coordinate the proof construction by data users; after receiving a Get result the user could submit a separate Get request to the digest table. For performance concern and saving the extra communications to the cloud, we implemented an alternate design that automatically triggers the proof construction upon the base table receiving a Get request. This is implemented by using HBase's CoProcessor interface [3] which allows a programmer to attach external actions to HBase's internal events (e.g. receiving a Get call). The proof is constructed by consulting the digest table which stores the full history of authenticated digest structures. After being constructed, the proof is encoded in the Get result to the user. Fourth, after receiving the Get result, the data user parses it and extracts the encoded proof, which is then sent to the verifier component to check the authenticity of the query result based on the owner's public key. The verifier then notifies the end user of the final authentication result. We illustrate our system implementation in Figure 4.

The internal of the authentication framework is implemented based on the cryptographic library in Java (i.e. `javax.crypto.*`). In particular, we used RSA for digital signature.

5. EVALUATION

We first describe our simulation result to study the performance

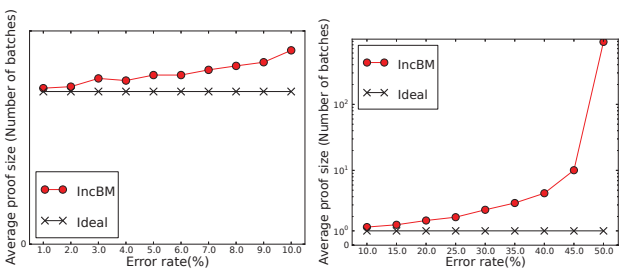
of the INCBM-TREE, and then report the experimental result based real system setup.

5.1 Simulation Study

We first conducted simulation-based performance study to justify the hierarchical design of the INCBM-TREE.

5.1.1 Proof Construction Cost

The extra cost for proof construction/verification plays a key role in INCBM-TREE's overall efficiency. We first studied the verification cost based on Equation 4. Particularly, we varied the error rate of Bloom filters in INCBM-TREE (by changing their sizes). Under each setting, we repeated the experiments for 100 times and plotted the average proof size in Figure 5. With small error rates (e.g., $< 10\%$), the proof size was very small, e.g., slightly above 1. With large error rates (e.g., $\geq 10\%$), the proof size exponentially increased as shown in Figure 5b. The result was consistent with our cost analysis in the sense that any Bloom filter should not have error rate too large (e.g. 10%).



(a) Small error rate (b) Large error rate

Figure 5: Effect of the error rate of a Bloom filter on the constructed proof size (Y axis in a log scale)

5.1.2 Non-membership Test Cost

We then compared INCBM-TREE with two alternate designs, KOMT-Only which only maintains KOMT, and KOMT-None which only maintains INCBM-TREE without KOMT. For simulation, we considered all approaches being memory-resident. Instead of the proof size, we used a metric Z in this experiment which measures the per-record cost during a non-membership test. This metric is more suitable for comparison since it is independent to the data and query distribution. For INCBM-TREE, we used the following formula to calculate Z in our simulator. Here, b_1 denotes the batch size used in the INCBM-TREE approach.

$$\begin{aligned} Z_1 &= X/(b_1 \cdot 2^l) \qquad (6) \\ &= \frac{1}{b_1 \cdot 2^l}[\frac{1-E_b}{1-2E_b} + (2E_b)^l(E_b \log b_1 - 2E_b\frac{1-E_b}{1-2E_b})] \end{aligned}$$

For KOMT-Only, metric Z is,

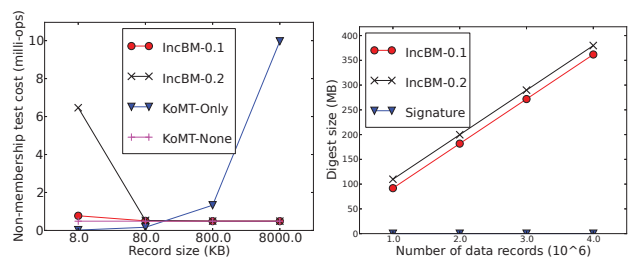
$$Z_2 = \frac{\log b_2}{b_2} = \frac{B}{S} \cdot \log \frac{S}{B} \qquad (7)$$

Here, we used two facts: 1) A non-membership test in an MHT of b_2 leaf nodes requires an authentication path from a leaf to the root, which costs $\log b_2$ operations, 2) The KOMT-Only design uses all memory space to store the data batch of KOMT, yielding $b_2 = \frac{S}{B}$.

For the KOMT-None approach, it can be thought as a special case of INCBM-TREE when the KOMT is built on individual data records. By plugging $b_3 = 1$ in Equation 6, we have the following.

$$Z_3 = \frac{1}{2^l}\frac{1-E_b}{1-2E_b}[1-(2E_b)^{l+1}] \qquad (8)$$

In our simulation, we used the memory size 1 GB = $10^9 \times 8$ bits, the size of key-value pairs ranging from 1 KB, 10 KB, 10^2 KB to 10^3 KB. We set the INCBM-TREE depth $l = 7$ for both INCBM-TREE and KOMT-None approach; such value is the largest one that obeys our constraints as in Equation 5. For the INCBM-TREE, we consider the different memory ratios, $q = 0.1$ and $q = 0.2$. We report the value of metric Z in the unit of Milli-ops (i.e. 10^{-3} operations) per record. We show our simulation result in Figure 6a. In the result, it is easy to see that approaches using BF, including INCBM-TREE and KOMT-None, incur fewer costs for non-membership tests than KOMT-Only. When the key-value records grow large, the cost of KOMT increases quickly. This result confirms the approximate-linear relationship between metric Z_2 and record size B as in Equation 7 (note that $S \gg B$). For INCBM-TREE, its cost stays small because of the compression effect of BF used in INCBM-TREE.



(a) Non-membership test cost (b) Digest size

Figure 6: INCBM-TREE performance study

5.1.3 Digest Size

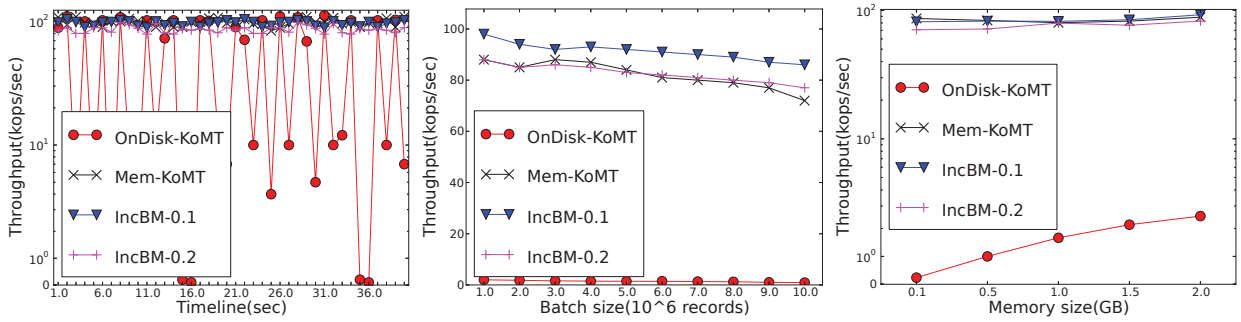
In order to evaluate the bandwidth cost of outsourcing, we measured the size of constructed digests. A digest size matters a lot to a cloud where a digest has to be fully stored there for proof construction. We measured the digest size for INCBM-TREE while varying the number of data records. We also included the size of digital signatures as comparison points as shown in Figure 6b. A digest size, measured by the number of hash values, increased linearly to the data size, measured by the number of records in the stream. A digest was significantly larger than a signature, which supported our design choice where we did not transmit a digest to a cloud.

5.2 Real Experiment Platform Setup

Based on our prototype system, we used YCSB to drive read-/write workloads, as shown in Figure 4. YCSB[6] is an industrial strength benchmarking tool to simulate various key-value workloads in the cloud. In experiments, we launched two YCSB instances, one to generate write-only workload to the owner, and the other to generate read-only workload to the users.

We deployed our experiment system in Emulab [30]. The data owner and users were set up on two separate machines while the HBase server cluster run on 10 machines for the cloud storage. In the HBase cluster, we installed HBase on top of HDFS. We used

[6]https://github.com/brianfrankcooper/YCSB/wiki



(a) Time series (b) Varying batch sizes (c) Varying memory sizes

Figure 7: Stream write performance

one machine for the master node (i.e. HBase HMaster and HDFS namenode); the rest 9 machines were used as the slaves. We used default software configurations for both HBase and HDFS. All machines were homogeneous with 3 GHz CPU and 2 GB RAM.

5.3 Write Performance

Based on the real platform, we first evaluated the write performance of INCBM-TREE. We describe our specific experiment setup and then report the evaluation result.

5.3.1 Experiment Setup

We measured the performance of ingesting a data stream. In our setup, 300 million key-value pairs were produced by YCSB; we used a write-only configuration to generate the workloads under a Zipf distribution. The data were driven into the owner-side signer; in this process, we measured the sustained throughput and latency. We saturated the system by setting the targeted throughput to be higher than estimated sustainable throughput.

For comparison, we considered the KOMT-Only design, as it is used by prior work [17, 25]. Particularly, we considered two realizations of KOMT-Only: One was memory resident, and the other was on disk. The memory-only KOMT occupied the whole memory space; that is, the batch size is exactly equal to the memory space. By contrast, the on-disk KOMT may spill data to and retrieve data from disk, and we set its batch size to be three times of the memory space. For INCBM-TREE, it always resided in memory and we tested with memory ratio $q = 0.1$ and $q = 0.2$. Here, one might argue that it is unfair to compare the memory-resident INCBM-TREE with on-disk KOMT. It is a fair setting as we will see that both approaches achieve similar performance in verification – The fact that the on-disk KOMT needs more memory footprint and needs to spill data to disk would show the superiority of the INCBM-TREE design. Likewise, we consider both the in-memory KOMT and in-memory INCBM-TREE; by this way, their write performances are similar (i.e. without disk IO upon signing) and we can then fairly compare them in terms of the verification costs.

5.3.2 Time-series Results

The time-series result along the data ingesting process was reported in Figure 7a. We did not include the initial data loading stage to exclude unstable system factors, e.g., cold cache. While the throughput of INCBM-TREE remained stable and high, the throughput of KOMT fluctuated along the time-line. At the valley points, KOMT was performing heavy disk I/Os to flush overflowing data and to load data from disk to memory for signing. Due to the reason, the average throughput of KOMT was lower than that of INCBM-TREE.

5.3.3 Average Throughput

We repeated the above primitive experiments multiple times under different settings and reported their average. We first varied batch sizes under a fixed memory size 0.5GB so that tested batch sizes were always bigger than memory sizes. Figure 7b reports the throughput. INCBM-TREE achieved an order of magnitude higher average throughput than the on-disk KOMT. As the batch size increased, the throughput of KOMT decreased because more disk accesses were required for signing. The throughput of INCBM-TREE remained almost the same across various batch sizes because of the incremental digest construction. INCBM-TREE achieved much higher throughput than KOMT due to the pure memory operations without disk I/O. We then varied the memory size under the fixed batch size of 4GB. As described in Figure 7c, the throughput of KOMT increased with larger memory size mainly because of fewer disk I/Os. The throughput of INCBM-TREE remained stable with different memory sizes.

5.4 Query Performance

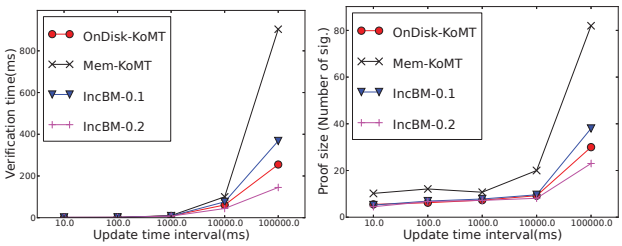
5.4.1 Experiment Setup

We further conducted experiments to measure the query verification cost. Before the experiment, we set up the cloud system in advance; we deployed the HBase cluster and populated it using a pre-materialized key-value dataset (which was generated earlier using YCSB). The cloud maintained full copies of different authentication structures (e.g. KOMT and INCBM-TREE), which followed the setting described previously. To conduct the experiments, we drove a read-only YCSB workload (more specifically, the workload-C in YCSB) into the system through data users (as in Figure 4). Recall that the cloud constructs the proof for the query result, and the user verifies the result based on the proof. Here, since we were not interested in the performance of the cloud part (which is typically not the system bottleneck in an outsourcing scenario), we were mainly concerned with the client performance, e.g. the user-side verification cost.

5.4.2 Verification Performance

In our query model (i.e. Get (k, t)), different keys k's are updated with different time intervals – While some keys are very frequently updated, other keys are not. Such update frequency plays a key role in determining the verification cost, and we used it as the

parameter in our experiment. As for the metric, we measured the proof size and actual verification time. The proof size is captured in the unit of batch numbers. We report our experiment result in Figure 8a for verification time and Figure 8b for the proof size. It is clear to see that both metrics increase linearly with the update time interval, which is expected. Because the older the last update is (as is usual for records updated in larger intervals) the longer history it has to dig into for the non-membership test. For different approaches, INCBM-TREE outperformed the rest, because it uses BF to index the non-membership information across KOMT batches and improves the test efficiency. In particular, on-disk KOMT achieves similar proof size with INCBM-TREE. However, this efficiency for on-disk KOMT comes at the expenses of much higher write overhead and lower write throughput as revealed in Figure 7b.



(a) Time (b) Proof size

Figure 8: Verification performance

6. RELATED WORK

6.1 Data Outsourcing and Security Issues

In the age of cloud computing, data outsourcing has become a popular practice to save the operational overhead. As most cloud services are provided in the public domain and by a third-party operating company, security issues become an inevitable concern in practice. Hacigumus et al. introduced general security issues in an outsourced database scenario [14]. Most security concerns can be attributed to data confidentiality and/or authenticity. While there is a large body of research work for information confidentiality in the public cloud (e.g. querying encrypted data [26]), the concern of our work is on the data authenticity aspect.

Work for authenticated data outsourcing can be classified from the system perspective. There are various outsourced data systems, ranging from pure computations (e.g. MapReduce [27, 8]), to pure storage, and to a hybrid (e.g. a database management system). In particular, the outsourced dynamic databases [16, 25] consider outsourcing data that is subject to updates from the data owners and serves SQL queries to the data users. The outsourced data streams [17, 10, 28] consider a continuous data stream as input and serves stream-specific queries, such as aggregations, continuous queries, etc. Comparing to the databases, data stream system is less concerned about storing data and enabling access to historical big data; instead, it is more interested in processing and serving relatively small data in memory (e.g. recent data within a time window). In the system community, outsourced file systems and key-value stores are considered. While file systems deal with sequential data reads/writes in addition to the random ones, key-value stores are more primitive, typically dealing the very basic Put/Get API. The key-value stores are scalable and highly available, which lends themselves to providing cloud services; there is a trend to superimpose various high-level functions (e.g. databases [5]) on top of key-value stores. Authentication is also studied in the context of P2P networks [15]. In their systems, multiple peers are located in different domains, which makes them mutually untrusted. This model is different from a key-value store, where all nodes are located in a single domain, namely the cloud domain.

In different systems, authenticity has different meanings. For a single data unit (e.g. a data blob), authenticity or data integrity requires that the content can not be changed by the cloud without the notice of end users. For a faulty cloud storage, PDP or provable data possession [6] is authenticity on the data-possession information. When a range query is considered, completeness [17, 16] is an important property that assures that no valid data in the queried range is missed in the result. Our work considers authentication of version freshness in outsourced key-value stores; such freshness is specific to the multi-version Put/Get API exposed by many key-value stores.

6.2 Freshness Authentication

To authenticate data freshness, existing digest structures mainly rely on two approaches; updating an MHT [29, 13] or offline auditing authenticated operation histories (by hash chains) [26, 12].

In the updating-MHT approach, the owner stores a local state for the remote MHT in the cloud. The local state could be the root hash of the MHT [29, 13] or the full copy [16, 20, 21]. Given data updates, the owner needs to update and sign the local state before sending the authenticated updates to the cloud. In this approach, authenticating the updates is inefficient as it either needs to maintain huge on-disk local data as in the full-copy variant or needs the owner to read back authenticated information from the cloud for updates as in the root-hash variant.

The other approach for freshness authentication is to audit operation histories from both sides of the users and cloud; at the end of each epoch, the trusted owner collects the operation histories from the cloud (which is assumed to be willing to collaborate) and data users, and then by comparing the histories, freshness can be authenticated or a violation is detected. By using hash chain and user-supplied randomness, the history information from the cloud can be authenticated so that it can effectively prevent the cloud from forging the history. The audit-based freshness authentication assumes logging responsibility on the user side and can only detect the violation in an offline and delayed manner (i.e. at the end of each epoch).

6.2.1 Systems

CloudProof [26] addresses the authentication of Put/Get in an untrusted key-value store service (e.g. Amazon S3). In the system, the trusted owner is offline in the sense that it is not present in any active Put/Get path from the data users. This design relieves the owner from being $24/7$ online. Data freshness requires a Get to return the latest available data written by Put. CloudProof audits the authenticated history to provide the freshness guarantee.

SPORC [12] supports multi-user collaborative applications on a P2P alike system with a thin server. The sole responsibility of the server is to coordinate and order the operations on the object shared by multiple users. In the system, users or logged-in clients trust each other but they do not trust the server. SPORC authenticates fork consistency (or variants) on multi-version data in a multi-user context. The proposed technique for that is to authenticate the committed write history of a user using hash chain and to verify the new writes by directly contacting the original client. In a similar setting, Depot [19] verifies the freshness of data obtained from the cloud by periodically communicating the trusted peer clients.

In the database community, various data management systems consider dynamic dataset updated by a stream of data writes. To guarantee data freshness under this dynamic setting, existing

work [25, 16] relies on the traditional mechanisms for document certificate validation [22], such as publishing revoked signatures and signing with expiration time (and resign upon expiration).

To authenticate freshness in real-time, prior work [31] assumes a trusted computing base in the cloud server. Iris [29] considers the outsourcing of an enterprise file system to the cloud; the reads and writes are issued from the users of the same enterprise domain and the trusted enterprise is present in every read/write path. In the presence of updates to file block, Iris addresses the freshness problem – it is the latest version of the file block in the cloud that should be returned. Iris updates the MHT for freshness; it is guaranteed by maintaining the latest version numbers of each file block in a Merkle tree and locally maintaining its root hash. Persistent authenticated dictionaries or PADs [11, 4] are a performance-optimized data structure to authenticate multi-version data with key-completeness in the presence of historical access. As PADs do not particularly address the freshness authentication problem, they are complementary with INCBM-TREE.

7. CONCLUSION

In this paper, we highlighted and articulated the problem of providing data freshness assurance for outsourced multi-version key-value stores. We proposed INCBM-TREE, a novel authentication structure which offers a set of desirable properties in intensive stream authentication: 1) lightweight for both data owners and end users, 2) optimized for intensive data update streams, and 3) immediate authentication of data freshness in the presence of real-time and historical data accesses. Through extensive benchmark evaluation, we demonstrated that INCBM-TREE provided throughput improvement (in an order of magnitude) for data stream authentication than existing work. The superior performance makes our approach applicable particularly to data owners and clients with weak computational capabilities, which is typical for outsourcing scenarios.

Acknowledgment

Yuzhe Tang and Ling Liu were partially supported by the National Science Foundation under Grants IIS-0905493, CNS-1115375, IIP-1230740, and a grant from Intel ISTC on Cloud Computing. This research was partially sponsored by the U.S. Army Research Laboratory and the U.K. Ministry of Defense and was accomplished under Agreement Number W911NF-06-3-0001. The views and conclusions contained in this document are those of the author(s) and should not be interpreted as representing the ofĭňĄcial policies, either expressed or implied, of the U.S. Army Research Laboratory, the U.S. Government, the U.K. Ministry of Defense or the U.K. Government. The U.S. and U.K. Governments are authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation hereon.

8. REFERENCES

[1] http://cassandra.apache.org/.
[2] http://hbase.apache.org/.
[3] https://blogs.apache.org/hbase/entry/coprocessor_introduction.
[4] A. Anagnostopoulos, M. T. Goodrich, and R. Tamassia. Persistent authenticated dictionaries and their applications. In *Information Security ISC 2001*, pages 379–393, 2001.
[5] M. Armbrust, K. Curtis, T. Kraska, A. Fox, M. J. Franklin, and D. A. Patterson. Piql: Success-tolerant query processing in the cloud. *PVLDB*, 5(3):181–192, 2011.
[6] G. Ateniese, R. C. Burns, R. Curtmola, J. Herring, L. Kissner, Z. N. J. Peterson, and D. X. Song. Provable data possession at untrusted stores. In *ACM CCS*, pages 598–609, 2007.
[7] B. H. Bloom. Space/Time trade-offs in hash coding with allowable errors. *Communications of the Association for Computing Machinery*, 13(7), 1970.
[8] B. Braun, A. J. Feldman, Z. Ren, S. T. V. Setty, A. J. Blumberg, and M. Walfish. Verifying computations with state. In *SOSP*, pages 341–357, 2013.
[9] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. Gruber. Bigtable: A distributed storage system for structured data (awarded best paper!). In *OSDI*, pages 205–218, 2006.
[10] G. Cormode, A. Deligiannakis, M. Garofalakis, and S. Papadopoulos. Lightweight authentication of linear algebraic queries on data streams. In *SIGMOD*, 2013.
[11] S. A. Crosby and D. S. Wallach. Super-efficient aggregating history-independent persistent authenticated dictionaries. In *ESORICS 2009*, pages 671–688, 2009.
[12] A. J. Feldman, W. P. Zeller, M. J. Freedman, and E. W. Felten. Sporc: Group collaboration using untrusted cloud resources. In *OSDI*, pages 337–350, 2010.
[13] M. T. Goodrich, C. Papamanthou, R. Tamassia, and N. Triandopoulos. Athos: Efficient authentication of outsourced file systems. In *ISC*, pages 80–96, 2008.
[14] H. Hacigümüs, S. Mehrotra, and B. R. Iyer. Providing database as a service. In *ICDE*, pages 29–38, 2002.
[15] A. Kapadia and N. Triandopoulos. Halo: High-assurance locate for distributed hash tables. In *NDSS*, 2008.
[16] F. Li, M. Hadjieleftheriou, G. Kollios, and L. Reyzin. Dynamic authenticated index structures for outsourced databases. In *SIGMOD Conference*, pages 121–132, 2006.
[17] F. Li, K. Yi, M. Hadjieleftheriou, and G. Kollios. Proof-infused streams: Enabling authentication of sliding window queries on streams. In *VLDB*, pages 147–158, 2007.
[18] J. Li, M. N. Krohn, D. Mazières, and D. Shasha. Secure untrusted data repository (sundr). In *OSDI*, pages 121–136, 2004.
[19] P. Mahajan, S. T. V. Setty, S. Lee, A. Clement, L. Alvisi, M. Dahlin, and M. Walfish. Depot: Cloud storage with minimal trust. In *OSDI*, pages 307–322, 2010.
[20] C. Martel, G. Nuckolls, P. Devanbu, M. Gertz, A. Kwong, and S. G. Stubblebine. A general model for authenticated data structures. *Algorithmica*, 39(1):21–41, Jan. 2004.
[21] R. C. Merkle. A certified digital signature. In *Proceedings on Advances in Cryptology*, CRYPTO '89, 1989.
[22] S. Micali. Efficient certificate revocation. 1996.
[23] M. Mitzenmacher and E. Upfal. *Probability and computing - randomized algorithms and probabilistic analysis*. Cambridge University Press, 2005.
[24] P. E. O'Neil, E. Cheng, D. Gawlick, and E. J. O'Neil. The log-structured merge-tree (lsm-tree). *Acta Inf.*, 33(4):351–385, 1996.
[25] S. Papadopoulos, Y. Yang, and D. Papadias. Cads: Continuous authentication on data streams. In *VLDB*, pages 135–146, 2007.
[26] R. A. Popa, C. M. S. Redfield, N. Zeldovich, and H. Balakrishnan. Cryptdb: protecting confidentiality with encrypted query processing. In *SOSP*, pages 85–100, 2011.
[27] I. Roy, S. T. V. Setty, A. Kilzer, V. Shmatikov, and E. Witchel. Airavat: Security and privacy for mapreduce. NSDI'10, pages 20–20, Berkeley, CA, USA, 2010. USENIX Association.
[28] D. Schröder and H. Schröder. Verifiable data streaming. In *ACM CCS*, pages 953–964, 2012.
[29] E. Stefanov, M. van Dijk, A. Juels, and A. Oprea. Iris: a scalable cloud file system with efficient integrity checks. In *ACSAC*, pages 229–238, 2012.
[30] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar. An integrated experimental environment for distributed systems and networks. In *OSDI*, 2002.
[31] H.-J. Yang, V. Costan, N. Zeldovich, and S. Devadas. Authenticated storage using small trusted hardware. In *CCSW*, pages 35–46, 2013.

SEER: Practical Memory Virus Scanning as a Service *

Jason Gionta
NC State University
jjgionta@ncsu.edu

Ahmed Azab
Samsung Electronics Co., Ltd.
ahmedmoneeb@gmail.com

William Enck
NC State University
whenck@ncsu.edu

Peng Ning
NC State University
pningncsu.edu

Xiaolan Zhang
Google Inc.
czhang.us@gmail.com

ABSTRACT

Virus Scanning-as-a-Service (VSaaS) has emerged as a popular security solution for virtual cloud environments. However, existing approaches fail to scan guest memory, which can contain an emerging class of Memory-only Malware. While several host-based memory scanners are available, they are computationally less practical for cloud environments. This paper proposes SEER as an architecture for enabling Memory VSaaS for virtualized environments. SEER leverages cloud resources and technologies to consolidate and aggregate virus scanning activities to efficiently detect malware residing in memory. Specifically, SEER combines fast memory snapshotting and computation deduplication to provide practical and efficient off-host memory virus scanning. We evaluate SEER and demonstrate up to an 87% reduction in data size that must be scanned and up to 72% savings in overall scan time, compared to naively applying file-based scanning approaches. Furthermore, SEER provides a 50% reduction in scan time when using a warm cache. In doing so, SEER provides a practical solution for cloud vendors to transparently and periodically scan virtual machine memory for malware.

1. INTRODUCTION

Virus Scanning-as-a-Service (VSaaS) has become a popular topic for research and industry [4, 9, 17, 20, 25]. However, existing solutions only scan secondary storage. As malware continues to mature, file-only virus scanning is insufficient. We have already seen several instances of "Memory-Only Malware" that evades detection by many popular virus scanners [19, 14, 15]. In fact, only 30% of the most popular host-based virus scanners support memory virus scanning [16], and the computational overhead of their signature matching techniques makes memory scanning an infrequent event.

Memory-based VSaaS would provide benefit to both cloud clients and providers as Virtual Machines (VMs), like traditional hardware environments, require proper security measures to minimize threats and detect intrusions. However, designing a practical VM memory scanning service is non-trivial as (1) virus scanning is traditionally a CPU intensive task that can negatively impacts guest VM operations, (2) memory that changes during analysis can lead to false positives [2], (3) scanning multiple VMs concurrently can impact VM and host operations (i.e., "anti-virus storms") [23], and (4) file scanning optimizations are less effective for memory.

In this paper, we propose SEER as an architecture alternative to in-host memory scanning that enables efficient, transparent Memory VSaaS for virtualized cloud infrastructures. SEER "peers" into running VMs to efficiently identify malware in resident memory. SEER is built on DACSA, a decoupled architecture for cloud security analysis, to minimize guest and host performance [7]. DACSA enables fast memory snapshots and memory extraction. SEER processes snapshot data off-host to identify similar memory and duplicate scanning computation thus leading to a reduction in scan time. In doing so, SEER allows cloud providers to scan guest memory much more frequently and efficiently than naively applying existing file based scanning approaches.

Computation deduplication in SEER is achieved using a new algorithm that we call Prefix Based Scanning. Prefix Based Scanning is optimized by identifying partial data similarities across all memory segments to scan and normalizing similar data. For example, many Windows processes load *ieframe.dll* each containing small differences based on load address and runtime state. As a result, Prefix Based Scanning reduces the amount of computation to scan by only scanning identical prefixes once. Furthermore, our Prefix Based Scanning algorithm also identifies memory pages not present and virtually "fills in" similar memory from other processes per VM to increase virus scanner accuracy.

As a proof of concept to analyze the efficiency of SEER, we adapt the popular open source virus scanner ClamAV [10] to enable Memory VSaaS with Prefix Based Scanning. Our results show that SEER is practical and can be deployed by cloud vendors to find both existing file based malware and stealthy Memory-Only Malware in memory.

We make the following contributions:

- We propose a new architecture for efficient Memory VSaaS. SEER addresses a major gap present in existing Virus Scanning-as-a-Service solutions. We describe and overcome the challenges of memory scanning that prevent naive application of traditional file based scan-

*This work is supported by U.S. National Science Foundation (NSF) under grant CNS-1330553.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.

ACSAC '14, December 08 - 12 2014, New Orleans, LA, USA
Copyright 2014 ACM 978-1-4503-3005-3/14/12 $15.00
http://dx.doi.org/10.1145/2664243.2664271.

ning techniques.

- We propose Prefix Based Scanning for efficiently scanning dynamic memory. Our proposed approach works to consolidate data thus reducing the amount of computation required to scan memory. The approach can be adopted by existing host based memory virus scanners to reduce scanning computation.
- We provide an experimental evaluation of SEER and show our design reduces the data size to be scanned up to 87% and the time to scan the data by up to 72%, compared to naive scanning approaches. We also demonstrate that SEER can correctly identify memory malware in infected machines.

The remainder of this paper proceeds as follows. Section 2 provides background on Memory-Only Malware and the challenges faced by memory scanning. Section 3 contains our design, implementation, assumptions and threat model. Section 4 evaluates SEER. Section 5 discusses deployment, security and privacy concerns, and limitations. Section 6 discusses related work. Section 7 concludes.

2. MOTIVATION AND BACKGROUND

2.1 Memory-Only Malware

Memory-Only Malware (**MoM**) identifies a class of stealthy malicious software that does not modify files on a target system. As such, MoM bypasses file-based detection techniques and the majority of popular virus scanners [16]. MoM trades persistence for stealth. While MoM will not survive system reboots, attackers are using new ways to gain persistence through "Watering Hole Attacks" [18].

MoM is not a new concept. The infamous Code Red Worm propagates only through memory [19]. More recently, security analysts have identified a Hydraq/McRAT variant that allows remote access to infected machines [15]. Finally, the popular MetaSploit framework provides an "out-of-the-box" MoM called Meterpreter [14].

Malware such as Zeus and Conficker also benefit from the lack of memory scanning. They only need to bypass virus scanning software once to gain a foothold. Once initial detection is bypassed, the malware uses code injection to infect benign active processes [12]. Injected code allows malware to stay resident in the running machine even when physical files are later found and removed. As a result, malware analysts recommend using memory dumps to discover and identify malware [12].

2.2 Scanning Memory

Memory scanning consists of analyzing virtual memory to find malware based on virus definitions. Previous work [9] has applied file-based scanners to physical memory. However, scanning physical memory will lead to incorrect results because virus definitions require correct context to match definitions. As a result, a memory based virus scanner must scan virtual memory to ensure correct context. For modern operating systems, each process has its own virtual memory.

The virtual memory of a process (p) is a set of *memory segments* S_p. Each memory segment $s \in S_p$ represents an individual memory allocation managed by the operating system (OS) with size s_{size} bytes. Memory segments can contain stacks, heaps, or mapped files. Then $p_{size} = \Sigma s_{size}$, $\forall s \in S_p$ represents the size of memory allocated for process p. Malware can potentially reside in any process memory segment, therefore **all** processes on the OS, represented as the set P, must be scanned. That is, $P_{size} = \Sigma p_{size}$, $\forall p \in P$ bytes must be scanned.

A naive approach would be to scan all P_{size} bytes of memory. However, this faces the following challenges:

C1. Scanning is computationally intensive requiring significant CPU resources to match signatures. Scanning often disrupts critical operations during analysis. A large P_{size} can result in minutes of scanning and thus is not ideal for production environments.

C2. Memory modified during analysis can impact the correctness of scanner results, leading to false positives and negatives [2]. This occurs because scanners must copy, examine, and analyze memory that is constantly changing. As a result, false assumptions are made about the consistency of the data being analyzed.

C3. Multiple VMs scanned at the same time on a single host leads to "Anti-Virus Storms", which can affect non-scanning guests and host operations [23].

C4. Hashing is widely used by file-based virus scanners as an optimization to limit the amount of data to scan. Memory is often unique which limits the effectiveness of hashing memory segments to identify previously scanned data. We find that previous hashing techniques used for file-based scanning [20, 17] only reduces the memory segments to scan by ≈14% for a single VM.

3. SEER

Next we give an overview of the SEER architecture. We discuss the disparate components for enabling SEER and the workflow for scanning VMs. We then detail specific design decisions for Prefix Match Scanning.

3.1 Overview

The SEER architecture enables practical, VM transparent, efficient memory scanning for virtualized cloud infrastructures. SEER overcomes challenges (**C1-C4**) by shifting memory scanning to a third party in which (1) guest VM memory is extracted and only unique data is shipped off-host, (2) a scan scheduler pre-processes data to identify similarities in data to scan, and (3) a modified scanner uses pre-processed meta-data to identify viruses in an isolated environment. The VM and host only incurs a small cost for extracting and shipping data (**C1**,**C3**) and prevents malware or exploits that targets the scanning process [17].

Figure 1 illustrates the main components and flow of SEER. The cloud provider will request/schedule (1) VMs for scanning. The scan scheduler initiates a scan (2) by requesting the memory extractor to extract memory and then ship memory meta-data and unique page data for pre-processing. The scheduler (3) stores the data and builds a scan configuration based on data extracted. The scan scheduler then triggers a new scan (4) using the generated scan configuration and (5) results are relayed to the cloud provider. For our design discussion, we consider QEMU/KVM as the host and Windows as the guest VM operating system.

On the virtualization hosts, we leverage DACSA to support fast, non-intrusive VM memory snapshots and memory

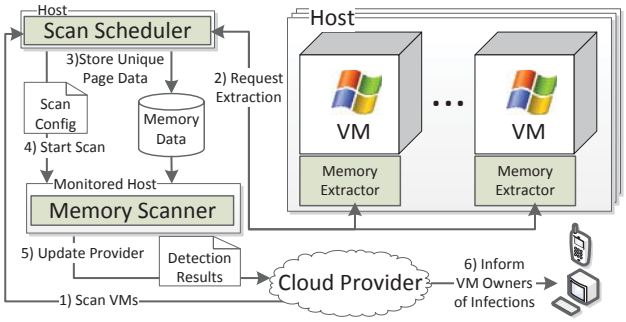


Figure 1: SEER Overview

extraction. DACSA creates a logical static copy of memory in milliseconds without copying data byte-by-byte [7]. This prevents consistency issues which can arise from analyzing memory that continues to change (**C2**). Memory segments to be scanned are extracted from the logical copy using pre-existing knowledge of the VM operating system similar to forensic techniques employed by Volatility [24]. Segment data meta-data is sent to the scheduler and instructs the extractor to send/store only unique page data. This prevents shipping large amounts of data over the network.

After the necessary page data is shipped, the scan scheduler beings pre-processing by reconstructing stored page data into memory segments for each VM. Each segment can be naively scanned with traditional virus scanners however the total size to scan (P_{size}) can be very large.

To address scanning large amounts of data, file based virus scanners use hashing to identify previously scanned data. Files hashed to known values do not need to be re-scanned. This technique is most beneficial on static data. Unfortunately, memory segments are often unique due to its dynamic nature making this simple optimization less effective.

We observe that while many memory segments are unique, many segments are similar to some other segments within a VM. For example, the Windows library *ieframe.dll* can be found mapped across many processes in a running system (e.g., Internet and Windows Explorer). Memory segments for *ieframe.dll* will share identical PE headers and code sections but will differ in their data sections. As a result, a naive scanner will simply identify both segments as unique.

Using simple hashing techniques to eliminate scanning duplicate data is not effective for memory. Furthermore, false positives can arise from scanning non-present or zeroed data due to coarse grained virus definitions. SEER employs four optimizations to address the above issues:

1. Only globally unique page data is sent off scanned VM hosts. This obviates shipping all snapshot memory and reduces the processing time to extract memory hence reducing impact to VMs, hosts, and network.
2. SEER matches identical prefixes across all memory segments. Matched prefix data determines where duplicate scanner computation occurs. SEER uses matched prefix data to scan identical prefixes only once. For example, two libraries extracted from memory will have duplicate header data but contain different runtime data. SEER will match the header as a prefix and only scan the header data once instead of multiple times.
3. Memory segments are normalized intra-VM to increase similarities between segments. Normalizing memory segments increases the lengths of matched prefixes. This results in less data being scanned.
4. The SEER scanner is provided meta-data and thus can avoid unnecessary computations by not processing non-present data.

The above optimizations allow SEER to efficiently scan unique data reducing the computation required to analyze the full set of memory segments (**C4**).

SEER leverages existing virus scanners for identifying and matching known malware signatures in memory. Thus, SEER's effectiveness at identifying malware is reliant on the adopted virus scanner's signature database and signature matching techniques. SEER does not look to improve matching of existing scanners but to provide a novel architecture for extending existing techniques to efficiently scan memory.

Assumptions and Threat Model: We assume that an attacker has full control over the guest VM. However, attackers wish to remain stealthy and avoid detection thus avoiding behaviors which raise suspicion such as attacks on availability or resource exhaustion.

3.2 VM Host Memory Segment Extraction

Next we discuss the process of memory extraction and segment data shipping off VM host machines. Memory extraction is built on DACSA to limit the impact to host and guest environments. We add discussion on shipping memory segments not previously discussed in DACSA [7].

3.2.1 DACSA: Fast Snapshots and Extraction

DACSA is a decoupled architecture for virtual machine memory analysis. DACSA introduced a reliable fast memory snapshot technique using Copy-On-Write to create a *logical copy* of Virtual Machine memory allowing memory to become a data source for analysis. Memory forensic techniques are then applied to extract security relevant information. DACSA showed to have minimal impact on host and guest operations during the snapshot and memory extraction process. Due to space constrains we limit the discussion on fast snapshotting and memory extraction. We encourage the reader to read previous work on DACSA [7].

3.2.2 Carving Segments from the Logical Copy

Memory segments[1] are carved from the logical copy based on the VM guest OS semantics. For Windows, memory segments are obtained from walking each process's Virtual Address Descriptor (VAD) tree structure [5]. The VAD tree is a binary tree where each node represents an allocated memory space by the process and thus a single memory segment. Each VAD tree node contains a starting and ending virtual address specific to the process. We extract memory segment data using these addresses and the process's page-tables.

3.2.3 Shipping Memory Segment Data

Each segment is processed during segment extraction page-by-page using the VAD starting and end address. A hash of each page is generated to determine if the page data needs to be sent to the schedule. First, the hash is looked up locally. If not found, the hash is added to a batch of hashes to send to the scheduler. Once the batch is sent, the schedule checks each page in a global hashtable and returns which pages are

[1]We use "memory segment" to refer to the carved unit of memory, which does not correspond to an x86 segment.

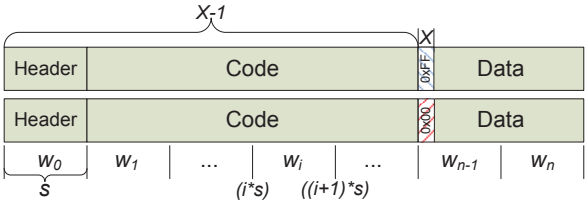


Figure 2: Two binaries that differ at byte X have an identical $X-1$ bytes prefix. Binaries are split into windows size s. Hashes of window w_{n-1} identifies the difference.

needed for scanning. If the page hash value has been stored before, meta-data for the hash is stored with the hash identifying the VM, segment, and offset. The scheduler then uses the hash to reconstruct the segment for pre-processing and scanning. This process is similar to VM memory deduplication techniques [8].

3.3 Segment Similarity Detection Off-Host

For segments that are similar but not identical, we observe the scanner may unnecessarily duplicate scanning effort. For example, Figure 2 depicts two identical binaries that contain only a single byte difference at index X. A traditional naive scanner will identify each segment as unique and scan each segment separately from low bytes to high. We observe that after each scan instance has processed the first $(X-1)$ bytes in the example, the scanner signature matching states will be equal as the prefix $(X-1)$ bytes are identical for both binaries. Thus, the scanner will have duplicated matching computation for this prefix.

Prefix information provided to the scanner allows a single unique prefix to be scanned only once across similar segments. This is beneficial for binaries used across many processes and machines as most will be unique but share some identical header and code sections.

3.3.1 Prefix Matching

SEER needs to quickly identify common prefixes across all segments. While byte-by-byte comparison provides very fine-grained matching, it comes at a large cost. We are only concerned with identifying identical prefixes which are aligned to the beginning of memory segments.

To quickly identify differences in data, previous techniques such as fuzzy hashing [11] and rolling hashes [22] have been proposed. Unfortunately, these techniques do not lend themselves to finding the longest prefix in pseudo-random data. Specifically, the effectiveness of these approaches relies on finding a suitable *trigger*. Instead, we choose a more straightforward approach which uses a rolling window to approximate the longest prefix.

Figure 2 provides a reference for partial segment hashes, which we discuss next. After segment extraction, each segment is divided into $n+1$ non-overlapping parts of data using meta-data to reconstruct segments. We refer to this data as window w. The ith window, w_i, represents bytes from $(i*s)$ to $((i+1)*s)-1$ where s is the window size. Each w_i is hashed by the scheduler in order from $i=0$ to $i=n$. Window hashes, window offsets $(i*s)$ and size (to handle boundary cases) can then be compared across segments beginning from low to high index. If two hashes at an index do not match, the index identifies the longest prefix for a pair.

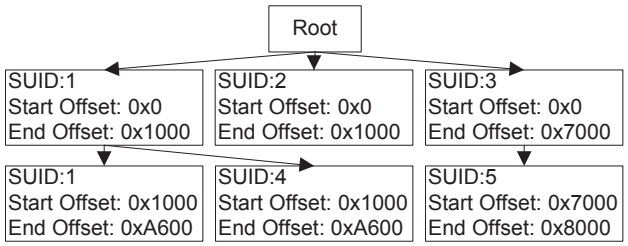
Encoding Prefix Information: SEER needs to quickly determine the longest prefixes across all segments and store relevant information for scanning. We achieve this through a combination of hash tables and building a rooted directed tree (e.g., as in Figure 3). The rooted directed tree naturally represents prefixes by encoding information in each node. As the tree is walked from the root towards children, data for segments can be processed and read in linear fashion.

Algorithm 1: Algorithm for updating scheduler.

```
   foreach S ∈ Segments do
1    node = root
     foreach w ∈ Windows do
2      if node.SUID = S.SUID
3        node.window = w
4      if w.index >= node.index + node.size
5        if hash(w) ∈ node.children
           node = node.children(hash(w))
6          AssociateWindow(node, S.SUID,w)
         else
           node =NewNode(node, S.SUID, w)
7      else
8        if node.hashes[w.index] = hash(w)
9          AssociateWindow(node, S.SUID,w)
         else
10         split node at w.index
           create two children
           node = new child for w
```

Prefix information is encoded in the tree nodes. Each node represents a unique prefix and maintains several pieces of information used for scanning and segment association. Specifically, nodes contain (1) associated segment ids ($SUIDs$) which track matched segments, (2) a range to explicitly define where the matched prefix starts and ends for the associated $SUIDs$, and (3) a hash table for looking up child nodes. To track hashes and data for each node, window hashes are stored for each index represented by the node. These hashes are used to later build and retrieve the data stored from segment extraction.

Algorithm 1 depicts how the tree is built using memory segments and windows. As windows are processed by the scheduler, the tree is walked and updated. For processing new segments, the walking begins at the root node (**line 1**). If the received hash's index does not fall in the current node's explicit range (**line 4**) then the hash value is not represented by the current node as a previous prefix has been defined for the hash's index. As a result, the current node's child hash table is searched for an association (**line 5**). A new node is added if no child node with the hash and corresponding index is found (**line 7**). If the hash matches a child, the hash is associated with the matched child node (**line 6**). During association, the SUID is added to the node's list of SUIDs with common prefix and matched indexes. If the hash's index falls in the current node's explicit range (**line 8**), the node's stored hash for index is compared (**line 9**). If identical, the associated SUID is updated for the node (**line 10**). Otherwise, a new prefix is identified and the current node is split at the window index creating two child nodes (**line 11**). To quickly add new unique prefixes, each node stores the first SUID used to create the node. If the SUID for the window hash is the same as the node's first SUID (**line 2**), the node only needs to update the current node with the window hash and index information (**line 3**). Such



SUID	VM	Address	Size	PID
1	1	0x112000	0xA600	872
2	1	0x60000	0x1000	872
3	1	0x1e000	0x7000	872
4	1	0x482000	0xA600	1788
5	1	0xff0000	0x8000	1788

Figure 3: Scan tree representing 5 memory segments.

is the case when the root node contains no children.

Figure 3 depicts a prefix matched tree with five unique memory segments. Segments with SUID 1 and 4 represent a mapped binary. We see that the longest prefix for 1 and 4 is found at offset 0x1000. Thus, the scanner would only need to scan the first 0x1000 bytes of SUID 1 and 4 one time instead of twice. SUID 2 represents a unique segment as reflected by a node with no children. SUID 3 and 5 represent a data file for browser history. We see that the longest prefix is found at offset 0x7000 when SUID 3 ends. SUID 5 is added as a child of SUID 3 as SUID 5 has an extra 0x1000 bytes in the segment. In this case, the scanner would only scan the first 0x7000 bytes of SUID 3 once. The total savings dictated by the tree is 0x8000 bytes.

3.3.2 Binary Normalization

Binaries mapped across processes often provide opportunities to match large prefixes, as per our *ieframe.dll* example. This property allows additional opportunities to deduplicate scanning work. We found that binaries have a semi-consistent structure in which static headers and code resides at the beginning. While nothing prevents code sections from being located at the end of binaries, this pattern is default for common compilers. This attribute of binaries ensures prefixes match at least header and code sections for segments with the same backing file on disk.

Unfortunately, we found that beginning memory is not always identical due to how the underlying OS manages memory for mapped files. For example, a page may be marked not present in a process's page table but mapped in another. As a result, we use a technique for normalizing binaries to extend the length of matched prefixes.

Filling Non-Present Pages: OS memory management units employ on-demand paging to reduce the memory footprint of processes and improve performance. Only processes accessing the page has its pagetable updated to include the new physical page mapping. As a result, pagetables between processes are not synchronized for common shared binaries.

The lack of inter-process pagetable synchronization impacts prefix matching for memory segments. For example, consider two processes that both use the *ieframe.dll*. The third page of *ieframe.dll* is accessed by process P_1 but not P_2. As a result, the third page for P_1 will be marked present while the same page for P_2 is non-present and the corresponding window hashes (w_2) will not match. The longest prefix for P_1 and P_2 will be the index of w_2.

SEER conservatively normalizes memory segments by filling in non-present pages with pages found present in other segments that are mapped to the same binary file. During memory extraction, segments are grouped by binary headers, corresponding load addresses, and size. In Windows, binary files loaded from the same file on disk will all be placed at the same address in memory by the default loader.

Once grouped, each segment's data is walked to identify present pages and indexes. If a page is present, the status is checked to see if a previous data page has been stored for that index. If previously stored, a hash of the data is compared to ensure equality. In some cases, present data at a page index may not be equal and conflict across segments in the group. This occurs if a page has been modified by a process, for example, when modifying data or malicious code manipulation. In this case, we conservatively mark the page status for the page index as *unknown* and do not use the page to fill non-present pages for groups of binaries. Page filling occurs with regards to given VM.

3.4 Prefix Matched Scanning

After all memory segments have been processed, the scheduler contains a tree of nodes representing unique prefixes covering all segments. Each node in the tree contains information used by the scanner to scan the unique prefix.

To scan all segments, each node must be processed and data represented by the node scanned. We process each node by walking the tree in a Depth First Search manner. As the scanner requests data to scan, node attributes are used to determine which data to provide. A node boundary is reached when the scanner asks for data at an offset greater than the current node's end address. As a result, the scanner takes actions to walk each child.

Each child node to be processed requires the same scan state as the parent. Thus, the scan state must be saved or duplicated so each child can have its own copy to continue scanning correctly. To facilitate coping scan state, the scanner forks execution for each child node. The last child does not fork, but only assigns the last child node for processing.

Ignoring Non-Present Memory: Windows of segments found to be non-present can be skipped by the scanner as they provide no information for matching. This provides two benefits: (1) it increases scanning efficiency and (2) it reduces the possibility of false positives occurring that match for non-present regions. When a non-present window is to be scanned, the scanner simply updates the index to be read to the next present page thus preserving scanner context.

Scanner Specific Considerations: Scanning with prefix matching assumes that memory is scanned in a linear fashion. This ensures all matching efforts for the previous prefix has been complete prior to forking. Thus the scanner should not peek at data ahead beyond the current window. Reading data in non-linear fashion will result in walking child nodes prior to matching efforts thus preventing savings from scanner effort deduplication.

3.5 Implementation

We prototype SEER using QEMU [1] v0.13.0 with KVM

Table 1: Average Impact on Throughput (MB/s) and Response Time (Resp. Time) of SEER on both the VM being scanned (i.e., snapshot and shipped) and other VMs on the same host.

VM Impact	MB/s	Resp. Time	Snapshot Time	Data Shipped
Scanned	8.02%	9.14%	4.90 sec	153.4 MB
Non-Scanned	0.70%	0.76%	4.48 sec	140.4 MB

virtualization software as the host. Our host is Ubuntu 12.04 64-bit and VM environments are Windows 7 SP1 64-bit.

The memory extractor and scheduler are configured to allow both storing complete memory segments to network storage or sending only unique data windows as discussed in Section 3.2.3. For our evaluation, we save complete memory segments to allow for better support for evaluating different configurations.

SEER requires minor modifications to virus scanner software to enable best efficiency and accuracy. Thus, we adapted ClamAV, a popular open source virus scanner, to the SEER architecture with minimal modifications by hooking I/O requests through a shared library. For example, read requests for scanned memory segments are intercepted and return specific data based on the current node. If a read request crosses a scan tree node boundary, the process can immediately fork and return the appropriate values for each read. ClamAV virus definitions for PE/COFF binaries support context aware definitions. For example, virus definitions can target specific sections for binaries. The sections found in binary disk files reside at different locations in memory as sections are often page aligned while sections on disk are block aligned. Thus we modify how ClamAV interprets PE/COFF files so that sections are aligned correctly with memory. In total, we only modified 9 lines of ClamAV code to enable memory virus scanning with SEER.

4. EMPIRICAL EVALUATION

We evaluated SEER on an IBM System X server with a Xeon E5450 Quad-Core CPU with 32GB RAM. The server hosts Window 7 SP1 64-bit VMs configured with 1GB RAM and 1 VCPU with QEMU/KVM as the host. An IBM System X server with an AMD Quad-Core Opteron 2384 and 32GB RAM was used for scheduling scans, storing, and scanning data. A third IBM System X server with an AMD Quad-Core Opteron 2384 and 16GB RAM was used to benchmark the host with VMs. These machines are interconnected via two 10Gbps switches, one managing public network and one for management network.

We investigated the impact of SEER's components across the scanning process. First, we performed benchmarks on host and guest VMs to investigate the impact of fast snapshotting and shipping memory. Next, we investigated at SEER's impact on matching efforts and total scan times under varying conditions, configurations, and caching. Finally, we evaluated SEER's ability to find malware in memory and discuss false positives.

4.1 SEER Impact on Guest VMs

The guest VM is momentarily paused to create the logical copy for carving and shipping. We measured the average time a VM is paused at 0.2112 seconds with a standard deviation of 0.07359 seconds.

We measured the impact of VM snapshotting and data shipping (i.e. phase 1) for VMs being scanned and co-located VMs that are not scanned. We used SpecWeb'2009 [21] to simulate a webserver under load to measure the reduction in throughput (MB/s) and response time for a single guest VM (i.e. VM under test). The VM under test was configured with IIS, ASP.Net, and the SpecWeb banking application. SpecWeb was configured to simulate 200 simultaneous users making at least one request every second. The benchmark was run under three conditions: 1) no VMs on the host in phase 1, 2) the VM under test in phase 1, 3) the VM under test is co-located with VM in phase 1. Table 1 contains the results. A VM can expect an 8.02% reduction in throughput and a 9.14% reduction in response time during the 4.9 second phase 1. A co-located VM not scanned can expect a 0.7% reduction in throughput and 0.76% in response time during phase 1. We note here that the average amount of data shipped was 153.4 MB and 140.4 MB for these configurations however this number is highly dependent on the processes and services running. The average time to snapshot and ship the data was less than 5 seconds under load. The majority of time was spent shipping data.

4.2 Scanned Data and Scan Times

We investigated SEER under the following configurations: Naive, Prefix Matched with Normalization (**PMN**), and Prefix Matched Normalization ignoring Non-Present data (**PMNP**). Naive refers to an unmodified ClamAV using full segment hashes to identify previously scanned segments. PMN provides the potential savings of adopting our techniques for host based memory scanners. PMNP shows the most efficient configuration for scanning memory that can be both present and non-present data.

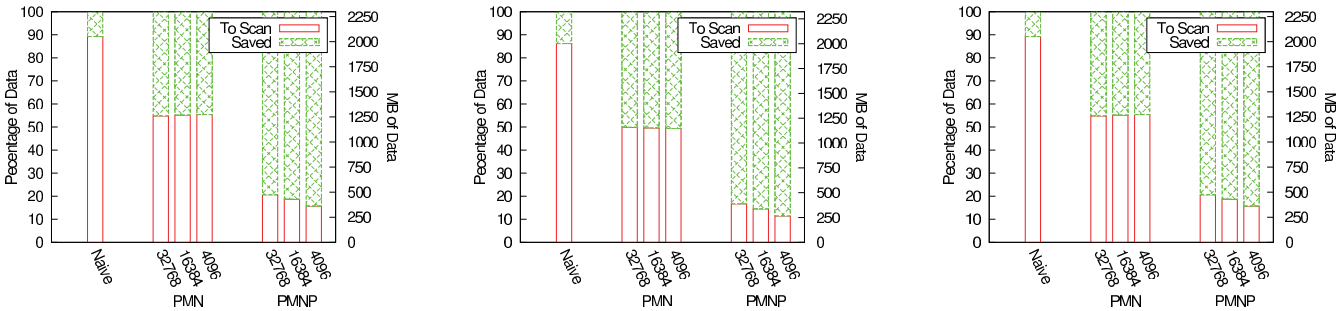
PMN and PMNP configurations were tested with different windows sizes to investigate window size impact on efficiency. For the smallest window, we used 4KB. This naturally maps to present/non-present page data. We also used different multiples of 4KB aligned windows. The Naive configuration does not have a window size but hashes the entire memory segment to identify duplicates.

We configure scanning to be performed serially on a single processor under all above configurations. While prefix based scanning is naturally parallelizable, we use a single processor to avoid unfairly biasing in favor of SEER.

To investigate potential savings under different working conditions, we scanned VMs under three different operating environments: (1) idle machine after booting to login, (2) a loaded webserver running IIS and MySQL with 100 simulated simultaneous clients navigating a PHPBB web application, and (3) VMs running a set of the following unique GUI applications: IDA Pro, Chrome, Internet Explorer 64-bit, Notepad, Windows Update, IIS Administrator, WampServer Installer, OllyDBG, Process Monitor, and Visual Studio Express 2010. All workloads are run on the same VM image. Workload (2) looks to investigate the potential impact of a high number of reads and writes but similar number of processes as workload (1) without loading into the Windows desktop. Workload (3) pertains to loading the desktop environment to investigate the impact of unique processes on the amount to scan and scan times. To remove non-deterministic nature of memory, carving and scanning were done on static snapshots of memory for each operating

Table 2: Average cost of snapshotting and shipping VM memory for remote scan.

	1 VM				10 VMs			
	Cold		Cached		Cold		Cached	
Type of VM	Time	Shipped	Time	Shipped	Time	Shipped	Time	Shipped
Login	6.62 Sec	223.8 MB	6.07 Sec	46.39 MB	5.89 Sec	140.0 MB	5.47 Sec	37.63 MB
Webserver	7.26 Sec	325.1 MB	5.85 Sec	76.24 MB	6.42 Sec	155.4 MB	5.29 Sec	36.56 MB
Profiled	8.73 Sec	382.0 MB	8.23 Sec	137.2 MB	7.79 Sec	239.9 MB	7.39 Sec	111.67 MB



(a) Boot to login (b) Loaded IIS webserver (c) Unique GUI applications

Figure 4: Percent Data savings and to scan for a single VM. X-axis is window size.

environment. We took 10 snapshots for each workload type and incremental snapshots for all 10 at 5 minute intervals.

We detail the data savings and total scan times for scanning both a single machine and across multiple machines. Using multiple machines looks at potential savings of matching prefixes across machines. During scanning, memory segment data is read off the disk and provided to the scanner for analysis. Table 2 shows the average cost for snapshots and data shipping of each operating environments for both a single VM snapshot (i.e. 1 VM) and all 10 VM snapshots.

Data Saved/Scanned: Figure 4 shows the amount of data scanned and saved for a single VM instance across workloads. The x-axis shows the windows sizes in bytes. We see that naive hashing can save 12-14% of data from being scanned across all workloads. For PMN, we see the window sizes have little effect on the amount of data scanned with a savings of 40-45%. PMNP shows the greatest savings of 80-90%. Window sizes have a small impact on PMNP at 5% impact as a result of the smaller windows being able to identify more non-present regions. The differences between Figures 4(a) and 4(b) show that a busier machine does not impact Prefix Matching performance. In fact, we saw that a busy machine had an additional savings of ≈5%. For Figure 4(c), we see that more data needs to be scanned as more libraries and processes are loaded and run. This will affect the total time to scan a VM.

Figure 5 shows the percentage of data scanned for each VM workload as more VMs are added to be scanned. As a result, the tree encodes prefixes across multiple VMs allowing the potential prefix matching between VMs. We remove the variable window size and set it to 4KB across each scan as the previous figure indicates windows sizes have little impact on percent data to scan or save.

For the idle machine at login and the loaded webserver, the graph shows additional savings by adding VMs to scan at ≈20% and ≈60% for Naive and PMN. PMNP has little additional savings by adding VMs to scan.

For the unique GUI based tests, we see no additional savings across VMs. This is a result of different libraries and processes being loaded for different applications. This indicates that adding VMs to scan only provides benefit when the VMs being scanned are running similar applications.

Total Scanning Time: SEER adds processing time overhead which needs to be accounted when evaluating the performance of scanning. The overhead of SEER can be found from building the scan tree and forking the scanner thousands of times. To investigate the impact of this overhead, we scan the memory segments for all of the above configurations to investigate the wall time savings of using SEER.

Figure 7 depicts the per VM total scan times across different configurations. Each graph breaks down the costs for pre-processing (i.e., prefix matching and storage), forking scanner processes, and scanning. Across graphs we can see that the window size impacts total scan times by increasing pre-processing overhead. For PMN configurations, a 4KB window adds 1.5-2 minutes of scan time compared to a window size of 32KB. For PMNP, a 4KB window still adds additional overhead, but this is mitigated by savings during scanning. Overall, PMNP reduces the total scan time by 62-72% compared to Naive for a single VM. In line with our data saving results, Figures 7(a) and 7(b) show little difference in total scan time under different workloads. Figure 7(c) has a larger total scan time due to the difference in the amount of data scanned.

Figure 6 depicts the scan times across multiple VMs for a window size of 32KB as it provided the most scan time savings for a single VM as per the previous graph. The data shows that by adding more VMs, the total savings of PMN becomes ≈46% more efficient then Naive for the same workload. On the other hand, PMNP has little extra savings by adding additional VMs at 73.5% total scan time savings over Naive for the same workload.

The overhead associated with "Pre-Process" phase is independent of the virus scanner complexity. Thus, the results discussed above represent the computational and time savings when adapting a virus scanner containing the same complexity as ClamAV. If a more complex scanner is adapted, the savings is expected to be greater than the results above.

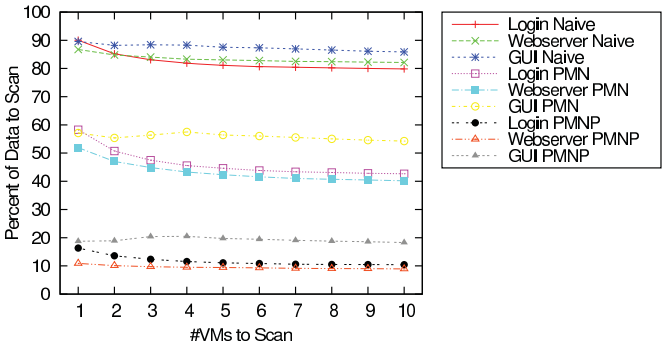


Figure 5: Percent Data to scan as VMs are scanned together. Windows size fixed to 4KB.

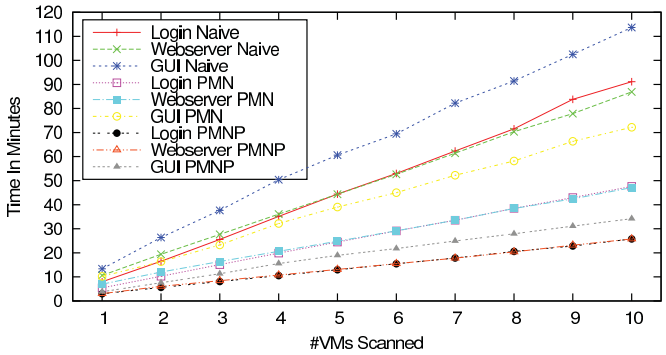
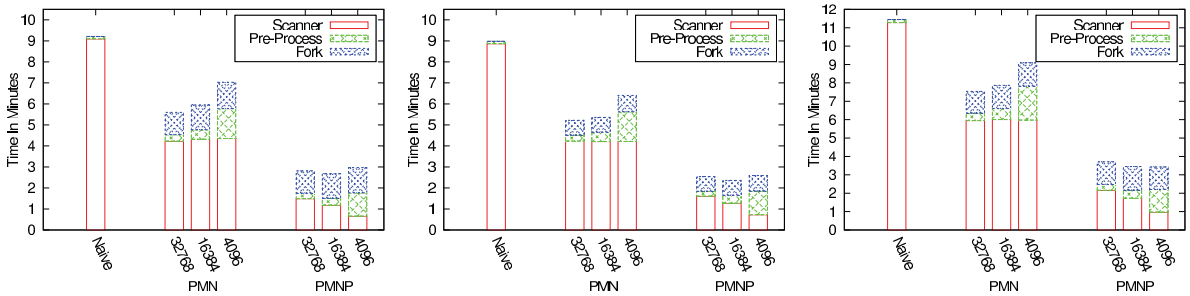


Figure 6: Total scan time across multiple VMs. Window sized fixed at 32KB.



(a) Boot to login (b) Loaded IIS webserver (c) Unique GUI applications

Figure 7: Total scan time for single VM under different loads and configurations.

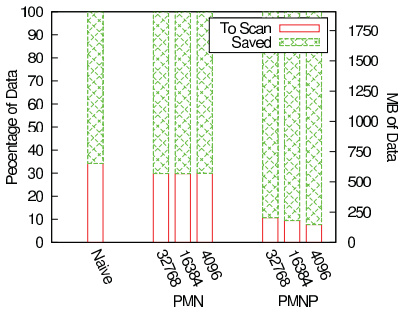


Figure 8: Percent data savings and to scan for warm cache Boot to login VM. X-axis is window size.

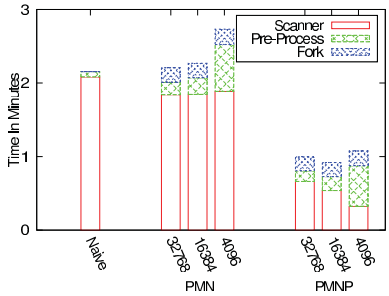


Figure 9: Total scan time for Boot to login VM with a warm cache.

Caching: After a VM has been scanned, hashes and status information of the scanned memory segments can be cached to prevent re-scanning identical segments in the future. We looked at the effectiveness caching segment hashes for future scans. Figure 8 and Figure 9 depict the benefits of warm caching for the "Boot to Login" configuration. Figure 8 shows a reduced percent data savings between Naive and PMN with a difference of 5-10% while PMNP and Naive have a savings difference of about 25-35%. Figure 9 shows the scan times for PMN and Naive were equal due to the overhead of pre-processing and forking. However, we still see a ≈50% reduction in scan times for PMNP at a window size of 32KB. We found similar results for both the loaded IIS server and GUI application configurations.

Write Working Set: During the Pre-Process phase in Figures 7 and 9, the QEMU process copy remains resident in memory and the guest VM continues to run. During this time, Copy-On-Write in the host is managing the Write Working Set, set of memory pages which have changed since forking. If the Write Working Set becomes too large, host memory can become depleted due to the dynamic nature of memory. We observed that our Write Working Set changes at rate of 100-300 MB per minute for the first minute. Given, the snapshot and shipping time is typically 4-10 seconds, the Write Working Set is not a concern.

4.3 Malware Identification/False Positives

SEER's effectiveness at identifying malware in memory is dependent on the adapted virus scanner's signatures. However, we still wish to verify the SEER correctly extracts memory segments that map to existing virus signatures for memory only malware and file based malware and investigate false positives using existing signatures.

To investigate ClamAV's ability to extract segments to find malware using existing signatures, we compromised one VM with the file based malware Cerberus RAT and another VM with the Memory-Only Malware Meterpreter. Both malware were configured to infect *explorer.exe*. SEER successfully found both malware running in the memory of the

Table 3: Malware samples from malc0de.com by type identified by SEER.

Malware Types	# Samples
Trojan.Adload	2
Trojan.NSIS.Agent	7
NSIS.Clicker.Agent	1
W32.AdInstall	1
Trojan.IRC.Zapchast	1
WIN.Trojan.DarkKomet	1
Trojan.Adload	1
Adware.Cpush	1
Trojan.Spy	2
Total	17

explorer.exe process showing that SEER can be used to identify both file-based malware and Memory-Only-Malware. We then collected 652 samples of malware from malc0de.com [13]. The samples were scanned with an unmodified ClamAV and 60 samples were identified as Malware. 17 samples were successfully run in a guest VM. SEER was able to correctly identify all 17 samples of malware from memory. Table 3 shows the malware types scanned identified by ClamAV.

To investigate false positives, we recorded all segments identified as malware by our modified ClamAV across each benign scan configuration. This set represents the false positives. We found all observed false positives can be categorized into two types. First, ClamAV marked host based virus scanner definitions as malicious. Specifically, ClamAV identified Windows Defender virus definitions loaded into memory as malware. On disk these definitions files are obfuscated thus do not trigger false positives. This problem can be addressed by white listing host scanner processes in memory. The second false positive type occurs from overly-broad ClamAV signatures which use MD5 hashes of zeroed data as virus definitions. We found 154 out of 2,056,340 MD5 signatures matched only zeroed data. These signatures should not be used for memory scanning. Upon removing these signatures from the definitions database and whitelisting scanner processes, no false positives were triggered.

5. DISCUSSION

Deploying SEER: SEER shows the best performance using PMNP configuration with a window size at 32 KB. The performance of PMNP is highly dependent on memory segments containing non-present data. If adapting Prefix Matched scanning for a host based memory scanner, non-present information is not encountered and thus performance would be comparable to PMN.

Security and Privacy Concerns: Memory contains the entire running state of a machine and thus contains very sensitive information such as encryption keys, browser data, etc. SEER extracts memory segments such that finding sensitive data may be easier for malicious parties. For example, a malicious party can quickly identify the webserver process which is known to contain stored SSL certificates. Thus, care must be taken when storing memory segments prior to scanning and properly removing benign data after analysis.

Virus scanners themselves have been known to be a target for attack [17]. Since SEER accesses sensitive data across potentially many VMs, care should be taken to ensure compromise of the SEER scanner itself is mitigated and identifiable. The simplest solution would be to isolate the scanner to a monitored machine and network.

Alternative Operating System Support: Our current implementation of SEER only supports Windows environments. However, SEER can support other operating systems by porting memory carving techniques for guest operating systems. We believe this to be a straightforward porting effort for well-known operating system structures. For example, Linux have structures called virtual memory areas that map directly to memory segments. For unknown operating systems, VMI techniques can be used to identify internal process structures and virtual memory [6].

Limitations: Currently, SEER does not attempt to extract paged-out memory from VM page-files or swap partitions. While it is technically feasible to extract paged-out memory from disk files, the overhead associated with extraction can have a significant negative impact on both host and guest VM operations. As a result, malware can attempt to avoid scanning by hiding in paged-out regions of memory which can cause false negatives. However, we argue that hiding malware in swap is non-trivial as (1) malware must use a deterministic mechanism to force the operating system to page out malicious content, (2) the deterministic mechanism must be in memory thus providing an indicator for hiding, and (3) scanning of memory cannot be bypassed.

SEER relies on memory forensic techniques to find and walk kernel data structures to extract memory segments. These techniques are commonly used by memory forensic tools [24] and popular Virtual Machine Introspection (VMI) solutions [6, 9]. However, walking kernel data structure is potentially vulnerable to Direct Kernel Object Manipulation (DKOM) attacks [3]. Unfortunately, identifying and preventing all DKOM attacks remains an open problem.

SEER provides detection of memory compromise but not protection. Thus, SEER cannot prevent infection of well-known malware or exploits. While security products often protect against infection by hooking file operations, exploitation commonly uses memory only techniques. As a result, security products can be disabled without notification once an exploit has code execution.

Finally, there is concern that attackers can allocate large memory segments by simple virtual memory allocation. SEER mitigates this potential impact by providing page status to the scanner to bypass non-present memory. To force SEER to scan data, the attacker must fill allocated memory with random data thus requiring the scheduler to store unique data. However, this process would deplete system resources exposing malicious activity.

Using Existing Virus Scanner Signatures: We show in our evaluation that existing signatures for file based scanning can be used to identify malware in memory. This obviates creating all new signatures to adapt to memory scanning. However, the effectiveness for matching viruses is dependent on the correctness and precision of signatures. Furthermore, signature based scanning has known limitations and thus SEER suffers from the same limitations.

6. RELATED WORK

Jiang et al. [9] provided virus scanning for individual virtual machines by scanning files on running Virtual Hard Drive (VHD) and physical memory from the hypervisor. Unfortunately, scanning from the hypervisor can impact host and VM performance. Also, scanner vulnerabilities can lead

to complete system compromise putting other VMs on the host at risk [17]. SEER instead performs scanning off host in an isolated environment while scanning virtual memory to correctly match virus definitions.

Wei et al. [25] presented scanning offline logical VHDs stored in a customized deduplicated image library. The authors were able to show a reduction in virus scanning efforts by only scanning unique files once across all logical VHDs. Similarly, Soules et al. [20] discussed how to efficiently schedule virus scanning for an enterprise network by scanning unique files only once. These approaches rely on the static nature of files on disk to reduce scanning efforts. However, SEER instead focuses on efficiently scanning dynamic memory for live running virtual machines.

Oberheide et al. presented CloudAV which describes how to increase the effectiveness of file virus scanning by using multiple pieces of scanning software [17]. Services such as VirusTotal, Jotti, and NoVirusThanks provide multi-scanner virus scanning similar to CloudAV without an end client. CloudAV and similar tools rely on data files being similar across machines and thus only unique files need to be scanned. However, the authors do not discuss how to handle dynamic data. SEER instead provides a new technique to increase scanning efficiency of memory.

Trend Micro recently released a VMWare appliance called "Deep Security" capable of agentless virus scanning of virtual machine files using a central virtual appliance for VMWare [4]. Unfortunately, Deep Security only supports file based scanning and relies on the static nature of files to enable scalability. In contrast, SEER tackles the hard problem of enabling virus scanning of highly dynamic data.

7. CONCLUSION

In this work we proposed SEER, an architecture for enabling Memory Virus Scanning-as-a-Service for virtualized environments. SEER overcomes several challenges of memory scanning by transparently and efficiently scanning memory outside of the context of the analyzed VM. SEER quickly and correctly acquires virtual machine memory and we show in the empirical evaluation our technique has minimal impact on VM operations and hosting environments. We propose a new approach for scanning memory by normalizing memory, matching similar prefixes across memory, and then scanning based on prefix data. Our results show that our approach has a 72% reduction in wall time and scanning efforts compared to existing naive scanning approaches. Our approach can be directly applied to existing host based memory scanners to reduce scanning efforts by up to 46%.

8. REFERENCES

[1] Fabrice Bellard. Qemu, a fast and portable dynamic translator. In *Proceedings of the annual conference on USENIX Annual Technical Conference*, 2005.

[2] Antonio Bianchi, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Blacksheep: detecting compromised hosts in homogeneous crowds. In *Proceedings of the 2012 ACM conference on Computer and communications security*, 2012.

[3] Jamie Butler. Dkom (direct kernel object manipulation). *Black Hat Windows Security*, 2004.

[4] Agentless security. Trend Micro. `http://www.trendmicro.com/cloud-content/us/pdfs/business/sb_vmware-agentless-security.pdf`.

[5] Brendan Dolan-Gavitt. The vad tree: A process-eye view of physical memory. *Digit. Investig.*, September 2007.

[6] Brendan Dolan-Gavitt, Tim Leek, Michael Zhivich, Jonathon Giffin, and Wenke Lee. Virtuoso: Narrowing the semantic gap in virtual machine introspection. In *Proceedings of the 2011 IEEE Symposium on Security and Privacy*, SP '11, 2012.

[7] Jason Gionta, Ahmed Azab, William Enck, Peng Ning, and Xiaolan Zhang. Dacsa: A decoupled architecture for cloud security analysis. In *Proceedings of the 7th Workshop on Cyber Security Experimentation and Test*. USENIX, 2014.

[8] Diwaker Gupta, Sangmin Lee, Michael Vrable, Stefan Savage, Alex C. Snoeren, George Varghese, Geoffrey M. Voelker, and Amin Vahdat. Difference engine: harnessing memory redundancy in virtual machines. *Commun. ACM*, 53(10):85–93, October 2010.

[9] Xuxian Jiang, Xinyuan Wang, and Dongyan Xu. Stealthy malware detection through vmm-based "out-of-the-box" semantic view reconstruction. In *Proceedings of the 14th ACM conference on Computer and communications security*, 2007.

[10] Tomasz Kojm. Clamav, 2004. `http://www.clamav.net`.

[11] Jesse Kornblum. Identifying almost identical files using context triggered piecewise hashing. *digital investigation*, 3:91–97, 2006.

[12] Michael Ligh, Steven Adair, Blake Hartstein, and Matthew Richard. *Malware Analyst's Cookbook and DVD: Tools and Techniques for Fighting Malicious Code*. John Wiley & Sons, 2010.

[13] Malc0de. *Malc0de*, 2007. `http://malc0de.com`.

[14] About the metasploit meterpreter. Offensive-security, 2012. `http://www.offensive-security.com/metasploit-unleashed/About_Meterpreter`.

[15] Ned Moran, Sai Omkar Vashisht, Mike Scott, and Thoufique Haq. Operation ephemeral hydra: Ie zero-day linked to deputydog uses diskless method. FireEye, Nov 2013.

[16] NSSLabs. Endpoint protection products 2010 group test summary. NSS Labs, 2010.

[17] Jon Oberheide, Evan Cooke, and Farnam Jahanian. Cloudav: N-version antivirus in the network cloud. In *USENIX Security Symposium*, pages 91–106, 2008.

[18] Fahmida Y. Rashid. Watering hole attacks scoop up everyone, not just developers at facebook, twitter. PC Mag, March 2013.

[19] Sans institute infosec reading room: What is code red worm. Sans Institutue, 2001.

[20] Craig A.N. Soules, Kimberly Keeton, and Charles B. Morrey, III. Scan-lite: enterprise-wide analysis on the cheap. In *Proceedings of the 4th ACM European conference on Computer systems*, 2009.

[21] Standard Performance Evaluation Corporation. Specweb2009.

[22] Andrew Tridgell and Paul Mackerras. The rsync algorithm, 1996.

[23] VMWare. Vmware vshield endpoint, 2010. `http://www.vmware.com/files/pdf/vmware-vshield-endpoint-ds-en.pdf`.

[24] The volatility framework: volatile memory artifact extraction utility framework. Volatile Systems.

[25] Jinpeng Wei, Xiaolan Zhang, Glenn Ammons, Vasanth Bala, and Peng Ning. Managing security of virtual machine images in a cloud environment. In *Proceedings of the 2009 ACM workshop on Cloud computing security*, 2009.

MACE: High-Coverage and Robust Memory Analysis For Commodity Operating Systems

Qian Feng Aravind Prakash Heng Yin
Syracuse University
{qifeng,arprakas,heyin}@syr.edu

Zhiqiang Lin
University of Texas at Dallas
zhiqiang.lin@utdallas.edu

Abstract

Memory forensic analysis collects evidence for digital crimes and malware attacks from the memory of a live system. It is increasingly valuable, especially in cloud computing. However, memory analysis on on commodity operating systems (such as Microsoft Windows) faces the following key challenges: (1) a partial knowledge of kernel data structures; (2) difficulty in handling ambiguous pointers; and (3) lack of robustness by relying on soft constraints that can be easily violated by kernel attacks. To address these challenges, we present MACE, a memory analysis system that can extract a more complete view of the kernel data structures for closed-source operating systems and significantly improve the robustness by only leveraging pointer constraints (which are hard to manipulate) and evaluating these constraint globally (to even tolerate certain amount of pointer attacks). We have evaluated MACE on 100 memory images for Windows XP SP3 and Windows 7 SP0. Overall, MACE can construct a kernel object graph from a memory image in just a few minutes, and achieves over 95% recall and over 96% precision. Our experiments on real-world rootkit samples and synthetic attacks further demonstrate that MACE outperforms other external memory analysis tools with respect to wider coverage and better robustness.

Categories and Subject Descriptors

D.4.6 [**Operating Systems**]: Security and Protection—*Invasive software*

General Terms

Security

Keywords

Memory Analysis, Random Surfer, Rootkit Detection

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.
ACSAC '14 December 08 - 12 2014, New Orleans, LA, USA
Copyright 2014 ACM 978-1-4503-3005-3/14/12 ...$15.00.
http://dx.doi.org/10.1145/2664243.2664248.

1. INTRODUCTION

Memory analysis has become increasingly valuable in digital crime investigation and malware analysis, as it extracts *live* digital evidence of attack footprints from the volatile memory state of a running system, which cannot be obtained from traditional hard disk based forensic analysis. Memory analysis is particularly beneficial for cloud computing security, because one can quickly scan a large number of virtual machine states to detect malicious activities, without installing security agents (which is inconvenient and can be easily subverted) inside the virtual machines. For example, a recent work proposed to detect rootkit infestation in homogeneous virtual machines in the cloud [3].

However, there exist several long-standing challenges in memory analysis especially for closed-source operating system (OS) such as Microsoft Windows.

(1) Low coverage. Without access to the source code of the commodity operating system, memory analysis tools can only resort to public symbols and documentations. As a result, these tools (e.g., Volatility [30]) can only identify documented objects (whose definitions are publicly available) and follow the pointers whose target types are also documented.

(2) Ambiguous pointers. Generic pointers (e.g., `void *`, `LIST_ENTRY`, and `struct list_head`) are prevalent in data structure definitions. It is hard to determine the exact target types for these generic pointers, and it is common for a generic pointer to have multiple type candidates. As a result, it is difficult to follow these generic pointers to identify the target objects. Pointers can also be dangling, and following the dangling pointers would lead to extraction of bogus objects.

(3) Lack of robustness. Because of such a low coverage, it is very easy for kernel attacks to evade memory analysis. Hiding an object can be as simple as manipulating the incoming links that are followed by the analysis tools. For example, FU rootkit [14] hides a process by unlinking the corresponding `EPROCESS` object from the active process list. To evaluate the validity of a memory object, the existing tools often rely on constraints that can be easily violated, such as pool tags, string constants, object lengths, etc. In Section 5.4, we demonstrate a synthetic attack that can completely defeat the utilities in Volatility by breaking these soft constraints.

Up to now, prior research efforts have been focused on tackling only one or two challenges above. No solution can address all the challenges in a holistic fashion. To improve robustness, several robust signature schemes have been proposed [11, 21]. These signature schemes can reliably detect important kernel objects by checking invariants (either strong value invariants or points-to relationship) in the kernel data structures. These signatures may not be distinct enough or may not even exist for many kernel objects (especially small ones). Therefore, we cannot rely on these robust

signature schemes to achieve high coverage, not to mention that performance overhead is high for repeatedly searching signatures one by one throughout the memory.

Some efforts on data structure reverse engineering (such as REWARDS [22] and Howard [29]) may help extract kernel data structures definitions from commodity OSes. Potentially these system can help identify previously undocumented objects and links, and thus improve the coverage. However, these systems have only demonstrate their capabilities on relatively small user-level programs. Complete reverse engineering of kernel data structures is still a daunting task due to the complexity of the commodity OS kernel code and the kernel data structures.

In this paper, we present MACE[1], a holistic solution that meets all the following requirements:

(1) **Binary only approach.** MACE uses only the binary code of an OS, the public symbols, and documented data structure definitions to perform memory analysis. As a result, MACE is well suited for external forensic analysts to analyze closed-source OS like Windows.

(2) **Robustness.** To achieve high robustness, MACE relies on *only* points-to relations (or pointer constraints), which are generally hard to violate, to identify kernel objects. Furthermore, MACE evaluates both deterministic and probabilistic pointer constraints throughout the entire kernel memory space, to find a nearly optimal solution. Therefore, even if an attacker manages to manipulate some pointers, these "injected" errors would likely be corrected by the remaining pointers in the memory during this *global* evaluation process. Thus, the attack impact on the overall identification results is minimized.

(3) **High coverage and accuracy.** MACE can reconstruct a nearly complete kernel object graph, which consists of both documented and undocumented kernel object instances, and the connections among them. For undocumented objects, MACE can further discover certain type information for the pointer fields in these objects. For instance, MACE can identify function pointers and target types for data pointers.

(4) **Good efficiency.** MACE can scan a memory image and build a kernel object graph just a few minutes[2]. In contrast, the existing robust signature schemes [11, 21] use several minutes to only identify objects of a single type.

The core idea of MACE is to conduct supervised learning on pointers. That is, we first collect pointer constraints from a set of training memory images, in which kernel objects are correctly labeled by dynamic binary analysis. With the collected pointer constraints, we then perform probabilistic inference on pointers in an unlabeled memory image in a collective and correlative manner, to correctly label the pointers in the image. From these labeled pointers, we then reconstruct a nearly complete kernel object graph, for memory forensic purposes.

We leverage a key insight that the kernel object graph is a small-world network [13]: most kernel objects can be reached from other kernel objects within a few hops. A link from one object to another imposes a type constraint (either deterministic or probabilistic) on each side. The type constraint indicates the likelihood of a directly connected object to be of a particular type. The type constraints will accumulate and propagate to the objects that are not directly connected. Eventually, these constraints will be broadcast to the entire network until a convergence is reached.

We evaluated MACE for two closed-source operating systems: Windows XP SP3 and Windows 7 SP0 and found that MACE can achieve high recall and precision (95% and 96%, respectively) for Windows XP and Windows 7. The errors mostly come from undocumented objects and volatile memory allocations.

We further evaluated the performance of MACE on memory images infected with real-world malware samples to demonstrate how MACE facilitates kernel rootkit identification. With a more complete coverage of kernel objects, MACE recognized malicious function pointers in both documented and undocumented data structures, and detected hidden objects more reliably. At last, we devised two synthetic kernel attacks to show how fragile the existing memory analysis tools (such as Volatility) can be, and how resilient MACE is against these attacks.

2. PROBLEM STATEMENT & OVERVIEW

2.1 Problem Statement

Given a memory image, we aim to reliably identify nearly all the kernel objects and connections between them, without access to the OS source code. We rely on public symbols, public data structure definitions. This public knowledge is used by the existing memory analysis tools (e.g., Volatility [30]). We attempt to improve the coverage and robustness of these third-party memory analysis tools, by leveraging the same amount of knowledge.

In addition to identifying documented kernel objects, we also aim to extract partial knowledge of undocumented kernel objects. In particular, we would like to discover types of the pointer fields, including both data pointers and function pointers. This knowledge on pointers can help obtain a big picture of the entire kernel object graph and benefit security analysis on this graph (e.g., kernel rootkit detection). In other words, our goal is not to reverse engineer the kernel data structure definitions as REWARDS [22] and Howard [29], although our technique can be combined with these techniques to improve the quality of data structure reverse engineering.

We formalize the problem of kernel object labeling as follows: $M = \{m_i | 1 < i < |M|\}$ denotes the kernel memory space, where m_i is the ith machine word and $|M|$ is the total number of machine words in the kernel address space. Our goal is to assign a label l to each m_i. A label l is defined as a pair of object type and offset $l = (t, o)$, where $t \in T$ and $o \in [0, \texttt{sizeof}(t))$. Here T denotes the space of all object types.

To ensure high robustness, our solution cannot rely on soft constraints that can be easily manipulated by attackers, such as integer and string constants. For example, checking pool tags and the object size from the `OBJECT_HEADER` in Windows definitely helps verify the object types (both Volatility [30] and MAS [8] use this method to resolve type ambiguity). However, kernel rootkits can easily violate these soft constraints to evade and mislead these memory forensic tools. It means that our solution can only rely on pointer constraints, which are more difficult to tamper with. We should also anticipate that although complete sabotage of pointer constraints is not possible, attackers may manage to manipulate a certain amount of pointers. Therefore, our solution should tolerate pointer manipulation attacks to a certain degree.

[1]MACE stands for Memory Analysis through Correlative Evaluation.

[2]The current implementation of MACE is mostly in Python for quick prototyping. A C/C++ implementation would further reduce the analysis time to tens of seconds.

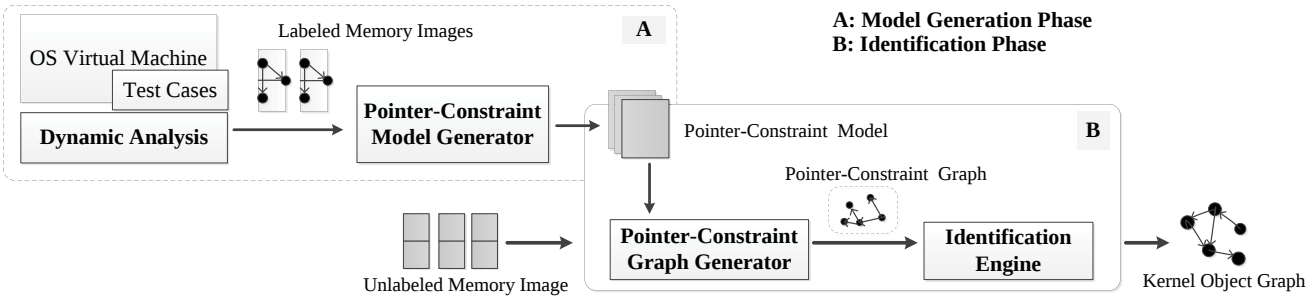


Figure 1: System Overview. The model generation phase A outputs the pointer-constraint model. The identification phase B detects the kernel object graph on the unknown memory image.

2.2 System Overview

We propose to take a probabilistic inference approach to label kernel objects based on their pointer constraints. Figure 1 depicts this workflow. Essentially, we propose a supervised learning technique. In the model generation phase, we perform dynamic binary analysis on the OS kernel to label kernel objects and learn a pointer-constraint model. Then in the identification phase, we will use this model to identify kernel objects in an unknown memory image.

Model Generation Phase. For a closed-source operating system (like Windows), we perform dynamic binary analysis to label kernel objects while the OS is running inside a virtual machine. These labeled kernel objects are then used to generate the pointer constraint model, which captures the probabilistic type constraints between pointer fields. To ensure the training is well-rounded, we conduct a set of test cases to exercise different components of the operating system, such as filesystem, network, IO, process/module/thread management, etc. Consequently, the recorded memory images (with labeled kernel objects) capture diverse system states under these workloads.

If the source code of an OS is available, we could generate this pointer-constraint model in two ways. We could perform points-to analysis on the source code directly to generate extended type graph (as done in KOP [4] and MAS [8]), and then simply derive a model from the extended type graph. For a generic pointer, we would have to assign equal probability to each possible target type. To generate a model that better reflects the system states at runtime, we could also perform dynamic analysis described above. It means that if a target type for a generic pointer appears more often than the others at runtime, it would have higher probability. The model generated this way would lead to better classification results.

Identification Phase. Given the pointer-constraint model for one OS version and an arbitrary memory image of the same OS version, in the identification phase, MACE tries to identify kernel objects and their pointer relationships.

The problem of labeling pointers in the memory image based on the pointer constraints is equivalent to searching the optimal type assignment for each pointer under the given constraints. A plausible solution to this problem is Maximum Likelihood Estimation (MLE), wherein every possible assignment solution is enumerated and evaluated in terms of likelihood, i.e., the number of satisfied constraints. However, this solution proves to be NP-hard. Therefore, it is too expensive to iterate through all possible types for tremendous amount of pointers in the memory.

We approach this problem by using the *random surfer model* [16], which has been commonly used for complex networks, such as page ranking on the web [16]. Intuitively, in random surfer model, a score associated with each node in the graph is equivalent to the likelihood of this node being visited by the "random surfer". The likelihood of a node being visited is determined by how likely its neighbors are visited and how likely the "surfer" travels from a neighbor to this node. The random surfer model allows for effectively evaluating the scores for all nodes in the graph such that it is scalable even for very large graphs (e.g., the internet). To the best of our knowledge, we are the first to apply the random surfer model to the problem of memory analysis.

In our problem domain, a node represents a labeled pointer, and an edge from one node to another dictates a confidence level that the source pointer has on the target pointer. In other words, the confidence level is a conditional probability on how likely the target pointer is correctly labeled given the source pointer is correctly labeled. We call this graph a *pointer-constraint graph*. We then apply the random surfer algorithm to calculate a nearly optimal score for each node (i.e., a pointer with a particular label). Finally, we compute object-level scores based on these pointers' scores and identify true kernel objects.

3. MODEL GENERATION

For a closed-source operating system, we generate the pointer-constraint model in two steps: 1) we conduct dynamic analysis to label kernel objects in the training data set; and 2) we learn the model by conducting statistical analysis on the training data.

3.1 Labeling Kernel Objects

We monitor the execution of the OS kernel and observe how kernel objects are allocated and de-allocated, and how these kernel objects are connected with each other. As we observe the actual binary execution of the OS kernel, we can obtain the ground truth, which is typically hard to get otherwise.

We leverage the dynamic analysis framework DECAF [17] to monitor the execution of an OS and construct the kernel data structure graph on the fly. In general, we monitor and label three kinds of kernel objects. We monitor kernel modules (e.g., `ntoskrnl.exe` and device drivers) by hooking `MmLoadSystemImage`. This is important because global data variables are located in these kernel modules. We hook `ObCreateObject` to monitor and label documented kernel objects (e.g., `EPROCESS`). Windows uses this function to create managed kernel objects (which are all documented). For other objects, we hook `ExAllocatePoolWithTag` and `ExFreePoolWithTag` to obtain a view of live memory objects in the dynamic memory pools. While there are other functions to allocate and free memory regions in the kernel, these two are the root functions. All the functions to be hooked are located in the main

kernel component `ntoskrnl.exe`, and these functions' offsets can be obtained from the public symbol information.

In this way, we can precisely label kernel modules and documented kernel objects. However, for undocumented objects, we rely on their pool tags obtained from the `ExAllocatePoolWithTag` function call. This pool tag labeling mechanism is fairly common in the modern OSes. For example, SLAB in UNIX-like systems is a similar mechanism. Of course, several problems may arise if we simply label undocumented objects by their pool tags: 1) an object allocated with a pool tag may consist of multiple inner objects, which become invisible; and 2) objects of the same type may be allocated using several different pool tags. As reverse engineering undocumented objects is not only main goal, we accept these limitations and leave a better labeling approach as future work. For example, we could leverage the calling context of the memory allocation routine to label the object.

Certainly, these function hooks are specific to Windows. For another closed-source operating system, we will need to rely on its public documentation and public symbol information to find a set of functions to hook and label objects properly. The general principle should remain the same.

To recognize links between these kernel objects, we examine each double-word within each object and see if the value in the double-word falls in the memory region of any kernel object. If this is true, we treat this value as a pointer field and we establish a link between these two kernel objects. Note that in the kernel space, it is common for a pointer to point to the middle of a kernel object. This approach may lead to an overestimation in our study because a non-pointer field may happen to have a pointer-like value and thus be treated as a pointer. In practice, these pointer-like data fields will not affect the detection accuracy of MACE, because these noises are filtered out in the statistical analysis (described in Section 3.3). Moreover, pointer fields may not be 4-byte aligned in certain packed data structures, so we have to search double-words in all byte locations.

3.2 Test Cases

In order to ensure that the generated model has a diverse set of kernel objects, the test programs used for dynamic analysis need to activate different OS functionalities that are as diverse as possible. We include both standard OS benchmark and common software programs to be run in the guest OS to maximize the variety and number of kernel objects created. For the standard OS benchmark, we choose `lmbench` [1], as it performs several diverse actions in networking (TCP, UDP, RPC, and pipe), filesystem (file creation and deletion, cached file read, etc.), signal handling, memory access, etc. We also select several common and complex programs to further increase the training coverage, including web browsers, media players, word processors, and PDF readers.

3.3 Statistical Analysis

Without source code, we conduct statistical analysis to learn kernel objects and their relationships based on labeled memory images. Specifically, we utilize the pointer constraint model to represent the kernel objects and their relationships. The pointer constraint model includes offset constraints and target constraints.

Offset Constraints: An offset constraint dictates the pointer offset in the kernel object. For example, the offset constraint OC(A) represented in Figure 2(b) shows that Object A with 12 bytes has three pointers at offset 0, 4, and 8.

To learn offset constraints for each object type t, we go over all the instances of that object type and examine the pointer fields in them. An offset o appears in the offset constraints of t if and only if

(a) Labeled Memory Image

Addr	Value	Label	Addr	Value	Label	Addr	Value	Label	Addr	Value	Label
0x8000	0x80B0	(A,0)	**0x8034**	0x80A8	(A,4)	**0x8068**	0x80BC	(A,8)	**0x809C**	0x1	(A,c)
0x8004	0x80A8	(A,4)	**0x8038**	0x80BC	(A,8)	**0x806C**	0x1	(A,c)	**0x80A0**	0x1234	(C,0)
0x8008	0x80BC	(A,8)	**0x803C**	0x1	(A,c)	**0x8070**	0x80B0	(A,0)	**0x80A4**	0x80B0	(C,4)
0x800C	0x1	(A,c)	**0x8040**	0x80B0	(A,0)	**0x8074**	0x80BC	(A,4)	**0x80A8**	0xff	(D,0)
0x8010	0x80B0	(A,0)	**0x8044**	0x80A8	(A,4)	**0x8078**	0x80BC	(A,8)	**0x80AC**	0x80B0	(D,4)
0x8014	0x80A8	(A,4)	**0x8048**	0x80BC	(A,8)	**0x807C**	0x1	(A,c)	**0x80B0**	0x1234	(B,0)
0x8018	0x80BC	(A,8)	**0x804C**	0x1	(A,c)	**0x8080**	0x80B0	(A,0)	**0x80B4**	0x1	(B,4)
0x801C	0x1	(A,c)	**0x8050**	0x80B0	(A,0)	**0x8084**	0x80BC	(A,4)	**0x80B8**	0x8000	(B,8)
0x8020	0x80B0	(A,0)	**0x8054**	0x80BC	(A,4)	**0x8088**	0x80BC	(A,8)	**0x80BC**	0xff	(E,0)
0x8024	0x80A8	(A,4)	**0x8058**	0x80BC	(A,8)	**0x808C**	0x1	(A,c)	**0x80C0**	0xff	(E,4)
0x8028	0x80BC	(A,8)	**0x805C**	0x1	(A,c)	**0x8090**	0x80A0	(A,0)	**0x80C4**	0x8008	(E,8)
0x802C	0x1	(A,c)	**0x8060**	0x80B0	(A,0)	**0x8094**	0x80BC	(A,4)	**0x80C8**	0	(D,0)
0x8030	0x80B0	(A,0)	**0x8064**	0x80BC	(A,4)	**0x8098**	0x80BC	(A,8)	**0x80CC**	0x80B0	(D,4)

(b) Generated Pointer Constraint Model

KOB	Offset Constraint	Target Constraint
A: 0x10 B: 0x0c C: 0x08 D: 0x08 E: 0x0c	OC[A] = [0, 4, 8] OC[B] = [8] OC[C] = [4] OC[D] = [4] OC[E] = [8]	TC[(A,0)] = {(B, 0, 0.9), (C, 0, 0.1)} TC[(A,4)] = {(D, 0, 0.5),(E, 0, 0.5) } TC[(A,8)] = {(E, 0, 1)} TC[(B,8)] = {(A, 0, 1)} TC[(C,4)] = {(B, 0, 1)} TC[(E,8)] = {(A, 8, 1)} TC[(D,4)] = {(B, 0, 1)}

Figure 2: An example of pointer-constraint model: (a) the labeled memory image for an OS version; (b) is the pointer-constraint model inferred from the labeled memory image. The first column of (b) means the object type and the size.

all the instances of t have valid pointers at offset o. For example, we can learn that OC[A]=[0,4,8], since all instances of A in Figure 2(a) have pointer-like values at offset 0, 4 and 8.

Target Constraints: A target constraint is imposed on the target of a pointer field. It includes the target type and the probability indicating how likely the pointer target is of the particular type. The target constraint is also can be learned through statistical analysis. By iterating through all labeled pointer fields and their targets in the training memory dumps, we can compute these target constraints. For example, the statistical analysis on instances of object A in Fig. 2(a) learns that the pointer at offset 0 in object A has two target labels, $(B, 0)$ and $(C, 0)$. By counting the numbers of instances of A with different target types, we also compute the probabilities of $(B, 0)$ and $(C, 0)$ to be 0.1 and 0.9 respectively. The target constraint ($TC(A, 0)$) is shown in Fig. 2(b).

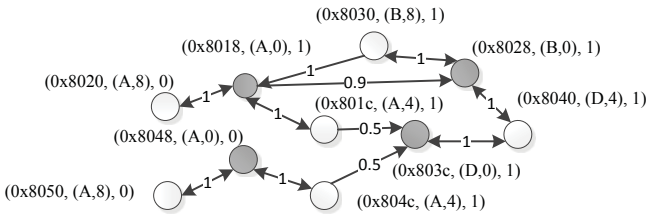
Variable-Length Arrays. The variable-length array is handled in the different manner, since its size is not a constant. We discover variable-length arrays using two conditions. (1) the size of a variable-length array is variable; (2) each entry of the array should have the same target type or a NULL pointer. This means that we only focus on object types with the variable size. For each object type with the variable size, we can determine the variable-length array by checking whether all entries of its instances share the same target type or zero. The arrays in our model will be labeled as "array" without the specific size. Its target offset constraint will be normalized to be relative to the start address of their hosting elements, instead of the base of the array.

4. KERNEL OBJECT IDENTIFICATION

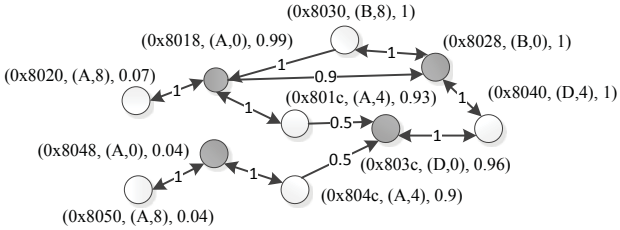
Given an arbitrary memory image, MACE tends to identify its kernel objects based learned constraint model from the training phase in the following steps: (1) it constructs a pointer-constraint graph from the memory image; (2) it applies the Random Surfer algorithm on the pointer-constraint graph until a convergence is reached; and (3) it selects true kernel objects based on the final scores on the pointer-constraint graph.

Address	Value	Address	Value
0x8018	0x8028	0x8038	0x4
0x801c	0x803c	0x803c	0x7
0x8020	0x803c	0x8040	0x8028
0x8024	0x1	0x8044	0x8
0x8028	0x11	0x8048	0x8034
0x802c	0xc	0x804c	0x803c
0x8030	0x8018	0x8050	0x8034
0x8034	0x6	0x8054	0x1

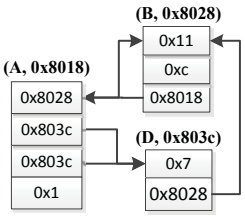
(a) Unlabeled memory image



(b) Initial constraint graph



(c) Converged constraint graph



(d) Identified kernel object graph

Figure 3: An example for random pointer surfing. A solid node in the graph represents a pointer with offset 0, indicating the base of a kernel object.

4.1 Pointer-Constraint Graph Construction

DEFINITION 1. *The pointer-constraint graph is a directed graph $G = (V, E)$. A node $v \in V$ is a tuple $(a, (t, o), r)$, where a is the address of a pointer, (t, o) labels the pointer as the object type t and the offset o with the object, and $r \in [0, 1]$ is the score indicating how likely this pointer is labeled correctly. An edge $e \in E$, $e = (u, v, r)$ represents the constraint from node u to node v, where $r \in [0, 1]$ specifies the conditional probability how likely v being correctly labeled if u is correct.*

As an example, we show a pointer-constraint graph in Figure 3(b), which is constructed from an unlabeled memory image in Figure 3(a), using the model presented in Figure 2. Algorithm 1 describes how to construct a pointer-constraint graph.

We construct a pointer-constraint graph, starting with a number of "root" nodes, and then perform breadth-first traversal to add new nodes and edges into the graph. To find root nodes, we select kernel objects (e.g., `EPROCESS` and `ETHREAD`) that have many pointer fields inside and thus their offset constraints are fairly unique. We use these offset constraints to scan the kernel memory and find possible kernel objects. Then a root node is created for each pointer field of these objects. Some of these root nodes may be in fact wrong. Their scores may be updated during the evaluation of their target constraints. At last, we will rely on the random surfer algorithm (described in Section 4.2) to evaluate their authenticity. For example in Figure 3, we choose object A to find "root" nodes, since it has the most number of offset constraints. If we scan the memory image using A's offset constraint, we find two instances for A. Therefore, we create root nodes for object A. MACE will create a node for each offset constraint of object A. We assign 1 to node $(0x8018, (A, 0), 1)$, $(0x801c, (A, 4), 1)$, $(0x8020, (A, 8), 1)$, $(0x8048, (A, 0), 1)$, $(0x804c, (A, 4), 1)$ and $(0x8050, (A, 8), 1)$ because at this stage, their offset constraints are all met.

To further expand the constraint graph, we need a working queue Q to perform this breadth-first traversal. To begin with, the root nodes are enqueued into Q. Then, on each iteration, a node v is dequeued from Q. We retrieve from the model PCM the target constraints TC for this node v. If v does have target constraints,

Algorithm 1: Pointer-Constraint Graph Construction

```
Input: Memory image M, Pointer-Constraint Model PCM
Output: Pointer-Constraint Graph G
Q ← G.V;
while Q ≠ ∅ do
    v ← Q.dequeue();
    TC ← PCM.GetTC(v.t, v.o));
    if TC ≠ ∅ then
        Matched ← False;
        for each tc ∈ TC do
            if PCM.CheckOC(M, M[v.a] − tc.o, tc.t) = True
            then
                Matched ← True;
                u ← (M[v.a], tc.t, tc.o, 1);
                if u ∉ G.V then
                    G.AddNode(u);
                    Q.enqueue(u);
                end
                G.AddEdge(v, u, tc.r);
                AddObjtoG(G, M[v.a] − tc.o, tc.t);
            end
        end
        if Matched = False then
            v.r ← 0;
        end
    end
end
return G;
```

we go over each target constraint tc in TC. tc tells us a possible target label $(tc.t, tc.o)$ and its likelihood $tc.r$. Then, to check if this target label is compatible with the target memory, we check the offset constraints of the target type $tc.t$ and the memory region starting at the corresponding object base address $M[v.a] - tc.o$. This is done in $PCM.CheckOC()$ function. In other words, we check if the memory words at the offsets specified in the offset constraint are valid addresses. If this is not true, this target type can be not valid, so we check the next target constraint in TC. Otherwise, we extend G into the target object.

To extend the graph G, we first check if the target node u (with the same address and label) has been already created. If not, we validate that its pointer locations are compatible with its offset constraints. For all the nodes that pass the validation check, we create u and set its initial score to 1. Otherwise we will set the score to be 0. We also enqueue u to Q for the subsequent breadth-first traversal. An edge $(v, u, tc.r)$ is added into G, where the edge's likelihood is obtained from the target constraint tc. For example, object A candidate fails the target checking at $0x8020$, $0x8048$ and $0x8050$, MACE has to update the value 1 to 0 for the node $(0x8020, (A, 8), 0)$, $(0x8048, (A, 0), 0)$ and $(0x8050, (A, 8), 0)$ in Figure 3(b).

Now we need to add the rest of pointers (if any) in the target object into G. To facilitate subsequent object-level classification, we always create a "base" node (whose offset is 0) for each object, even if the field at offset 0 is not a pointer. We further use this "base" node to bind all the pointers in that object by adding both incoming and outgoing edges between the "base" node. In this way, the scores on the pointers within one objects can flow back and forth to each other until a convergence is reached. In Figure 3, these "base" nodes are marked as solid circles.

If it turns out that none of the labels in target constraints of v is compatible with the target memory region, we set its score $v.r$ to 0 because v's label $(v.t, v.o)$ may be wrong. It is worth noting that we do not conclude that v is absolutely wrong and remove it immediately from G. Note that a pointer in a true object may occasionally point to invalid target (or a new target that does not exist in our model). We leave it to the Random Surfer algorithm below to decide if this pointer is indeed labeled wrong.

4.2 Random Surfer Algorithm

We adopt the random surfer algorithm in [12] to find nearly optimal solution on the pointer-constraint graph. It is also proved to converge. Suppose $\mathbf{r}$ is a $|v|$-dimensional column vector called score vector, where $|v|$ is the number of nodes in G. $\mathbf{r}_i$ is the score of the ith node in G (for the convenience we represent vectors and matrices in bold). Besides, we define a transition matrix $\mathbf{M}$, where $\mathbf{M}_{ij}$ is the transition probability from the ith node to the jth node in G. If there is no transition from i to j, $\mathbf{M}_{ij}$ is assigned 0. Because the matrix M is a stochastic matrix we normalize $\mathbf{M}$, such that each row of $\mathbf{M}$ sums to 1.

Equation 1 describes how to calculate the scores based on the neighbors' scores:

$$\mathbf{r}^{(k+1)} = (1 - \alpha - \beta)\mathbf{M}^T\mathbf{r}^{(k)} + \alpha\mathbf{p} + \beta\mathbf{r}^{(0)} \quad (1)$$

where $\mathbf{p} = [\frac{\sum_i^{|v|} r_i}{|v|}]_{|v|\times 1}$ is a constant score vector, where $|v|$ is the number of node in G, and $\mathbf{r}^{(k)}$ indicates the score vector at iteration k. α is the damping factor used to jump out of isolated loops or clusters during surfing. In order to ensure that $\mathbf{r}^{(k)}$ finally converge, β is introduced as another damping factor that controls the frequency jumping to the initial score distribution $\mathbf{r}^{(0)}$. Empirically, following [12] we set $\alpha = 0.7$, $\beta = 0.1$ to guarantee a good rate of convergence.

Algorithm 2 details how we update the score of each node in the constraint graph. For each iteration, the algorithm updates the score of each node based on scores of its neighbors and the constraints among them. After several iterations, the score of each node in $\mathbf{r}$ stabilizes. For each iteration, we calculate the mean square error between the current score vector and the previous one. If the error is smaller than the threshold ϵ, we consider it to have converged. The final score vector will approximate a globally optimal solution that satisfies the constraint graph.

Algorithm 2: Random Surfer algorithm

input : the transition matrix $\mathbf{M}$, the initial value vector $\mathbf{d}$, damping factor α, β and the vector constant $\mathbf{p}$
output: the converged score vector $\mathbf{r}$

$\mathbf{r}^{(0)} = \mathbf{d}$;
while $\delta \geq \epsilon$ **do**
  $\mathbf{r}^{(k+1)} = (1 - \alpha - \beta)\mathbf{M}^T\mathbf{r}^{(k)} + \alpha\mathbf{p} + \beta\mathbf{d}$;
  $\delta = ||\mathbf{r}^{(k+1)} - \mathbf{r}^{(k)}||_2^2$;
end

Figure 3(c) shows the converged pointer-constraint graph after we applied Algorithm 2 on the graph in Figure 3(b). Although the object A at $0x8018$ fails the target checking at offset 8, its score is 0.99. The overall voting through the constraint graph still considers object A at $0x8018$ is more likely to be true. Although the offset constraints for object A at $0x8048$ show it could be A, the converged constraint graph tells us the score for A at $0x8048$ is 0.04. Therefore, $0x8048$ is impossible to be the base address for object A. This observation verifies that the label decision of an object type judges on overall situation rather the individual pointer constraint.

4.3 Kernel Object Labeling

The kernel object labeling utilizes k-means [31] method where $k = 2$ in our scenario to cluster base nodes in the constraint graph. $k = 2$ means that we only split labeled nodes of the same type into two sets including the true set and false set. The set with higher scores as true set means that all nodes in the set are correctly identified. In detail, the kernel object labeling clusters base nodes of same object type by their scores and generates the identified kernel object graph from the cluster with the higher score. For example, k-means splits (0x8018,(A,0),0.99) and (0x8048,(A,0),0.04) into two sets and we considers (0x8018,(A,0),0.99) as the true set. As for the example in Figure 3(d), we are able to classify the base nodes in the converged graph, and construct a kernel object graph. From the result, we can see that MACE can still find object A at $0x8018$, even if the constraint checking for the offset 8 at object A failed.

5. IMPLEMENTATION AND EVALUATION

The implementation of MACE comprises 3 components. One component performs an initial scan to recognize pointers on a memory image, which include one plugin (with 570 lines of Python code) to Volatility and one (with 78 lines of c code) to DECAF [17]. Another component is the plugin of DECAF with additional 800 lines of C code to gather the ground truth for kernel objects. The third component is a stand-alone Python program consisting of 6.2K LOC used for learning the pointer-constraint model and kernel object identification.

We evaluated MACE from the following aspects: Section 5.1 presents the model generation results, including how fast the model converges, how big the model is, and how long it takes to generate the model; Section 5.2 measures the accuracy and runtime performance of kernel object identification; Section 5.3 demonstrates MACE's capability of detecting rootkit footprints using realworld rootkit samples; Finally, Section 5.4 presents synthetic attacks demonstrating the attack tolerance of MACE over the other memory analysis tools.

Experiment Setup. We evaluate MACE on four sets of memory images: 1) 150 memory images from Windows XP Service Pack 3 including 100 images for training and 50 for detection (All these images are of 512 MB RAM); 2) 145 memory images from Windows 7 Service Pack 0 including 100 images for training and 45

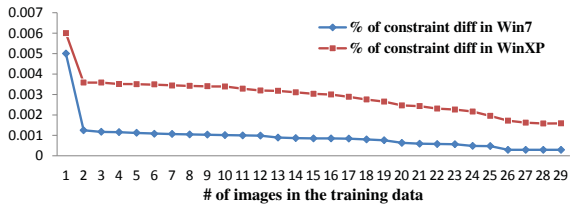


Figure 4: Changes in the model quality δ (number of new target constraints + new offset constraints) across images. The small number of images can achieve a stable model.

for detection (All these images are of 1.5 GB RAM); 3) 8 memory images of 512 MB RAM from kernel malware analysis; and 4) 1 memory image of 1.5 GB RAM for synthetic attack. The first 295 memory images are derived by using our dynamic analysis component (as discussed in Section 3.1). All experiments were conducted on a machine with Intel(R) Xeon(R) CPU E5-2650 ($2.00GHz \times 8$) and 128 GB RAM running 64-bit Ubuntu 11.04.

5.1 Model Generation

Model Convergence. We generate a model from 100 memory images for Windows XP and Windows 7 respectively. In order to predict how close this model is to a theoretically perfect model, we randomly select k images to generate another model, and compute a "diff" between these two models with respect to their offset constraints and target constraints. In this way, we can see how quickly the model converges when number of training images increases.

Figure 4 shows the model quality evaluation results for both Windows XP and Windows 7. It illustrates that as the number of images in the training set increases, the missing constraints of the model generated from k images (compared to the model generated from 100 images) decreases exponentially and becomes stable very quickly. Even a model generated from one image is 99.4% similar to the model generated from 100 images.

Model Generation Runtime. Obtaining a labeled memory image took nearly 2 minutes for Windows XP and 3 minutes for Window 7. To speed up the image retrieving process, we run 10 virtual machines in parallel and each one conducts 10 different test cases. Finally, it takes 20 minutes on average to obtain 100 Windows XP images and 30 minutes on 100 Windows 7 images. For each image, the model generator extracts the pointer information from the memory image. It takes approximately 7 minutes per memory image for Windows XP and 15 minutes for Windows 7. This task can be finished within 40 and 80 minutes respectively for XP and Windows 7 by using 20 processors in parallel. In the end, it takes 20 minutes and 30 minutes to merge the results on 100 images and construct the pointer-constraint model for Windows XP and Windows 7 respectively. Overall, the model generation takes less than 2 and 3 hours for Windows XP and Windows 7 respectively.

5.2 Kernel Object Identification

We evaluate MACE's identification capabilities with respect to the accuracy and the runtime performance.

Accuracy. We use the two metrics, *Recall* and *Precision*, to measure the accuracy of the detection results. We calculate the recall and the precision using the following formulas:

$$Recall = \frac{Correctly\ labeled\ bytes}{Total\ bytes\ labelled\ in\ ground\ truth} \quad (2)$$

$$Precision = \frac{Correctly\ labeled\ bytes}{Total\ bytes\ labelled\ by\ MACE} \quad (3)$$

Steps	Time (Sec)	
	Windows XP	Windows 7
Initial Scan	3.0 ± 2.1	7.0 ± 4.9
Graph Generation	180 ± 2.0	315 ± 5.6
Kernel Object Inference	22 ± 0.7	55 ± 1.6
TOTAL	205 ± 4.5	377 ± 12.6

Table 1: MACE's Identification Runtime Performance

We measured the above two metrics over 45 memory images for Windows XP and 7 respectively. Figure 5 shows MACE can achieve good identification results for both Windows XP SP3 and Windows 7 SP0. More specifically, MACE achieves the 95% recall on average and 98% precision on average for Windows XP SP3, and the 96% recall and 95% precision on average for Windows 7 SP0. The detection result of MACE is close to KOP [4] which relies on the source code. Furthermore, we observed zero false negatives and false positives in the kernel objects of high forensic values (the ones extracted by Volatility). 5% false negatives are from the undocumented objects, such as the objects with pool tags 'IoNm' and 'GH0<'. 2% false positives are caused by the undocumented kernel objects of small sizes, such as 'Mmpv'.

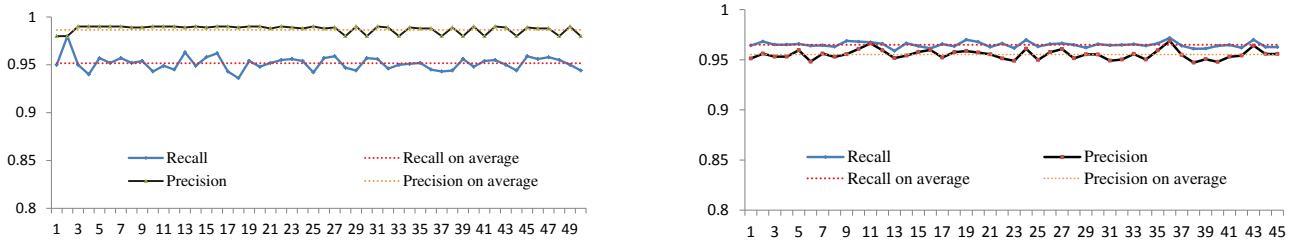
Runtime Performance. We evaluated MACE's identification runtime performance in two scenarios. In the cloud computing scenario, the virtual machine state has been loaded in memory, so scanning through the virtual machine memory is very fast. We used virtual machine snapshots (50 Windows XP SP3 memory images with 512 MB RAM and 45 Windows 7 SP0 images with 1.5 GB RAM) from KVM/QEMU to evaluate this scenario. On the contrary, in the memory forensics scenario, the memory content is first dumped into a file, then the forensic analysis is performed on the file. The analysis for this scenario will be slower, due to the time for loading the file into memory and other factors. The result for the first scenario is shown in Table 5.2.

As we see, MACE finished the kernel object identification in 205 seconds for Windows XP and 377 seconds for Windows 7 on average. The identification for Windows 7 takes longer, as the memory images for Windows 7 are larger and contain more kernel objects. Note that our current implementation is in Python mainly for fast prototyping. The identification performance can be significantly reduced to tens of seconds for a C/C++ implementation.

5.3 Detecting Kernel Rootkit Footprints

As a case study, we show how to use the kernel object graph constructed by MACE to detect kernel rootkit footprints. To this end, we developed a tool to analyze memory images infected with kernel rootkits. Using the kernel object graph constructed by MACE, the tool can detect malicious function pointers and hidden objects in the infected images. For the sake of fair comparison, this tool follows the similar logic as SFPD and GHOST that were built on top of KOP [4]. Note that KOP requires the source code of Microsoft Windows to construct the extended type graph and traverse the kernel objects, whereas MACE does not. From the viewpoint of external memory analysts, we would like to see whether the tool built on top of MACE can reach the same detection performance as the ones built on KOP.

More specifically, to detect malicious function pointers, our tool iterates through all the kernel objects and examine the function pointers in them. MACE can differentiate function pointers from data pointers, because the target of a function pointer must be located in the text section of a kernel module, which can be determined from parsing the headers of that module. As the pointer-constraint model contains all the valid targets for each function



(a) Windows XP SP3

(b) Windows 7 SP0

Figure 5: Precision and Recall.

pointer during the training phase, to determine a malicious function pointer, we can check whether the actual target of this function pointer does not belong to any of the valid targets. To be consistent to SFPD, our tool also excludes manipulations in System Service Dispatch Tables (SSDTs) and Interrupt Descriptor Table (IDT).

To detect hidden objects, we use Volatility as the reference system. In particular, we use common commands in Volatility, such as `pslist`. Then we compare these objects obtained from Volatility and the kernel object graph from MACE. If a kernel object of one of the above types appears only in the result from MACE, it is deemed a hidden object.

We collected 8 memory images infected with various real world kernel rootkits. Two images (Stuxnet.vmem and ds_fuzz_hidden _proc.img) were downloaded from the Volatility Google Code website. The rest of memory images (including TDSS, Fakeuinit, ZeroAccess, Papras, Haxdoor, and FuTo) were recorded by running these samples separately in a virtual machine. It demonstrated that MACE can tolerate small changes in the kernel code and the locations of global pointers, making it practical to analyze realworld memory images. We list rootkit detection results in Table 5.2.

Malicious Function Pointers. It is not surprising that our tool built on MACE can detect malicious function pointers in the common data structures like `_DRIVER_OBJECT`, etc. The other rootkit detection tools would have the same coverage, assuming that they can identify these data structures correctly. More interesting results are found for TDSS and Fakeuinit, and they are highlighted in the table. For TDSS, we found two malicious function pointers located in the data section of "ntoskrnl.exe". With help of IDA Pro [18], we confirmed that tampering with these two function pointers can effectively hook `IofCompleteRequest` and `IofCallDriver` and thus manipulate the communication between the main kernel and the device drivers. As for Fakeuinit, we found 5 malicious function pointers in undocumented kernel objects, whose pool tags are 'NDpp', 'NDpb and 'NDmo' respectively. Through manual investigation, we determined that these kernel objects are operated by `NDIS.sys` (the central networking module in Windows) to manage the network stack. Manipulating these function pointers can effectively intercept the network communication. Here, note that these pool tags identified from our model are not from the infected memory images. So even if the actual pool tags are modified by the rootkits, our detection results would stay unaffected.

Hidden Objects. Our tool also detected hidden processes and modules for several kernel rootkits. It shows that instead of just examining several known linked lists and tables, MACE discovers kernel objects in a global scope. We notice that a recently developed tool (`psxview`) in Volatility can detect hidden processes, by checking other data structures in addition to the active process linked list. In comparison, our tool checks `_EPROCESS` objects in the entire kernel data structure graph, not only the ones publicly known to the memory analysts, not to mention that our tool can also detect other kinds of hidden objects.

5.4 Attack Tolerance

To evaluate the attack tolerance of MACE, We devised two synthetic attacks: 1) pool tag manipulation; and 2) deterministic pointer removal.

Pool Tag Manipulation. This synthetic attack is as simple as modifying pool tags for the objects like `_KDBG`, `_EPROCESS`, etc. After these modifications, the Windows system continued to run properly, indicating that there is no integrity check on pool tags in Windows. Then we tested the commands in volatility, and none of them output any results. The commands like `psscan` and `thrdscan` rely on pool tag as a constraint to scan particular kinds of kernel objects, so simple modifications on pool tags can easily sabotage these commands. Other commands like `pslist` and `threads` also failed, even though they did not use pool tag as a constraint extensively to scan kernel objects. The failure of these commands is due to the missing `_KDBG`. These commands must scan `_KDBG` to determine the right Windows version and thus locate the right start address of the relevant data structures. For the same reason, the new `psxview` command also failed. In contrast, MACE was not affected by this synthetic attack at all, because by design MACE does not use pool tags and other kinds of soft constraints to identify kernel objects.

Deterministic Pointer Removal. We suppose that an attacker can manage to remove a fraction of pointers to hide certain kernel objects, without causing a system crash. In particular, we would like to see how MACE's identification performance degrades while a fraction of deterministic pointers are removed, because all the existing tools only examine deterministic pointers. Furthermore, we simulate a "strawman" system as the theoretical upper-bound for any memory analysis system that only examines deterministic pointers. This "strawman" system starts with global variables in the data sections of kernel modules, and only follows deterministic pointers and always makes a right decision whether an object is valid even when some of its pointers or pointer targets are invalid.

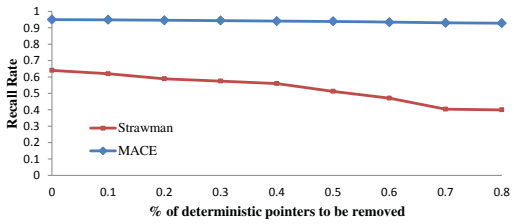


Figure 6: Recall Degradation on Link Sabotage Attacks

Name	Malicious Location	#	Cat.
Backdoor: W32TDSS	**ntoskrnl.exe:0x7c484**	1	M
	ntoskrnl.exe:0x7c480	1	M
	_GENERIC_CALLBACK.Callback	2	M
	_DRIVER_OBJECT.DriverStart	1	M
	_DRIVER_OBJECT.DriverInit	1	M
	_LDR_DATA_TABLE_ENTRY.EntryPoint	1	M
	_LDR_DATA_TABLE_ENTRY	1	H
stuxnet.vmem	_GENERIC_CALLBACK.Callback	1	M
	_NOTIFICATION_PACKET. NotificationRoutine	1	M
	_DRIVER_OBJECT.DriverStart	1	M
	_DRIVER_OBJECT.DriverInit	1	M
	_DRIVER_OBJECT.MajorFunction[]	3	M
	_LDR_DATA_TABLE_ENTRY.EntryPoint	1	M
Trojan-Spy.Win32. Fakeuinit.a	**NDpp:0x18**	1	M
	NDmo:0x38	1	M
	NDmo:0x50	1	M
	NDmo:0x40	1	M
	NDpb:0x4c	1	M
	_DRIVER_OBJECT.DriverInit	1	M
	_DRIVER_OBJECT.DriverUnload	1	M
	_DRIVER_OBJECT.DriverStart	1	M
	_ETHREAD.StartAddress	1	M
	_LDR_DATA_TABLE_ENTRY.EntryPoint	1	M
Backdoor. Win32. ZAccess.dl	_DRIVER_OBJECT.DriverInit	1	M
	_GENERIC_CALLBACK.Callback	1	M
	_ETHREAD.StartAddress	1	M
TrojanPSW. Win32.Papras	_DRIVER_OBJECT.DriverInit	1	M
	_DRIVER_OBJECT.DriverUnload	1	M
	_DRIVER_OBJECT.DriverStart	1	M
	_LDR_DATA_TABLE_ENTRY.EntryPoint	1	M
ds_fuzz_ hid den_proc.img	_EPROCESS	7	H
	_DRIVER_OBJECT.DriverUnload	1	M
	_DRIVER_OBJECT.DriverStart	1	M
	_DRIVER_OBJECT.MajorFunction[]	4	M
	_LDR_DATA_TABLE_ENTRY.EntryPoint	1	M
Win32. Haxdoor	_DRIVER_OBJECT.DriverStart	1	M
	_DRIVER_OBJECT.DriverInit	1	M
	_DRIVER_OBJECT.MajorFunction[]	2	M
	_LDR_DATA_TABLE_ENTRY.EntryPoint	1	M
RootKit: Futo	_DRIVER_OBJECT.DriverStart	1	M
	_DRIVER_OBJECT.DriverInit	1	M
	_DRIVER_OBJECT.DriverUnload	1	M
	_DRIVER_OBJECT.MajorFunction[]	4	M
	_LDR_DATA_TABLE_ENTRY.EntryPoint	1	M
	_EPROCESS	1	H

Table 2: Rootkit Footprints Detected By MACE**. In the column of "Category", M means "malicious function pointer", and H stands for "hidden object".**

To evaluate this attack, we randomly remove a fraction of deterministic links (e.g, 10%, 20%, 30%, etc.) from a labeled memory image, and compute the recall for both MACE and the strawman system. For the strawman system, we consider it would miss a kernel object if there is no deterministic path from a kernel module to it. Figure 6 presents this result for a Windows XP image. It shows how the recall degrades more or less with the increase of the percentage of sabotaged deterministic pointers.

We can see that even when the attack is absent (0% pointers are removed), the recall for the strawman system is only 65%, demonstrating the necessity of incorporating non-deterministic pointers into the analysis. The performance of the strawman system degrades to about 40% when 80% deterministic pointers are removed. This result appears to be reasonable. However, this is just a theoretical upper-bound. The real memory analysis systems that only follow deterministic pointers will certainly perform worse than it. Further, the strawman system failed to identify many important kernel objects. For example, out of 22 process objects, it missed 20.

In contrast, the recall degradation for MACE is barely noticeable even when 80% deterministic pointers are removed, thanks to the small-world effect of the kernel object graph and the global evaluation nature of random surfer model.

6. DISCUSSION

In this section, we discuss several potential and practical issues and concerns related to MACE. Also, we make clarifications and suggest countermeasures if necessary.

Kernel Patches. By design, we need to train one model for each OS version. Kernel patches introduce changes in the main kernel module and possibly certain data structure definitions. So we would need to train a new model for every single kernel patch. In reality, it is not necessary, because the changes introduced in these patches are usually small. The major kernel data structure definitions remain unchanged. As demonstrated in our experiment, we can still use the model generated for Windows XP Service Pack 3 to analyze two memory images downloaded from the Volatility website and obtain good results. In these two memory images, patches have been applied to Windows XP Service Pack 3.

Third-party Device Drivers. A memory image under the analysis may have third-party device drivers loaded, which have not been observed during model generation phase. In this case, MACE will not be able to identify kernel objects defined in these device drivers. However, MACE will still detect these device drivers, because a number of documented objects (e.g., `DEVICE_OBJECT`) will be created for them and they will be detected by MACE. It is still an open research problem to discover data structures in these third-party device drivers. We leave it as future work.

7. RELATED WORK

Memory Analysis Frameworks. Memory analysis came into limelight after 2004 work by Carrier et al. [5]. There exist plenty of open-source and commodity memory analysis tools [2, 6, 9, 10, 19, 23–25, 27, 28, 30, 32].The memory analysis is also extended to the analysis of hypervisors and virtual machine [15].

Robust Signature Schemes. To improve robustness for memory analysis [26], two signature schemes [11, 21] have been proposed. These two signature schemes detect kernel objects by relying on invariants that are hard to be manipulated and evaded. Although the work by Dolan-Gavitt et al. [11] is based on data invariants, most of the identified invariants are indeed on pointer fields. In comparison, MACE leverages the insights in pointer invariants and takes a fresh look into the problem of memory analysis.

Source Code based Memory Analysis. The knowledge of data structure definitions can be directly obtained from the kernel source code. SigGraph [21] extracts points-to relationships directly from the Linux source code and creates pointer-based signatures. SigGraph only extracts deterministic points-to relationships, and omits generic pointers. In order to obtain a nearly complete data structure graph, KOP [4] and MAS [8] perform points-to analysis on the Windows kernel source code. They identify points-to relationships for generic pointers and generate extended type graph. In contrast, MACE is designed for external forensic analysts who often do not have access to the kernel source code of an investigated system.

Probabilistic Memory Analysis. Several systems also take probabilistic approaches in memory analysis. Laika [7] applies Bayesian unsupervised machine learning algorithm to infer a type graph from a memory snapshot of a user-level program execution. In comparison, the inference algorithm used in MACE is supervised learning. In the training set, the kernel objects are classified and labeled using dynamic analysis. Some assumptions made in Laika do not hold in kernel data structures. For example, Laika assumes a pointer should point to the beginning of an object. This is not true for kernel data structures in common OSes like Windows and Linux.

To identify data structure instances that have been freed, DIMSUM [20] takes a probabilistic inference approach. Given a data structure definition, DIMSUM constructs a factor graph and computes marginal probabilities of all the candidate memory locations that satisfy the data structure constraints. In comparison, MACE tackles a similar problem but in a larger scale. The computational overhead would be too high to compute marginal probabilities for all pointers.

8. CONCLUSION

In this paper, we presented MACE, a memory kernel object mining tool that can accurately identify kernel objects in a robust manner. We evaluated MACE on 100 memory images for Windows XP SP3 and Windows 7 SP0. The experimental results showed that MACE can achieve the recall of 95% and the precision of 98% on average. Furthermore, the experiment also demonstrates the the robustness and the good efficiency. To illustrate the strength of MACE, we also conducted synthetic attacks on a memory image from Window XP SP3. The detection result showed that MACE outperformed other external memory analysis tools with respect to wider coverage and better robustness.

9. ACKNOWLEDGEMENT

We would like to thank anonymous reviewers for their comments, and insightful discussions with Dr.Weidong Cui. This research was supported by NSF grants #1018217 and #1054605, AFO SR grant #FA9550-14-1-0119, McAfee Inc, and VMware Inc. Any opinion, findings, conclusions, or recommendation are those of the authors and not necessarily of the funding agencies.

References

[1] LMBench – Tools for Performance Analysis. `http://www.bitmover.com/lmbench`.

[2] N. Beebe. Digital forensic research: The good, the bad and the unaddressed. In *Advances in Digital Forensics V*. 2009.

[3] A. Bianchi, Y. Shoshitaishvili, C. Kruegel, and G. Vigna. Blacksheep: Detecting compromised hosts in homogeneous crowds. In *Proceedings of the 2012 ACM conference on Computer and communications security (CCS'12)*, 2012.

[4] M. Carbone, W. Cui, L. Lu, W. Lee, M. Peinado, and X. Jiang. Mapping kernel objects to enable systematic integrity checking. In *Proceedings of the 16th ACM Conference on Computer and Communication Security (CCS'09)*, 2009.

[5] B. D. Carrier and J. Grand. A hardware-based memory acquisition procedure for digital investigations. *Digital Investigation*, 1(1):50–60, 2004.

[6] A. Case, L. Marziale, and G. G. Richard, III. Dynamic recreation of kernel data structures for live forensics. *Digital Investigation*, 7:S32–S40, 2010.

[7] A. Cozzie, F. Stratton, H. Xue, and S. T. King. Digging for data structures. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI'08)*, 2008.

[8] W. Cui, M. Peinado, Z. Xu, and E. Chan. Tracking rootkit footprints with a practical memory analysis system. In *Proceedings of USENIX Security Symposium*, 2012.

[9] B. Dolan-Gavitt. The vad tree: A process-eye view of physical memory. *Digital Investigation*, 4:62–64, 2007.

[10] B. Dolan-Gavitt. Forensic analysis of the windows registry in memory. *Digital Investigation*, 5:S26–S32, 2008.

[11] B. Dolan-Gavitt, A. Srivastava, P. Traynor, and J. Giffin. Robust signatures for kernel data structures. In *Proceedings of the 16th ACM Conference on Computer and Communications Security(CCS'09)*, 2009.

[12] A. Farahat, T. LoFaro, J. C. Miller, G. Rae, and L. A. Ward. Authority rankings from hits, pagerank, and salsa: Existence, uniqueness, and effect of initialization. *SIAM Journal on Scientific Computing*, 27(4):1181–1201, 2006.

[13] Q. Feng, A. Prakash, H. Yin, and Z. Lin. Mace: High-coverage and robust memory analysis for commodity operating systems. Technical Report SYR-EECS-2014-05, Syracuse University, 2014.

[14] FU Rootkit. `http://www.rootkit.com~/project.php?id=12`, 2005.

[15] M. Graziano, A. Lanzi, and D. Balzarotti. Hypervisor memory forensics. In *Proceedings of Symposium on Research in Attacks, Intrusion, and Defenses (RAID'13)*, 2013.

[16] T. H. Haveliwala. Topic-sensitive pagerank: A context-sensitive ranking algorithm for web search. *IEEE Transactions on Knowledge and Data Engineering*, 15(4):784–796, 2003.

[17] A. Henderson, A. Prakash, L. K. Yan, et al. make it work, make it right, make it fast. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA'14)*, 2014.

[18] The IDA Pro Disassembler and Debugger. `http://www.datarescue.com/idabase/`.

[19] J. D. Kornblum. Using every part of the buffalo in windows memory analysis. *Digital Investigation*, 4(1):24–29, 2007.

[20] Z. Lin, J. Rhee, C. Wu, X. Zhang, and D. Xu. Dimsum: Discovering semantic data of interest from un-mappable memory with confidence. In *Proceedings of the 19th ISOC Network and Distributed System Security Symposium (NDSS'12)*, 2012.

[21] Z. Lin, J. Rhee, X. Zhang, D. Xu, and X. Jiang. Siggraph: Brute force scanning of kernel data structure instances using graph-based signatures. In *Proceedings of the 18th Annual Network and Distributed System Security Symposium (NDSS'11)*, 2011.

[22] Z. Lin, X. Zhang, and D. Xu. Automatic reverse engineering of data structures from binary execution. In *Proceedings of the 17th Annual Network and Distributed System Security Symposium (NDSS'10)*, 2010.

[23] MANDIANT Memoryze. `http://www.mandiant.com/resources/download/memoryze`.

[24] S. Mrdovic, A. Huseinovic, and E. Zajko. Combining static and live digital forensic analysis in virtual environment. In *Proceedings of XXII International Symposium on Information, Communication and Automation Technologies, 2009.*

[25] N. L. Petroni, Jr., A. Walters, T. Fraser, and W. A. Arbaugh. FATKit: A framework for the extraction and analysis of digital forensic data from volatile system memory. *Digital Investigation*, 3(4):197–210, 2006.

[26] A. Prakash, E. Venkataramani, H. Yin, and Z. Lin. Manipulating semantic values in kernel data structures: Attack assessments and implications. In *Proceedings of 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'13)*, 2013.

[27] A. Schuster. Searching for processes and threads in microsoft windows memory dumps. *Digital Investigation*, 3:10–16, 2006.

[28] A. Schuster. The impact of Microsoft Windows pool allocation strategies on memory forensics. *Digital Investigation*, 5:S58–S64, 2008.

[29] A. Slowinska, T. Stancescu, and H. Bos. Howard: a dynamic excavator for reverse engineering data structures. In *Proceedings of the 18th Annual Network and Distributed System Security Symposium (NDSS'11)*, 2011.

[30] Volatility: Memory Forencis System. `https://www.volatilesystems.com/default/volatility/`.

[31] K. Wagstaff, C. Cardie, S. Rogers, and S. Schrödl. Constrained k-means clustering with background knowledge. In *Proceedings of the Eighteenth International Conference on Machine Learning (ICML'01)*, 2001.

[32] R. Zhang, L. Wang, and S. Zhang. Windows memory analysis based on kpcr. In *Proceedings of the Fifth International Conference on Information Assurance and Security(IAS'09)*, 2009.

Assisted Deletion of Related Content

Hubert Ritzdorf Nikolaos Karapanos Srdjan Čapkun

Institute of Information Security
Department of Computer Science
ETH Zurich, Switzerland
firstname.lastname@inf.ethz.ch

ABSTRACT

On primary storage systems content is often replicated, converted or modified, and the users quickly lose control over its dispersal on the system. Deleting content related to a particular project from the system therefore becomes a labor-intensive task for the user. In this paper we present *IRCUS*, a system that assists the user in securely removing project-related content, but does not require changes to the user's behavior or to any of the system components, such as the file system, kernel or applications. IRCUS transparently integrates within the user's system, operates in user-space and stores the resulting metadata alongside the files. We implemented and evaluated our system and show that its overhead and accuracy are acceptable for practical use and deployment.

1. INTRODUCTION

Recent cases of information leakage as well as laws and court orders that compel citizens and companies to surrender their devices and data [6, 9, 26, 29, 34], have motivated the need for *secure data deletion*. When faced with a coercive adversary, who is able not only to get access to the devices or data, but can also force the user to disclose all access credentials (e.g., passwords), removing data from a system is the safest way to truly preserve its confidentiality.

However, when deleting content from a personal storage device, a typical user faces the following challenges. Content is replicated across different files, possibly in different formats, e.g., file copies or file containers, and the user has to manually identify all files which share pieces of sensitive content or are related in some other way (e.g., belonging to the same project). This procedure should allow the user to perform meaningful deletion without leaving any sensitive content inadvertently exposed (i.e., not deleted). This is, nevertheless, a tedious and complicated task on modern systems with large storage and complex file dependencies. Moreover, if the system has been running for a longer time, the user will likely not be aware or not even remember where certain files and content reside. In most cases, the user is left with the task of manually inspecting files and explicitly choosing which files to delete.

Previous work has studied secure deletion for complete devices, data blocks or files [2, 5, 7, 14, 33]. However, all of these approaches assume that the user is able to identify the files or data blocks that need to be deleted, as well as those that must be preserved. None of those approaches consider easy secure deletion for content that is replicated across different files. To identify files with shared content, previous work has studied information flow [8, 21], file provenance [20, 36] and causality between files [24]. However, these works were application [20, 43] or operating-system specific [36] and required modification of existing systems [13, 24, 25] or virtualization [16, 21].

In this paper, we introduce *IRCUS* (Identification of Related Content from User Space), a system designed to assist the user with file or project deletion, by proposing related files upon deletion. More precisely, the goal is to assist the user in identifying related files that belong to the same project, by leveraging common content, file co-location and similar access patterns. Once the user wants to delete a file, IRCUS presents all related files and the user decides which files to delete in order to delete the content. IRCUS requires no user actions prior to the file deletion, such as labeling. In accordance with the security principle of least privilege and to allow easy adoption in all types of local and remote storage, IRCUS is implemented in user space.

IRCUS is based on a generic approach for discovering project-related files, independent of operating system, storage type or specific applications. As IRCUS executes in user space, it can be installed as a user application and adapted to specific user requirements. Contrary to previous work, our work requires no modification to other applications (e.g., text processing applications) or the operating system. The system overhead is smaller than in most previous proposals. Low overhead is important, because secure deletion is an infrequent operation that should not interfere with normal system operation.

In summary, we make the following contributions.

- We propose a novel approach for the continuous identification of project-related files with the goal of assisting the user in securely deleting sensitive content.
- We design a generic architecture that does not depend on specific applications or systems. Moreover, it requires no modification to the existing applications or the underlying operating system and it can instead be executed as a user-space application.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.
ACSAC '14, December 08 - 12 2014, New Orleans, LA, USA
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-3005-3/14/12 ...$15.00
http://dx.doi.org/10.1145/2664243.2664287.

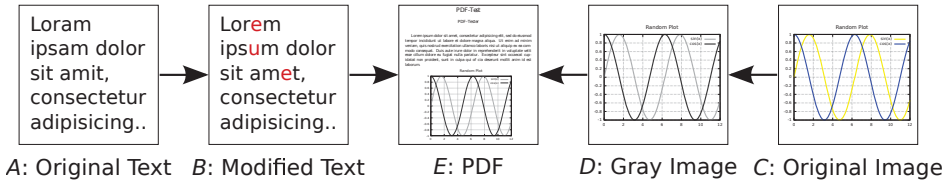


Figure 1: Example PDF Generation: As they share content, files A, B, C and D are related to file E. Securely Deleting just E does not securely delete its content.

- We create a prototype implementation of our system, explain our reasoning behind certain implementation decisions and conduct an experimental evaluation of our system in order to demonstrate its effectiveness as well as its performance.

The rest of the paper is structured as follows. In Section 2 we present the problem statement and the matching background in Section 3. In Section 4 we discuss the design principles of our approach and its features. We present our solution and its results in Sections 5 and 6. We discuss related work in Section 7 before concluding in Section 8.

2. PROBLEM STATEMENT

We consider the problem of secure deletion of project-related content. More specifically, our goal is to design a system which, based on observed file system events, learns which user files belong to the same "project" or more generally to the same context. Furthermore, we want our system to be easily integrated into existing platforms and work well across local and remote (e.g., cloud) storage. As a result, the system should assist a user during the file and project deletion processes by displaying related files and the types of their relationships.

Although this problem could be solved by relying on users to label files and their relationships, that approach would require a change in the user behavior and in the mode of system use, which is not realistic in most applications. Instead, our goal is to build a system that will track user behavior and file content and suggest file relationships to the user upon deletion, therefore reducing the user's workload.

On a modern system content is copied, slightly modified, converted or bundled up with other content, e.g., in archives. Common examples for shared content include text documents, presentations and attachments to e-mails. An example more specific to (computer science) research is the compilation of PDF files from text and image files, where the information ends up being contained both in the compiled output and in the source files. In this example, the deletion of the PDF file therefore does not remove any content from the system. In addition, users re-use texts, presentations and implement manual versioning, inadvertently leaving traces of various information within their local and remote systems [23]. After a longer operation of the system the users are therefore unaware of all the copies and relations between different files on the system and require assistance in identifying the files that contain project information.

Identifying files that belong to the same project and/or are a part of the broader context can only be fully done by the user. The notion of a common context is user and possibly organization-specific. No system can fully capture this notion unless it takes input from the user. Since we want to minimize if not fully remove user's involvement we focus on file relations that can be inferred by the system without explicit user input; these relationships can then be presented to the user, who can decide if they are truly meaningful. The relationships that we consider are primarily based on overlapping content, correlated access and file co-location within the file system.

Relying solely on content overlap misses high-level project connections such as in the case of projects that contain a project description and the implementation source code files that have little or no content overlap. We also miss connections with files that are encrypted or encoded in an unrecognized format, e.g., proprietary file formats. To discover these semantic connections we assume that files with related content will frequently be accessed together. This assumption is validated by our small user study in Section 3.3. We therefore classify files with strongly correlated access patterns as related to discover project-based relations.

In summary, our goal is to enable secure deletion of content by finding copies or different versions (through shared content) or files that are part of the same project (through shared access patterns or co-location). These files are presented to the user for deletion. Finally, only the user can decide which files should be securely deleted together to delete specific content.

3. BACKGROUND

Before presenting our proposed architecture, we provide an intuitive description on which files we consider to be "related", such that they have to be deleted together, in order for the secure deletion operation to have the desired result. Furthermore, we motivate the need for designing an efficient system, by showing that the naive approach to solving the given problem would lead to an impractical system due to poor performance. To understand file system operations generated by user actions, we also performed a small user study and present the derived conclusions.

3.1 Related Files

We consider two types of related files. First, files with overlapping content which we call *content-related*. Second, files that exhibit patterns of being accessed together which we call *access-related*.

The content which is shared between two content-related files, say, A and B might be considered sensitive by the user. Thus, when the user chooses to delete file A, our system should also propose file B to the user for deletion.

Content-related files can be explained with the example of PDF generation. Imagine that the user creates a confidential PDF file which contains text and an image. The components

are shown in Figure 1. The original text A was modified and saved under a name B, while the original image C was converted to grayscale and saved as D. Finally, the PDF E is then generated out of B and D.

If the user decides to securely delete the PDF, without deleting its components, the PDF can be easily restored. Therefore, files A, B, C and D are content-related to the PDF and should be deleted together with the derived PDF. We can also observe that files A and B are *similar* and are symmetrically content-related, while file B is *contained* in the PDF E.

Access-related files may not share any content at all, however, they can be semantically linked. As an example, consider a project description and a source code file. They are part of the same project and would be frequently accessed together. Linking them can reveal related files to the user. Additionally access relation can link files with overlapping content that is hard or impossible to identify from file system operations, such as encrypted content or proprietary file formats. To increase the accuracy of access relations, we take file co-location into account.

3.2 Naive Approach

One way to address the problem of identifying related files for deletion is the following naive approach. We can implement a system that simply searches for related files on demand, i.e., right before deleting a chosen file. In this reactive approach the identification of access-related files would become much harder, as only the file timestamps are available at that time. Additionally, all files would have to be checked for their timestamps, which would not be scalable.

The reactive approach would also require at least one search through the complete file system for content-related files. After the initial search, further searches might be necessary in order to build a relation graph as in Figure 1.

Using the same system as presented in Section 5 we implemented such a reactive search for content-related text documents. We found that most users had at least 128MB of text data, with an average file size of roughly 16KB. For such file sizes our system spent an average of 1.37 seconds per MB for comparison. Therefore, the reactive approach would take at least 175 seconds for most users, not including the time for file lookup, comparisons to other file types or additional searches to establish a relation graph.

Overall, the reactive approach would have an impractical runtime. Even after applying parallelization the waiting time would significantly impact the user experience. As this approach is not scalable, we conclude that it is not suited to solve the problem.

Alternatively, the naive approach could also be used in a proactive manner in order to eliminate the user waiting time before each deletion. However, this would require the aforementioned scan of all relevant file types to be performed after any change in the file system. This constant load would render the naive approach unusable.

3.3 User Study

To gain a better understanding of realistic file system behavior, we performed a small user study. As we record file system operations and their content, the study was very sensitive, so that we could only get four participants. Participants had the chance to remove sensitive entries from the recordings before sending the recordings back to us.

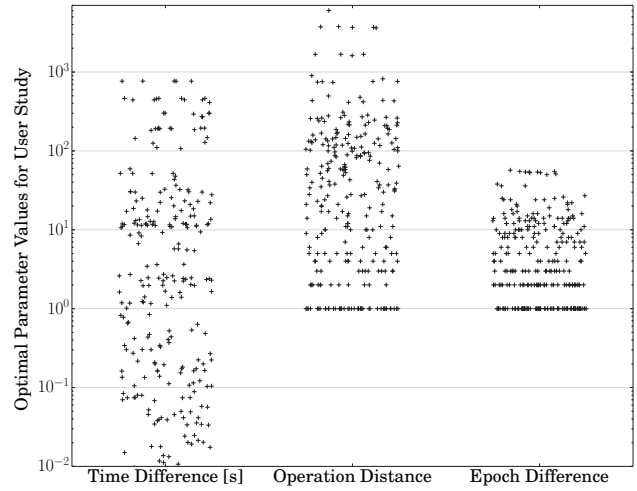


Figure 2: Parameter evaluation based on the user study: Epoch Difference has the smallest variance. Time Difference is measured in seconds and Operation Distance is measured in number of operations.

After manually labeling the user study we observed that related files were mostly accessed shortly after each other both in terms of access time and operation sequence. Often we observed no intermediate unrelated operation. As in previous work [37] we therefore find a temporal proximity between related operations. To determine the parameter for temporal proximity, we performed an analysis as seen in Figure 2. Besides the time difference between related operations and their distance in the operation sequence, we also measured the *epoch difference*.

A new epoch begins, when a new file is written to. Therefore, subsequent write operations to the same file, possibly interleaved with other read operations, do not start a new epoch. Intuitively, the epochs should capture project tasks, e.g. generating a PDF. As the time span of a task, e.g. time taken to edit a file, or the number of operations, e.g. due to the number and size of input files, can differ greatly, using the epoch distance should be more stable. Figure 2 validates this theory as the y-axis is logarithmic and the variance in the epoch difference is therefore magnitudes smaller.

However, epochs will also contain noise in the form of unrelated operations close to related operations. We tried to find features to identify related operations. The number of temporally close operations and their interleaving allowed us to more precisely identify valid access relations. This was partly due to the fact that configuration files and unrelated files were usually just read once, e.g., to parse the configuration or check the file's magic number.

Additionally we compared the paths of related operations and found a strong correlation between the longest common subsequence of paths and whether the corresponding operations were related. This was due to users mostly keeping related files in the same part of the file system and is consistent with previous results [3]. We therefore use co-location to identify related files. However, using the process ID for classification proved to be an unreliable identifier. While it pointed to true positives, it also included false positives, such as configuration files or previously accessed documents.

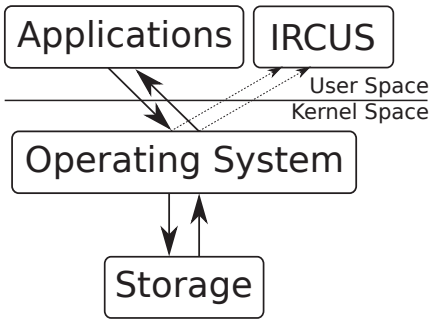


Figure 3: General Architecture: The operating system notifies IRCUS about file system events and their contents. IRCUS can operate in user space.

4. ARCHITECTURE

In this section, we describe the architecture of our proposed solution, called *IRCUS* (Identification of Related Content from User Space). Figure 3 depicts the overall architecture. IRCUS relies on the observation of file system operations in order to identify relations between files. To achieve this, IRCUS, which runs as a user-space application owned by the user, receives file system events from the underlying system which are triggered by the user's running applications. These events include typical file operations such as open, read, write, flush, rename and delete. In the case of read and write operations, the associated content that was read or written is expected to be included in the received event notification, as this is necessary in order to identify content-related files. More specifically, each read/write event notification contains the following information:

$$\langle operation,\ path,\ offset,\ num_bytes,\ content,\ pid \rangle$$

where *operation* is the type of the file operation (read or write) on the file specified by *path*. The *content* of size *num_bytes* is read from or written to the *offset*. Optionally, *pid* is the process ID of the process that triggered this operation.

Since IRCUS only looks at file system level operations, its operation is agnostic to and independent of the kind of applications that generate these operations. It also does not have to monitor any kind of interprocess communication (e.g., copying and pasting a text excerpt from one application to another). This is due to the fact that if any kind of interprocess communication leads to a file relation, the relation will be eventually captured by IRCUS through the inspection of the file operations initiated by the communicating processes.

We assume that the user of the system will not behave maliciously, i.e., will not try to actively hide file relations or insert dummy operations. In this work, we also ignore the possibility of malware on the system.

As mentioned above, IRCUS runs in user space and is owned by a specific user of the system. IRCUS is only allowed to monitor file system operations that were initiated by processes running with the privileges of that particular user. We assume that there is a pre-defined set of directories that should be monitored for file system operations relevant for that user. The most typical scenario, which we follow in our implementation, is monitoring the user's home directory.

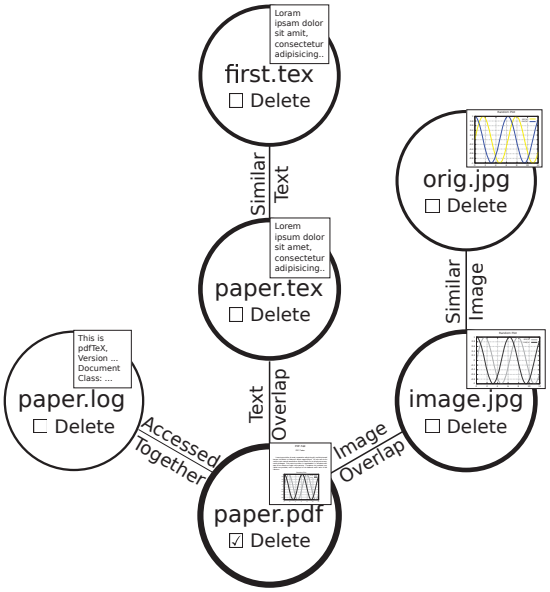


Figure 4: Generated Relation Graph: When planning to delete `paper.pdf`, the user is shown such a relation graph. Clicking on vertices or edges reveals additional information.

In a multi-user system, each user may have his own version of the IRCUS application running and monitoring each user's own directory. The underlying system should dispatch file operation events triggered by each user's applications to the corresponding instance of IRCUS.

To avoid extra file system operations and to efficiently perform the required operations for discovering relations between files, IRCUS maintains a cache that stores recent file operations. The operations are grouped by the accessed file and the files are stored in subdivision based on their file type. Every file type has a separate cache, because we want to establish content relations between certain file types. IRCUS does compare text documents to the text extracted out of a PDF document, while it does not compare text documents to image documents. A subdivided cache provides flexibility and efficiency for this chosen approach.

We picked the caching strategy based on the results of the user study, presented in Section 3.3, that related operations have a small epoch difference. We therefore keep operations from recent epochs in the cache along with the corresponding files. As previously discussed, the operations enable us to track the access relation of files more precisely. Furthermore, the robustness of epoch difference allows us to reliably evaluate content relations independent of specific operation patterns.

Once IRCUS has identified an access or content relation it stores the relation information alongside the two files as previously suggested [25]. The stored information contains the type of relation and the strength of the relation, e.g., the similarity of two images. The stored information is accessible from user space and can be used to educate the user about file relations.

Whenever the user wants to securely delete a chosen file he is able to consult IRCUS which other related files he might

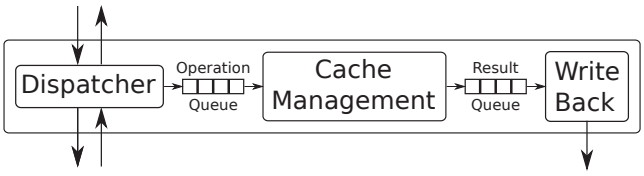


Figure 5: Implementation Design: Based on FUSE, IRCUS intercepts file system operations and multiplexes them for analysis.

also want to delete. IRCUS takes the user-provided file as input and creates a relation graph by transitively evaluating the file's relations. An example of the visualization of such a graph that would be displayed to the user is sketched in Figure 4. The graph only contains the files with the strongest relation to the provided file and provides information on the type of relation between files as well as the strength of each relation (conveyed by the thickness of each vertex). The score-based filtering [25, 36] and visualization [3] of such graphs have been previously studied. Finally, the user is able to select the subset of the graph, which should be securely deleted.

5. IMPLEMENTATION

In this section we outline how we designed our implementation, how we applied the different parts of the architecture and how we improved the performance.

The implementation of IRCUS is subdivided into multiple parts as seen in Figure 5. To observe file system operations the dispatcher thread is based on FUSE [15]. FUSE is more powerful than required, as it intercepts file system operations and enables request or result modifications. The dispatcher thread uses FUSE to multiplex the file system operations. Before returning the operation result to the OS, the dispatcher copies the operation and its content into the operation queue.

The cache management thread receives the operations, organizes the caches and evaluates access and content relation. For IRCUS we have implemented four types of caches: Text, Image, PDF and Binary. Binary is the default that contains all non-matched entries. These caches allow us to demonstrate the capabilities of IRCUS for detecting content relation between simple and complex file types.

The cache entry for a file consists of a sparse representation of the file parts that were read or written. Additionally the cache contains a file-type specific addendum that is used for more efficient content comparison. To perform content comparison IRCUS uses different algorithms for the different file types. These algorithms are not optimal, yet still they demonstrate the feasibility of our scheme.

Previous work [23] suggested the Longest Common Subsequence as a text comparison metric. However, in IRCUS this was too computationally expensive. We therefore adopted Broder's resemblance [4]. It allows us to compute a fixed size "sketch" for every text document and compute the resemblance using two sketches. We store the sketch as the addendum, in order to avoid multiple computations.

For image comparison we extract the color histograms using OpenCV and perform a histogram comparison. Here again we can save the histogram as addendum. When comparing binary data we try to detect identical blocks of data. However, these blocks might be unaligned, therefore we use the rsync algorithm [39]. The rsync algorithm uses a rolling hash to quickly identify potential candidates and then uses a strong signature, saved in the addendum, to compare the data blocks. We filter the results so that blocks of mostly or only zeros are not identified as content related. We also limit the maximum cache size of a binary file, to avoid memory exhaustion when large files are read or written.

PDF is a complex file type, as it is a file container. For PDF files we perform three types of comparison: PDF-to-Text, PDF-to-Image and PDF-to-PDF. To enable comparison we extract text and images from the PDF using the poppler library and save them as addendum. For PDF-to-Text we compare the extracted text to the given text with the previously presented text comparison. Analogously for PDF-to-Image we compare all images from the PDF to the given image. Finally to perform PDF-to-PDF we extract both PDFs and perform text comparison and pairwise image comparison.

IRCUS has to avoid unnecessary content comparisons. For some file types content comparisons are only useful after caching the complete file. To make the most use of content comparisons and to avoid comparisons that have to be repeated after a file update, IRCUS triggers the content comparisons on two events: when a file gets flushed or when a file gets evicted from the cache. The flush operation is triggered by FUSE when the file is closed. This usually means that this file has been completely updated. Furthermore IRCUS performs a content comparison before a file is evicted to detect similarity with recently inserted files.

To further reduce the number of content comparisons, IRCUS performs a comparison between a pair of files only if at least one of the files has been written. This is based on the assumption that IRCUS was in place from the beginning, so that a potential content relation would have previously been detected. This heuristic avoids a lot of content comparisons, as lots of files are only read, e.g., configuration and input files. In case IRCUS was not present from the beginning an initial search could be performed as described in Section 3.2.

However, even with the presented approach we still observe duplicate content comparisons. If file A is compared to B shortly before B is evicted from the cache, the comparison between A and B might be duplicate. Whether it is duplicate depends on a possible modification of A or B in the meantime. To keep track of the order of events, we introduce a logical clock [22]. The logical clock is incremented whenever the cache state changes. For every cached file we save the logical timestamp of the last modification and the last comparison. By checking these timestamps we can avoid unnecessary comparisons.

Once a content relation is detected, it is saved in the cache entries of the corresponding files. The cache entries also contain scores for the access relation between files. We found that the following way of computing the access relation was both simple and powerful:

$$AR(A,B) = \sum_{op_A \in Ops_A} |\{op_B \mid ed(op_A, op_B) \leq T_{ED}\}|$$

To compute the access relation score of two files A and B, IRCUS counts the number of temporally close operations on B around every operation on A. Temporally close operations have an epoch difference $ed(\cdot)$ smaller or equal to the

epoch difference threshold T_{ED}. Then IRCUS adds up all the counts for the final score. Therefore, the access relation is symmetric, simple to compute and allows effective filtering as explained in Section 3.3. From the results of the user study we chose the threshold for "closeness" $T_{ED} = 12$. This is a conservative choice to avoid an overfitting to our small dataset.

The cache entry for a file is treated as a logical unit and so it is evicted atomically once the file has not been accessed in the past T_{ED} epochs. The value of T_{ED} is tradeoff between higher detection and higher overhead. After the eviction-triggered content comparisons have been performed, the cache entry is removed from the cache and inserted into the result queue.

The write back thread takes elements out of the result queue. It is responsible for permanently storing the relation information. We decided to store the information in the extended attributes of the file system. Before storing the relation information of file A, the write back thread filters the relations to reduce false positives and thereby also reduce storage overhead. Based on our labelled user data we tested eight different techniques with different parameter values for filtering false positives while trying to preserve true positives.

Three techniques worked well:

Top X Percent Keep the highest access relation scores of A. For this technique the top 5% gave the best results.

X-times Bigger than Mean Compute the mean of A's access scores and keep only scores at least 4-times bigger than the mean.

X-times Bigger than Local Graph Mean Compute the mean score of the local graph consisting of A and all its related files and only keep scores at least 4-times bigger than the mean.

As it preserved most of the true positives while filtering false negatives we picked the "4-times bigger than Mean" filter. Besides such access relations we store all content relations and access relations of co-located files. We define files as co-located, if their paths have a Longest Common Subsequence of at least 75%.

In the extended attributes alongside each file we store its related files with the corresponding access relation and content relation score. If the files were accessed by the same process we also store the process name for later display to the user. The file relations are stored as bi-directional links. This allows efficient updates, e.g., in the case of rename operations. In this case we can quickly identify the related files and update their extended attributes.

Once the user wants to perform a secure deletion, the related files are available in the extended attributes. Using multiple lookups the user can construct a relation graph as in Figure 4 that can be enriched with the stored information, such as the relation strength and the process names.

6. EVALUATION

In this section we present our experimental results. We tested the performance in terms of throughput and latency and conducted a case study for detection results.

As IRCUS is designed for a desktop system, we tested our implementation from Section 5 on a Lenovo Thinkpad X230 running GNU/Linux equipped with a regular hard disk. As underlying file system we picked reiserfs, a standard Linux file system with support for extended attributes.

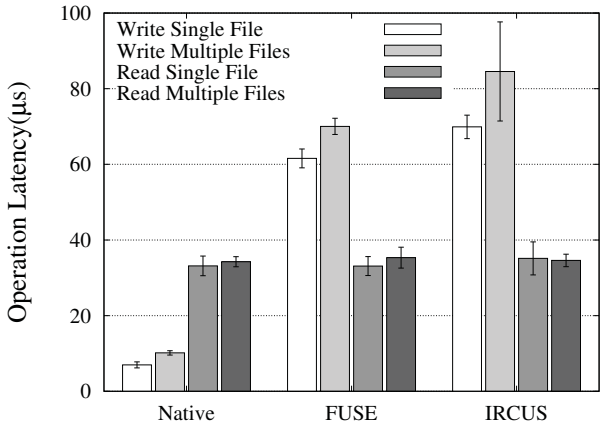


Figure 6: Latency Measurements for different scenarios averaged over 20 repetitions with standard deviation.

6.1 Performance

To evaluate the runtime performance of our system we test its throughput and latency in different scenarios. As points of reference we chose the native file system performance and the performance of a "simple overlay" FUSE file system. This way we can investigate the overhead introduced by FUSE. As "simple overlay" we chose the FUSE example file system "fusexmp_fh" that simply forwards requests.

In order to test latency and throughput we generate 32 pseudo-random, binary 16 MB files. We then use all or some of the files as follows. All the described operations are sequential.

Write Single File Measure the performance of writing a single file from memory to disk 32 times, repeatedly overwriting it.

Write Multiple Files Write all 32 files from memory to disk to fill the cache. Then, measure the performance for writing all 32 files from memory to disk but using different file names.

Read Single File Measure the performance of reading a single file from disk repeatedly.

Read Multiple Files Measure the performance of reading all 32 files sequentially from disk.

We chose these measurements to highlight potential performance differences. When running measurements with multiple files our system has to evaluate and manage file relations, e.g., perform content comparisons. None of this is necessary if only a single file is used. Therefore, "Write Multiple Files" tests the worst-case performance of our system as all files are written and the cache is filled from the beginning.

In order to eliminate the influence of caching, we clear the Linux page cache before each of the 32 measurement

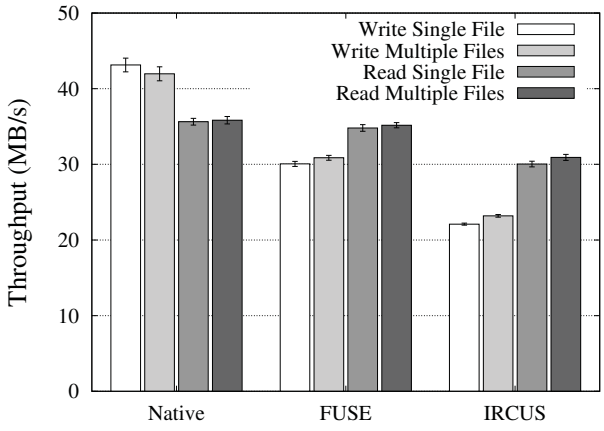


Figure 7: Throughput Measurements for different scenarios averaged over 20 repetitions with standard deviation.

steps. We did not use the FUSE `directio` option to reduce caching, as this option led to reduced functionality of the file system.

The latency measurements are shown in Figure 6. Each of the tests has been repeated 20 times and results have been averaged. We see that FUSE introduces a significant increase in write latency. This is due to the additional copy operations and transitions between kernel and user space. Additionally, IRCUS requires time for the dispatcher to copy the file system operation into the operation queue. As explained "Write Multiple Files" tests the worst-case performance of our system. Evaluating the different file relations occupies the cache management thread so that the operation queue fills up and blocks the dispatcher thread. We picked a fixed-size operation queue of 32 elements to demonstrate such performance bottlenecks.

Overall, the additional latency of IRCUS over FUSE is caused partially by multiplexing the operation inside the dispatcher thread that we cannot avoid. This latency overhead is not general for the architecture, but is specific for our implementation based on FUSE. A different implementation could avoid such overhead.

In Figure 7 we present our throughput measurements. As the file operations in the tests are sequential, the results for the native system are directly related to the latency results. The throughput of FUSE is lower due to its extra latency. FUSE also introduces an overhead when all the write operations are performed on the same file. Finally the throughput of IRCUS is the lowest due to the additional latency on top of FUSE. However, the IRCUS throughput is sufficient to watch high definition videos without interferences.

6.2 Detection

As it makes use of heuristics, IRCUS is not perfect. The exact detection accuracy, however, depends on the specific workload and the used applications. To provide a deeper understanding about the features and shortcomings of IRCUS we present multiple test cases. These test cases are separated into different work profiles as previous work has shown that work profiles differ with roles [23]. The test cases were performed in the setup described above.

All of these tests are only performed once. Additional repetitions would increase the accuracy as random correlations and other noise could be filtered out.

Office Workload

These test cases simulate the usage of IRCUS in an office environment. Based on previous work [17], these test cases contain web content, emails, documents, pictures and presentations. We use the following applications: Firefox, Thunderbird, LibreOffice, Gimp and Gedit. To create a realistic scenario, Firefox and Thunderbird are constantly running, while other applications are opened and closed. We now describe the performed test cases and the resulting detection by IRCUS.

Saved Attachment A binary file is received as a mail attachment and automatically stored inside the mail database. The user then saves the attachment as a file.

As the attachment is encoded when it is stored inside the mail database and IRCUS is unaware of the database's format, IRCUS cannot establish a content relation. However, due to the correlated access patterns, IRCUS establishes a link between the saved file and the mail database. Additionally, an incorrect access relation to an unrelated file is generated.

Shared Image A text document containing an image is opened, the image is copied to the clipboard and inserted into a newly created presentation.

Even though these files are opened directly after another, the text document has already left the cache before the presentation is cached. This is due to the many additional file system operations by LibreOffice. However, as the used clipboard manager saves clipboard images to the disk, the document and the presentation are indirectly linked through the clipboard file. These links are access relations as the image is represented in different formats. Besides this access relation, both files have a few false positives, such as a LibreOffice configuration file used for both files.

Multiple Downloads A file is downloaded to the Downloads folder, after additional browsing and three intermediate downloads, the file is downloaded again and saved in a different folder.

The additional browsing creates so much noise in the form of write operations that the two identical downloads are never compared to each other. Therefore this relation is undetected.

Copy Operation After a binary file has been downloaded, it is copied into a new file.

The copy operation is successfully detected through a content overlap. Additionally both files are linked to the Firefox history through access relations. This is due to the original file download.

Save-As DOC A document is opened in LibreOffice and small changes are made throughout the document. The resulting document is saved under a different name.

This relation is undetected since IRCUS can't interpret DOC files and they are therefore treated as binary. Additionally, there was too much noise for the creation of an access relation.

Save-As Text A text file is opened inside Gedit, where small changes are made and a paragraph is repositioned. The resulting text file is saved under a different name.

As IRCUS analyzes the text differences it finds a content relation between the two files. This demonstrates the importance of file types when comparing to the **Save-As DOC** test case.

Copy-Paste DOC An unchanged text sequence is copied from one document into another document.

Similarly as before this is undetected, as DOC files are treated as binary files and the binary representations differ.

Copy-Paste Text An unchanged text sequence is copied out of one text file into another text file.

As the copied percentage was sufficiently big, a content relation between the two files is observed by IRCUS. The other access related files, however, are false positives.

Copy-Paste JPG A JPG is opened in Gimp, and a part of it is selected and copied into a new image.

As IRCUS performs a histogram-based comparison the content overlap is detected. Additionally, the two files have an indirect access relation through a Gimp temporary file.

Engineering Workload

These test cases simulate the usage of IRCUS in an engineering environment. They are based on compilations for which IRCUS tries to identify the input files.

PDF Generation A PDF is compiled out of a text and two image sources using `pdflatex`.

The three input files are detected as inputs through content relations, access relations and the common process ID. Additionally, the generated bibliography file is linked to the PDF through an access relation and the common process ID. This demonstrates the feasibility of detecting related files for complex file types.

Source Code Compilation An executable is compiled out of multiple source code files using `gcc` and `g++`.

Access relations link the resulting executable to all source and object files. Based on the executable all related files can be identified.

6.3 Discussion

We observe that IRCUS is able to detect relations for the "known" file types (Text, Image and PDF) reliably and precisely. For other file types, it relies on access relations, which contain false positives, especially if only a single operation is observed. We could filter some of these false positives using blacklists, but we aim to provide a general approach. Instead we want to use aggregated information to detect frequent false positives, e.g., a configuration file that is related to many documents.

One could argue that IRCUS needs to understand all file types to be useful. However, previous work [17] has shown that relatively few file types cover most of the related files. Additionally, IRCUS successfully detected access relations, such as between the source code files and the executable or the saved attachment and the mail database.

When relations including databases, e.g., mail database or browsing history, are detected the user should be warned upon deletion to delete such content through the appropriate application interface.

We can also observe the trade-off between increased runtime overhead and improved detection results due to the number of cached operations. Caching less operations (smaller T_{ED}) requires fewer content comparisons and thereby reduces resource usage. When encountering noise due to unrelated operations as in the **Multiple Downloads** test case, however, content relations might be missed due to the smaller threshold T_{ED}.

Overall, implementing additional file types and tuning the parameters could significantly improve the detection results. Fortunately, as IRCUS runs in user space, users or organizations can add such file types based on their specific needs and adapt the system performance to their user profile.

7. RELATED WORK

In this section we survey related work from the different areas this work touches on.

Information Flow Tracking

Systems such as TaintDroid [8] or Panorama [41] employ kernel/OS modifications to perform taint tracking. They face issues as over- or undertainting. CleanOS [38] secures sensitive data on Android by encrypting it in memory and modifying the Dalvik interpreter to decrypt it with keys from the cloud. The Aquifer system [28] restricts information flow based on a policy framework to prevent accidental data disclosure on Android. It limits interactions between applications based flow data provided by systems like TaintDroid or CleanOS. PRIVEXEC [30] creates virtual, encrypted file systems for different applications. This isolates applications from each other. TightLip [42] uses doppelganger processes with bogus data to observe the flow of labelled data. TightLip triggers hen the two processes use different system call arguments.

In contrast our system does not require kernel or OS modifications, is not dependent on taint tracking and does not impose any restrictions on the system.

Forensics

Related work in forensics includes Backtracker [21], a tool to identify the events leading to a malicious file operation. Backtracker tracks all system calls through virtualization or kernel introspection and displays a graph of the relevant events. To reduce the significant number of false positives, later work [37] has enhanced it with data flow analysis. Forensix [12] performs full system call logging and identifies information sources for specific files based on the logs. Keypad [10] and Flight Data Recorder [40] log file system events in a robust and efficient manner to aid analysis. Other systems [1, 43] have been proposed to undo file system changes by malicious processes using external servers. Privacy Oracle [19] analyzes network traces generated by black-box testing and identifies reordered, fragmented or modified sensitive data.

Our system does not require virtualization or additional servers and is targeted at a non-malicious scenario.

Provenance and Causality

Related work has used provenance to improve the quality of desktop search results [36] or to helpfully annotate files during normal user operations [20]. However, these systems require tracking the data flow in all running Windows applications and hooking into specific Windows applications respectively. Thereby, they are OS- and application specific and cannot be generally applied. Previous work also defined Provenance-Aware Storage Systems [25], their functionality and concluded that provenance information should be stored alongside the files as metadata. The stored provenance information was complex including also used executables, their arguments and libraries. This information was stored as a directed acyclic graph, as the approach avoided the generation of cycles. Work building upon that [18] tried to identify the sources of information leaks through "transient provenance". Other work [13, 24] provided undo operations based on causal file relationships using kernel instrumentation.

Our system has a different goal and is independent of the OS and the applications.

Studies on Provenance/Reuse

A study on desktop provenance [17] used application instrumentation to identify that "53.7% of all Excel, PowerPoint, Word, PDF and text files . . . were related to at least one other file". The relationships were usually established through "Copy-and-Paste" or "Save as" events. Most relationship clusters were found to be small. Another study on reuse [23] identified content reuse in presentations within an organization. It found Longest Common Substring Distance to work well for identifying related texts and reported some partial and many exact copies. Overall reuse was around 10-20% and the majority of employees preferred starting from existing work.

These studies show significant reuse and frequent occurrence of related documents and therefore motivate our work.

Identification of Content Overlap or Similarity

Different approaches have been proposed to evaluate content overlap or similarity based on the different usage.

To probabilistically identify similarity between multiple pieces of high-dimensional data, techniques such as locality-sensitive hashing [11] can be used. These functions will probabilistically assign similar data to the same bucket.

Different techniques such as Rabin Fingerprints [27] have been proposed to identify overlaps in binary data. The synchronization tool rsync [39] identifies unmodified file parts to avoid unnecessary copy operations. To do this efficiently, rsync computes a fast rolling checksum and a more secure checksum in case of a match.

To identify text similarity more complex metrics than longest common subsequence and longest common substring have been explored. Broder [4] studied how the resemblance and containment of two documents can be evaluated efficiently. After extracting a fixed-size "sketch" through random sampling, sketches can be used to compare documents. This saves storage and computation time when comparing many documents. Similarly, other work tried to compute a document fingerprint [35] to find matches of a certain length.

As we do not claim to provide an optimal implementation, we have only experimentally tested some of these techniques while integrating others might increase the performance of the system.

Secure Deletion

Secure Deletion has been studied in different scenarios and for different devices [31]. Previous work studied how to securely delete data through overwriting [14] and on flash devices with wear-leveling [32]. Based on the older work on how to extend such properties [2, 7] recent work has expanded into policy- and cloud-based solutions [5, 33].

These approaches assume that the user knows exactly what to delete, whereas our approach assists in finding the data that has to be securely deleted.

8. CONCLUSION

We introduced the concept of assisted deletion of related content, where the user is presented with files that should be securely deleted together to protect data confidentiality. We have demonstrated the feasibility through a user space prototype and shown that it can analyze complex file types and use heuristics to identify related data independent of its binary representation.

In the future we plan to study more user data in order to differentiate different user types and generate more accurate results. As the adoption of such systems depends on their usability, we plan to investigate how to make related content deletion more user-friendly.

9. ACKNOWLEDGEMENTS

This work was supported by the Swiss National Science Foundation (SNSF) through a Sinergia Project under grant No. CRSII2_136318/1.

10. REFERENCES

[1] A. Bacs, R. Vermeulen, A. Slowinska, and H. Bos. System-Level Support for Intrusion Recovery. DIMVA'12. Springer-Verlag, 2013.

[2] D. Boneh and R. J. Lipton. A Revocable Backup System. In *USENIX Security Symposium*, 1996.

[3] M. A. Borkin, C. S. Yeh, M. Boyd, P. Macko, K. Z. Gajos, M. Seltzer, and H. Pfister. Evaluation of Filesystem Provenance Visualization Tools. *IEEE Transactions on Visualization and Computer Graphics*, 19(12):2476–2485, Dec. 2013.

[4] A. Broder. On the Resemblance and Containment of Documents. In *Proceedings of the Compression and Complexity of Sequences 1997*, SEQUENCES '97. IEEE Computer Society, 1997.

[5] C. Cachin, K. Haralambiev, H.-C. Hsiao, and A. Sorniotti. Policy-based Secure Deletion. CCS '13. ACM, 2013.

[6] R. Chirgwin. Surrender your crypto keys or you're off to chokey, says Australia. The Register, 2014.

[7] G. D. Crescenzo, N. Ferguson, R. Impagliazzo, and M. Jakobsson. How to Forget a Secret. In *STACS*, Lecture Notes in Computer Science. Springer, 1999.

[8] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. TaintDroid: An Information-flow Tracking System for Realtime Privacy Monitoring on Smartphones. OSDI'10. USENIX Association, 2010.

[9] J. Garside. Vodafone reveals existence of secret wires that allow state surveillance. The Guardian, 2014.

[10] R. Geambasu, J. P. John, S. D. Gribble, T. Kohno, and H. M. Levy. Keypad: An Auditing File System for Theft-prone Devices. In *Proceedings of the Sixth Conference on Computer Systems*, EuroSys '11. ACM, 2011.

[11] A. Gionis, P. Indyk, and R. Motwani. Similarity Search in High Dimensions via Hashing. In *Proceedings of the 25th International Conference on Very Large Data Bases*, VLDB '99, pages 518–529, San Francisco, CA, USA, 1999. Morgan Kaufmann Publishers Inc.

[12] A. Goel, W.-c. Feng, D. Maier, W.-c. Feng, and J. Walpole. Forensix: A Robust, High-Performance Reconstruction System. ICDCSW '05. IEEE Computer Society, 2005.

[13] A. Goel, K. Po, K. Farhadi, Z. Li, and E. de Lara. The Taser Intrusion Recovery System. In *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles*, SOSP '05. ACM, 2005.

[14] P. Gutmann. Secure Deletion of Data from Magnetic and Solid-State Memory. In *USENIX Security Symposium*, 1996.

[15] C. Henk and M. Szeredi. FUSE, Filesystem in Userspace. http://fuse.sourceforge.net/.

[16] Y. Huang, A. Stavrou, A. K. Ghosh, and S. Jajodia. Efficiently Tracking Application Interactions Using Lightweight Virtualization. In *Proceedings of the 1st ACM Workshop on Virtual Machine Security*, VMSec '08. ACM, 2008.

[17] C. Jensen, H. Lonsdale, E. Wynn, J. Cao, M. Slater, and T. G. Dietterich. The Life and Times of Files and Information: A Study of Desktop Provenance. CHI '10. ACM, 2010.

[18] S. Jones, C. Strong, D. D. E. Long, and E. L. Miller. Tracking Emigrant Data via Transient Provenance. TaPP '11, 2011.

[19] J. Jung, A. Sheth, B. Greenstein, D. Wetherall, G. Maganis, and T. Kohno. Privacy Oracle: A System for Finding Application Leaks with Black Box Differential Testing. CCS '08. ACM, 2008.

[20] A. K. Karlson, G. Smith, and B. Lee. Which Version is This?: Improving the Desktop Experience Within a Copy-aware Computing Ecosystem. CHI '11. ACM, 2011.

[21] S. T. King and P. M. Chen. Backtracking Intrusions. *ACM Trans. Comput. Syst.*, 23(1), Feb. 2005.

[22] L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM*, 21(7), July 1978.

[23] Y. Mejova, K. De Schepper, L. Bergman, and J. Lu. Reuse in the Wild: An Empirical and Ethnographic Study of Organizational Content Reuse. CHI '11. ACM, 2011.

[24] K.-K. Muniswamy-Reddy and D. A. Holland. Causality-based Versioning. *Trans. Storage*, 5(4), Dec. 2009.

[25] K.-K. Muniswamy-Reddy, D. A. Holland, U. Braun, and M. Seltzer. Provenance-aware Storage Systems. ATEC '06. USENIX Association, 2006.

[26] L. Munson. Lavabit appeals contempt of court ruling surrounding handover of SSL keys. Naked Security, 2014.

[27] A. Muthitacharoen, B. Chen, and D. Mazières. A Low-bandwidth Network File System. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles*, SOSP '01. ACM, 2001.

[28] A. Nadkarni and W. Enck. Preventing Accidental Data Disclosure in Modern Operating Systems. CCS '13. ACM, 2013.

[29] J. Oates. Youth jailed for not handing over encryption password. The Register, 2010.

[30] K. Onarlioglu, C. Mulliner, W. Robertson, and E. Kirda. PrivExec: Private Execution as an Operating System Service. In *IEEE Symposium on Security and Privacy (S&P)*, May 2013.

[31] J. Reardon, D. Basin, and S. Capkun. SoK: Secure Data Deletion. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy*, SP '13. IEEE Computer Society, 2013.

[32] J. Reardon, C. Marforio, S. Capkun, and D. Basin. Secure Deletion on Log-structured File Systems. *ASIACCS*, 2012.

[33] J. Reardon, H. Ritzdorf, D. Basin, and S. Capkun. Secure Data Deletion from Persistent Media. CCS '13. ACM, 2013.

[34] A. Rusbridger. David Miranda, schedule 7 and the danger that all reporters now face. The Guardian, 2013.

[35] S. Schleimer, D. S. Wilkerson, and A. Aiken. Winnowing: Local Algorithms for Document Fingerprinting. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, SIGMOD '03. ACM, 2003.

[36] S. Shah, C. A. N. Soules, G. R. Ganger, and B. D. Noble. Using Provenance to Aid in Personal File Search. ATC'07. USENIX Association, 2007.

[37] S. Sitaraman and S. Venkatesan. Forensic Analysis of File System Intrusions Using Improved Backtracking. IWIA '05. IEEE Computer Society, 2005.

[38] Y. Tang, P. Ames, S. Bhamidipati, A. Bijlani, R. Geambasu, and N. Sarda. CleanOS: Limiting Mobile Data Exposure with Idle Eviction. OSDI'12. USENIX Association, 2012.

[39] A. Tridgell. *Efficient Algorithms for Sorting and Synchronization*. PhD thesis, The Australian National University, February 1999.

[40] C. Verbowski, E. Kiciman, A. Kumar, B. Daniels, S. Lu, J. Lee, Y.-M. Wang, and R. Roussev. Flight Data Recorder: Monitoring Persistent-state Interactions to Improve Systems Management. OSDI '06. USENIX Association, 2006.

[41] H. Yin, D. Song, M. Egele, C. Kruegel, and E. Kirda. Panorama: Capturing System-wide Information Flow for Malware Detection and Analysis. CCS '07. ACM, 2007.

[42] A. R. Yumerefendi, B. Mickle, and L. P. Cox. TightLip: Keeping Applications from Spilling the Beans. NSDI '07. USENIX Association, 2007.

[43] N. Zhu and T. Chiueh. Design, Implementation, and Evaluation of Repairable File Service. In *Dependable Systems and Networks*, 2003.

Morpheus: Automatically Generating Heuristics to Detect Android Emulators

Yiming Jing[†], Ziming Zhao[†], Gail-Joon Ahn[†], and Hongxin Hu[‡]
[†]Arizona State University [‡]Clemson University
{ymjing,zmzhao,gahn}@asu.edu, hongxih@clemson.edu

ABSTRACT

Emulator-based dynamic analysis has been widely deployed in Android application stores. While it has been proven effective in vetting applications on a large scale, it can be detected and evaded by recent Android malware strains that carry detection heuristics. Using such heuristics, an application can check the presence or contents of certain artifacts and infer the presence of emulators. However, there exists little work that systematically discovers those heuristics that would be eventually helpful to prevent malicious applications from bypassing emulator-based analysis. To cope with this challenge, we propose a framework called Morpheus that automatically generates such heuristics. Morpheus leverages our insight that an effective detection heuristic must exploit discrepancies *observable* by an application. To this end, Morpheus analyzes the application sandbox and retrieves observable artifacts from both Android emulators and real devices. Afterwards, Morpheus further analyzes the retrieved artifacts to extract and rank detection heuristics. The evaluation of our proof-of-concept implementation of Morpheus reveals more than 10,000 novel detection heuristics that can be utilized to detect existing emulator-based malware analysis tools. We also discuss the discrepancies in Android emulators and potential countermeasures.

Categories and Subject Descriptors

D.4.6 [**Operating Systems**]: Security and Protection—*Invasive software*; D.2.5 [**Software Engineering**]: Testing and Debugging—*Emulators*

Keywords

Android, emulator, malware

1. INTRODUCTION

Recent years have witnessed an explosive growth of mobile applications. According to the Gartner report [18], there were 64 billion application downloads worldwide in 2012 and it was increased to 102 billion in 2013. The growth should partially give credit to application stores such as Apple AppStore and Google Play. With these services, users enjoy a centralized and trusted source for browsing and purchasing applications. Meanwhile, developers also get chances to reach a wider audience and more profits. Unfortunately, such advantages also make application stores an appealing place for distributing malicious mobile applications. To infect more unsuspecting users, adversaries would attempt to publish their malware in application stores without being detected.

To effectively mitigate such attempts, the security community and the application stores have deployed emulator-based dynamic analysis. Compared with static analysis that can be thwarted by obfuscation and encryption [15], emulator-based dynamic analysis can vet runtime behaviors of applications on a large scale. As a result, researchers have launched such systems [3, 4, 6, 23] to inspect file, network, and cellphone operations. Moreover, Google Bouncer [17] vets applications using QEMU-based emulators in its cloud infrastructure [20]. And Bouncer helped drop the number of malware downloads in Google Play by 40% in 2011 [17].

Despite the apparent success, a flaw of emulator-based dynamic analysis lies in the discrepancies between emulators and real devices. Such discrepancies, if observable by applications, may lead to detection heuristics (*a.k.a.*, "red pills") that indicate the fabricated reality of Android emulators. Taking advantage of these heuristics, Android malware can build split personalities and circumvent dynamic analysis as previously observed in PC malware [8]. Indeed, the security community has already discovered Android malware samples that use such heuristics to evade dynamic analysis [9, 11].

The heuristics in newly discovered Android malware samples, unlike their ancestors in PC malware, exploit the peculiarities of Android. Such heuristics check the presence or contents of certain artifacts (*e.g.*, files, APIs) in Android emulators. They do not depend on native code because native code has been of particular interest to some malware analysis systems [30]. Furthermore, they can be integrated into Android applications with simple algorithms. For example, the presence of a file `/sys/qemu_trace` indicates QEMU-based emulators [19]. A full-zero string returned by the Android API `getDeviceId` indicates Android SDK emulators [21]. Meanwhile, researchers have demonstrated the effectiveness of similar heuristics [21, 25]. This alarming trend calls for a comprehensive study of these detection heuristics to better understand their magnitude and accuracy.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ACSAC '14, December 08 - 12, 2014, New Orleans, LA, USA
Copyright 2014 ACM 978-1-4503-3005-3/14/12 ...$15.00.
http://dx.doi.org/10.1145/2664243.2664250.

Regrettably, all known detection heuristics that target Android emulators are discovered piece by piece in an ad-hoc fashion. For example, some heuristics are discovered through dissecting malware samples [9, 11]. Such a reactive approach cannot predict unknown heuristics. Other known heuristics are derived from manual analysis on specific components of Android emulators [19, 24, 25]. Even though this approach is proactive, manual analysis inevitably cannot address the multitude of components in Android emulators.

To convey the severity of the problem and get ahead of malicious adversaries, we propose a framework called Morpheus that proactively and automatically generates Android emulator detection heuristics. Unlike existing approaches, Morpheus retrieves and analyzes Android system artifacts observable by Android applications in their sandboxes, called *observable artifacts*. In particular, Morpheus starts from an analysis of the current Android application sandbox to identify sources of observable artifacts. Afterwards, Morpheus employs a probe application to automatically retrieve observable artifacts from both Android emulators and real devices. Morpheus then analyzes the artifacts and their contents to produce candidate heuristics. Finally, Morpheus ranks the candidates as its output.

We evaluate Morpheus with two steps. First, we apply Morpheus against widely deployed QEMU-based Android emulators and emerging VirtualBox-based emulators. The results are more than 10,000 novel detection heuristics that can be used to detect both types of emulators as a whole or either of them. We also investigate them to reveal and characterize the discrepancies between Android emulators and real devices. Second, we assemble a group of the top-ranked detection heuristics and evaluate their accuracies against 9 emulator-based Android malware analysis tools and 128 real devices. This group of heuristics could accurately detect the evaluated emulators and real devices. To better mitigate threats of emulator detection, we will release our discovered heuristics to the security community at `http://honeynet.asu.edu/morpheus/`[1].

We summarize the contributions of this paper as follows:

- *New techniques.* We develop new techniques that make the first step towards proactive and automated generation of Android emulator detection heuristics. We integrate these techniques into a framework, called Morpheus, to systematically generate heuristics that offer low false-positives and low false-negatives.
- *New findings.* We discover a large number of novel detection heuristics and reveal the underlying discrepancies between Android emulators and real devices. Our experiments against existing malware analysis tools and a large number of real devices demonstrate high accuracy of our discovered heuristics.

The remainder of this paper proceeds as follows. Section 2 provides the background of emulator detection and the threat model for our work. Section 3 describes the design of Morpheus and its components. Section 4 elaborates our discovered heuristics and their exploited discrepancies. Section 5 presents our experiments with a group of the top-ranked detection heuristics. Section 6 discusses countermeasures and limitations of our approach. Section 7 describes the related work. Section 8 concludes this paper.

[1]To prevent misuse, we may require verifying user identities before the dataset can be downloaded. Please visit our project website for further information.

2. BACKGROUND AND THREAT MODEL

In this section, we first describe the background of emulator detection in Android malware. We then present the attack model that this work is based on.

2.1 Detection Heuristics

Due to the peculiarities of Android, we argue that Android malware would be reluctant to reuse previous PC emulator detection heuristics. First, Android malware faces a unified runtime environment whose underlying implementation details (*e.g.*, hardware differences) are concealed by the Android middleware and APIs. At the same time, Android malware has been deprived of many capabilities that allow accessing low-level system artifacts by the Android application sandbox. In addition, Android malware would prefer detection heuristics implemented with Java code rather than native code. As native code is used by only a small fraction of benign Android applications but most malicious root exploits [30], native code would draw attention of analysis tools, breaking a detection heuristic's basic purpose of evading analysis.

The detection heuristics found in newly discovered malware samples seem to be in line with our argument. They allow an application to detect emulators without bypassing the application sandbox and without the assistance of native code. For example, a popular detection heuristic involves an Android API `getDeviceId` that returns the IMEI of an Android device. This heuristic calls `getDeviceId` and tests whether "`000000000000000`" is a substring of the returned value of `getDeviceId`. It can be implemented with only two lines of Java code and thus leaves relatively small footprints. Despite that researchers have discovered similar detection heuristics and evaluated their effectiveness against Android SDK emulators, the magnitude and accuracy of such heuristics remain unknown, which results in an impediment to the development of comprehensive countermeasures.

2.2 Threat Model

In our threat model, we assume emulators that run Android with default configurations. We also assume the presence of passive anti-detection techniques, which do not proactively instrument the application to suppress the execution of detection heuristics. This is also the common setup of the existing deployed emulator-based dynamic analysis systems.

In addition, we assume a malicious Android application that does not bypass the application sandbox or carry any native code. Meanwhile, we allow this application to request any Android permission. In other words, this application's capabilities are no more than those of the benign applications in application stores. Afterwards, it applies detection heuristics that check the presence or contents of certain artifacts. Based on the result, it determines whether it is running in an emulator or not.

Once this application detects emulators, it could stay dormant or exhibit legitimate behaviors. Furthermore, this application can use dynamic external code loading to evade both static and dynamic analysis, because it only downloads the malicious payload when it is in a real device. Alternatively, it can perform reconnaissance within the emulator and phone home to facilitate generation of up-to-date detection heuristics for future attacks. We attempt to understand whether the detection heuristics can be successful in terms of detecting Android emulators and real devices.

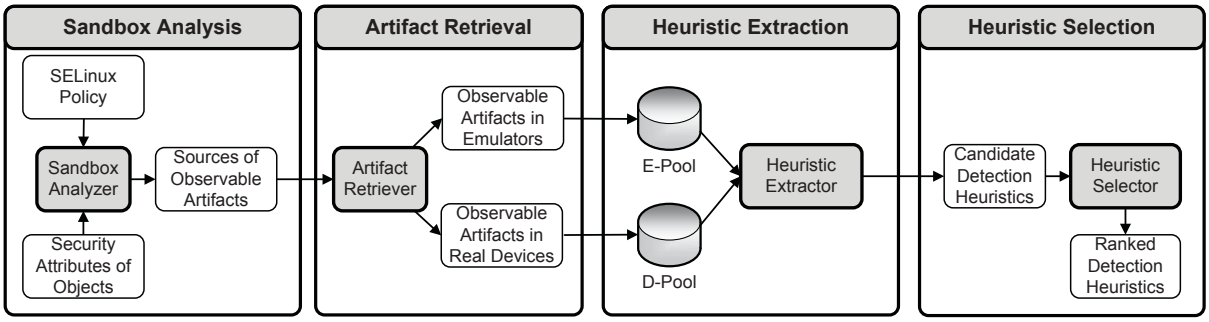


Figure 1: Morpheus: System Architecture

3. DESIGN AND SYSTEM COMPONENTS

In this section, we describe our framework Morpheus which is equipped with a suite of techniques for supporting automatic generation of detection heuristics. Despite the fact that Android emulators may contain plenty of discrepancies, a malicious application cannot exploit them unless it can observe them from within the application sandbox. Here we define *observable artifacts* as artifacts (*e.g.*, files, APIs) whose presence can be probed or whose contents can be read by any Android application in its sandbox. For example, suppose a file is not readable but its parent directory is listable, this file is still an observable artifact. The key idea of Morpheus is to retrieve and analyze observable artifacts.

As depicted in Figure 1, Morpheus consists of four components. The sandbox analyzer analyzes the default configurations of the Android application sandbox to identify sources of observable artifacts. For respective sources, the artifact retriever enumerates observable artifacts and retrieves their contents. The retrieved observable artifacts are uploaded to two pools for both emulators and real devices, respectively. The heuristic extractor then analyzes the pools by finding the artifacts or substrings of their contents that appear in most emulators but a small fraction of real devices, and vice versa. These artifacts and substrings constitute candidate detection heuristics. Finally, the heuristic selector ranks the candidates as the output of Morpheus. We next describe each component in detail.

3.1 Sandbox Analyzer

Applications' accesses on artifacts are regulated by the Android application sandbox, which is based on discretionary and mandatory access control (DAC and MAC). The Linux kernel provides DAC, which grants accesses by checking permissions of objects. Security-Enhanced Linux (SELinux) adds MAC over DAC starting from Android 4.3. SELinux grants accesses by checking domains of subjects (*e.g.*, `untrusted_app`), types of objects (*e.g.*, `wallpaper_files`), and SELinux permissions (*e.g.*, `open`, `read`)[2].

To identify sources of observable artifacts, we need to access all the objects in the Android OS. However, it is infeasible to do so in off-the-shelf Android devices due to the application sandbox and lack of root privileges. Instead, we propose the sandbox analyzer that analyzes the reference SELinux policy in Android and the security attributes (*e.g.*, owners, permissions, xattr) of objects in rooted reference Android devices (*e.g.*, Nexus devices). Specifically, we attempt to identify the objects whose security attributes expose themselves to third-party applications. Given that third-party applications are automatically assigned into the `untrusted_app` domain during installation, we simulate DAC and MAC checks to identify the following objects: (1) objects that are world-readable or under world-listable directories; and (2) objects that are accessible by `untrusted_app` using read-like SELinux permissions (*e.g.*, `read`, `recv_msg`, `ioctl`). From such objects, we then distill the sources of observable artifacts based on their owners and SELinux types, along with proper methods to retrieve them. For example, `/dev/binder` has the SELinux type `binder_device`. Its SELinux type indicates that it belongs to the Binder IPC subsystem that allows an application to access remote artifacts in system services. Such artifacts would require Binder-specific methods to retrieve. As variations in the hierarchy of objects across different Android versions are insignificant, the sources of observable artifacts derived from the reference inputs should be applicable in emulators and real devices.

We stress that the sandbox analyzer is much more conservative compared with the current Android application sandbox. SELinux in Android 4.3 is configured to permit every access. Even in Android 4.4, SELinux only protects several critical system daemons and does not confine third-party applications (*i.e.*, `untrusted_apps`). With that said, the true amount of observable artifacts in current Android devices could be much larger. However, considering the possibility that SELinux may extend its coverage in the upcoming versions of Android, we choose to be conservative for the future effectiveness of our detection heuristics.

3.2 Artifact Retriever

The artifact retriever is essentially a probe application. It requests all the available Android permissions to maximize its capabilities within the confinement of the application sandbox. Based on the identified sources of observable artifacts, we implement the corresponding methods in the artifact retriever to automatically retrieve the observable artifacts as well as their contents.

To address the various sources of observable artifacts, we propose three foundation modules in the artifact retriever: a directory walker, a Java function caller, and a Binder IPC caller. They are tailored to the peculiarity of Android and can be easily adapted and combined. Specifically, the directory walker traverses file-like artifacts. The Java function caller enumerates and manipulates both public and hidden Android APIs. The Binder IPC caller directly triggers remote system services (*e.g.*, `TelephonyManagerService`) with dynamically constructed Binder IPC messages.

[2]We ignore users, roles, and security levels for brevity because they are rarely used in the context of Android.

We launch the artifact retriever into both Android emulators and real devices. It probes the surrounding observable artifacts with its carried modules. It technically captures the first 1KB of each artifact's contents if readable. Upon explicit errors (*e.g.*, denied access), it records the error messages as the retrieved contents. Upon implicit errors (*e.g.*, blocking read), it uses a timeout to ensure that it does not hang there infinitely. We note that, the artifact retriever must upload artifacts to the correct pool according to where the artifacts are observed. For example, artifacts collected from emulators should never go into D-Pool. This is critical for the heuristic extractor to work effectively, because arbitrary noises could make the problem of heuristic generation NP-hard [16].

3.3 Heuristic Extractor

The inputs of the heuristic extractor are two pools, namely E-Pool and D-Pool, which contain instances of observed emulators and real devices, respectively. Each instance is a collection of key-value pairs that map retrieved artifacts to their contents. A key (artifact) occurs in an instance once at most, although it can occur in multiple instances. And a value (content) can be null if the artifact retriever fails to read the contents. Next, we describe two categories of detection heuristics that generate decisions based on the artifacts and their contents, respectively.

3.3.1 Artifact-based Heuristics

We start from a category of heuristics that make decisions based on the presence of artifacts. First, we attempt to discover the artifacts that are exclusively used by emulators, such as emulator-specific hardware, software, and configurations. As we use their presence to imply emulators, we refer to them as Type E artifacts. Furthermore, we also look for the artifacts that appear in most real devices, which become our Type D artifacts.

We propose two metrics, $COV_E(a)$ and $COV_D(a)$ to denote the fractions of instances in E-Pool or D-Pool that contain artifact a, *i.e.*, $COV_E(a) = \frac{|E_a|}{|E|}$, and $COV_D(a) = \frac{|D_a|}{|D|}$. Intuitively, our heuristics should at least perform better than a 50/50 guess. Thus, we choose Type E and Type D artifacts from all the artifacts in both pools according to their values of $COV_E(a)$ and $COV_D(a)$ as follows:

- **Type E artifacts:** $COV_E > 50\%$, $COV_D < 50\%$
- **Type D artifacts:** $COV_E < 50\%$, $COV_D > 50\%$

3.3.2 Content-based Heuristics

However, there are plenty of artifacts that are prevalent in both emulators and real devices. For example, Android APIs would have both COV_E and COV_D larger than 50%. Inspired by Hamsa [16], we propose a category of detection heuristics whose decisions are based on tokens, where token is a contiguous byte subsequence in the contents of an artifact. Similar to what we introduce for artifact-based heuristics, we attempt to find Type E and Type D tokens.

Specifically, for an artifact a and its retrieved contents in E-Pool, we extract a set of tokens by computing common substrings among the contents. We then extract another set of tokens for D-Pool. Combining these two sets of tokens as a token set T, we compute $COV_E(a,t)$ and $COV_D(a,t)$, which are the fractions of instances in E-Pool and D-Pool whose contents of artifact a contain token t, *i.e.*, $COV_E(a,t) = \frac{|E_{a,t}|}{|E|}$ and $COV_D(t) = \frac{|D_{a,t}|}{|D|}$. Based on the values of $COV_E(a,t)$ and $COV_D(a,t)$, we select two type of tokens as our content-based heuristics as follows:

- **Type E tokens:** $COV_E > 50\%$, $COV_D < 50\%$
- **Type D tokens:** $COV_E < 50\%$, $COV_D > 50\%$

There are various algorithms that effectively compute common substrings. We opt for a suffix array in our heuristic extractor. Constructing a suffix array runs in $O(nlogn)$ time in worst case scenario and consumes $5n$ bytes of memory, where n is the total size of the contents of an artifact in a pool. Extracting tokens from the constructed suffix array can be implemented using a binary search. Furthermore, as we prefer longer tokens in the context of generating detection heuristics, we add one more step to prune tokens that are substrings of the other tokens as long as they share the same COV_E and COV_D.

The output of the heuristic extractor is a set of Type E and Type D heuristics. Each heuristic is represented as a 3-tuple $(artifact, token, type)$. $token$ can be null for artifact-based heuristics. $type$ implies the decision to be made once the observed artifact/token matches the artifact/token specified in the heuristic. The matched Type E heuristics indicate emulators and the unmatched ones indicate real devices. Conversely, Type D heuristics imply the opposite decision.

3.4 Heuristic Selector

We propose the heuristic selector to rank the candidate detection heuristics generated by the heuristic extractor. In general, we reduce the problem of ranking the candidates to the problem of feature selection in supervised learning. E-Pool and D-Pool comprise a training set consisted of instances that are correctly labeled with "emulator" or "real device." Furthermore, we have extracted a set of detection heuristics that can be considered as binary features. Now we need to select the relevant and non-redundant detection heuristics that would best classify future observations.

We propose to use a random forest [14], which is an ensemble learning method that leverages a multitude of decision trees for classification. Each individual decision tree covers a random subset of the features and is trained with a random subset of training samples. Afterwards, the random forest fits the training set by letting each decision tree predict its unseen samples and evaluate the errors. During this process, an *importance score* for each feature is measured based on how significant the error rate would change if the feature is removed from the decision trees.

We use this importance score as a metric to rank the candidate heuristics. On one hand, relevant heuristics that contribute much to classification naturally get higher importance scores. On the other hand, redundant heuristics that exploit the same artifact/token as other heuristics are assigned zero or lower importance scores. As such, the final output of the heuristic selector is a set of relevant and non-redundant detection heuristics as sorted by their importance scores derived from the random forest.

As the number of detection heuristics is much larger than the number of instances in the pools, the random forest may suffer from over-fitting, which overestimates the importance level of some heuristics. To suppress over-fitting, we choose to increase the number of decision trees in the random forest. As more trees are added, its tendency to over-fit generally decreases as no single feature can affect every decision tree.

4. FINDING DETECTION HEURISTICS

We ran our experiments with Morpheus against QEMU-based Android SDK emulators [1], VirtualBox-based Genymotion emulators [2], and real devices. In this section, we elaborate our experiments that lead to the findings of 10,632 detection heuristics. We then characterize the heuristics according to the underlying discrepancies that they exploit.

4.1 Experimental Setup and Findings

To understand the observable artifacts in the reference Android devices, we adopted an instance of the SDK emulator and a Galaxy Nexus phone that both run Android 4.4. We traversed their mounted file systems to obtain the security attributes of objects. We then acquired a copy of the default SELinux policy from the Android Open Source Project (AOSP). Using these as inputs, the sandbox analyzer identified 33 sources of observable artifacts. However, retrieving all of them requires plenty of domain-specific knowledge for tasks such as enumerating artifacts and constructing valid inputs. In this work, we only retrieved 3 sources that could possibly lead to discrepancies and cover a sufficient number of observable artifacts.

Procfs and Sysfs: Procfs and sysfs are both pseudo file systems that expose kernel objects to userspace programs. Specifically, procfs presents system information, such as loaded kernel modules, mounted filesystems, and network stacks. Sysfs exports hardware information such as connected block devices, buses, and power states. Our implementation of the artifact retriever traversed these two file systems mounted at `/proc` and `/sys`. In particular, we slightly adapted the directory walker to handle looped symbolic links that are prevalent in procfs and sysfs.

Android APIs: A large number public and hidden APIs are exposed by Android system services. For example, `TelephonyManagerService` exposes APIs that return unique device identifiers to applications. Actually, the APIs are implemented with the underlying Binder IPC framework, which handles the IPC between applications and system services through a Binder device node located at `/dev/binder`.

To probe APIs behind Binder, we implemented two approaches in the artifact retriever. We used the reflection-based Java function caller to enumerate and call APIs. We also adapted the Binder IPC caller to construct and send IPC messages to the remote system services. The returned Java objects and Binder IPC messages were converted into byte sequences as the retrieved artifacts' contents. For Java objects that are not of Java primitive types, we leveraged their `toString` method to acquire more information about them. In this paper, we are particularly interested in Android APIs that do not have any input parameters. According to [22], such APIs are more likely to be "sources that return non-constant values into application code." As a result, we covered approximately 15% of the 1,326 APIs exposed by Android system services.

Android System Properties: Similar to the Windows registry, Android includes a subsystem that centrally stores system configurations and status. This subsystem, usually dubbed as "property system," has been extensively used by Android system services. For example, a system property `ro.kernel.qemu` is read by the Android debugging bridge daemon (adbd) to determine the presence of emulators. System properties also cover meta information about the hardware, such as device models and manufacturers. Despite that SELinux in Android protects system properties, we inspected the implementation of the property system and found that the security check is only in the function `property_set()`, meaning that SELinux does not prevent reading system properties at all. Moreover, applications are allowed to read `/dev/__properties__`, which is the interface to system properties. Therefore, system properties are observable by every installed application. To retrieve system properties, we adapted the artifact retriever to call a binary executable located at `/system/bin/getprop`. It enumerates system properties so that the Java function caller can read the contents of each property. We note that this executable is only for the artifact retriever. It is not required by the detection heuristics that read system properties.

Afterward, we ran the adapted artifact retriever against 16 instances of QEMU-based SDK emulators, 11 instances of VirtualBox-based Genymotion emulators, and 25 real devices. The SDK emulators covered three CPU architectures, namely ARM, x86, and MIPS. The Genymotion emulators covered x86, which is the only architecture they support. Both emulator types covered Android versions from 2.3 to 4.4. The real devices covered four manufacturers (Samsung, HTC, Motorola, and LGE), three ARM SoC families (Qualcomm Snapdragon, Texas Instruments OMAP, and Nvidia Tegra), and Android versions from 2.1 to 4.4. In particular, the real devices were borrowed from the participants we recruited through university mailing lists under the study protocol reviewed by our institution's IRB. Anecdotally, it took approximately 5-20 minutes for the artifact retriever to retrieve and upload the observable artifacts on each device.

The retrieved artifacts contributed to four pools for QEMU-based emulators (E_Q-Pool), VirtualBox-based emulators (E_V-Pool), all the emulators (E-Pool), and real devices (D-Pool). We then fed these pools to the heuristic extractor and the heuristic selector. The heuristic selector ranked the candidate heuristics with 10,000 decision trees and pruned the heuristics with zero importance scores. Table 1 shows a breakdown of the discovered 10,632 detection heuristics. In the remainder of this paper, we will respectively refer to these three categories of heuristics as *file* heuristics, *API* heuristics, and *property* heuristics, in the interest of brevity.

Table 1: Discovered Detection Heuristics

Pools	Detection Heuristics			
	File	API	Property	Total
D-Pool + E-Pool	2,121	81	82	2,284
D-Pool + E_Q-Pool	2,961	163	132	3,256
D-Pool + E_V-Pool	4,782	150	160	5,092
Total	9,864	394	374	10,632

4.2 Characterizing Detection Heuristics

Next, we characterize the discovered heuristics based on the discrepancies they exploit. We first discuss the common detection heuristics that exploit the discrepancies shared by both QEMU-based and VirtualBox-based emulators. We then discuss the heuristics that leverage the QEMU-specific or VirtualBox-specific discrepancies, respectively. Our discussion does not aim to be exhaustive, instead we attempt to convey the scope of discrepancies in Android emulators. Considering that an attacker can possibly use this section as hints to craft detection heuristics, we suggest provisional but deployable countermeasures in Section 6.

4.2.1 Common Detection Heuristics

Network. These detection heuristics exploit the discrepancies in network interfaces, Netfilter modules, and kernel modules. For example, we found that all the emulators exclusively use `eth0`, whereas the real devices use `wlan0` and `rmnet`. The emulators also miss several IPv6-specific interfaces. In addition, the network interfaces in the emulators are not tetherable, because the emulators are missing the Remote Network Driver Interface (RNDIS) drivers that enable tethering. Netfilter is another source of discrepancies. The real devices include Netfilter modules for several network protocols that are rarely used in the context of mobile devices. Finally, Android introduces a kernel module to track data usage of installed applications. This module does not exist in the emulators.

Power management. This type of heuristics focuses on the power management subsystem. For example, the emulators lack the voltage and current regulators. The emulated CPU does not support frequency scaling. Another heuristic lies in the prevalence of multi-core CPUs in real devices. All the emulators only have a single core, whereas 75% of the real devices have at least two cores.

Audio. A handy feature of Android is headset detection, which allows the audio output to automatically switch between speakers and headsets. This feature is supported by GPIO/I2C buses. Notably, the emulators do not emulate these buses, while 95.6% of the real devices in our experiments have them. Furthermore, the differences in the implementation of audio subsystems between the emulators and real devices result in disparate audio drivers.

USB. Recently, USB On-The-Go (OTG) has been widely adopted in popular Android phones and tablets. It allows mobile devices to act as hosts and control USB peripherals. Intuitively, the mobile devices have to pre-install corresponding drivers of USB peripherals. As a result, we found that the real devices in our experiments carry drivers for Apple Magic Mouse, joysticks, and external displays. On the contrary, the emulators do not have such drivers and do not support USB OTG.

Radio. The software-emulated radio can lead to detection heuristics as well. For instance, the name of the baseband in all the emulators is "`unknown`." Moreover, the emulators use a default reference implementation of the radio interface layer (RIL), while the real devices typically use customized ones with different names. Similarly, the phone numbers, voicemail numbers, device serial numbers of the emulators are also constants and can be fingerprinted.

Software components and configurations. Despite that most of the discovered detection heuristics are related to hardware, we also identified several heuristics that exploit certain software components and their configurations. For example, the emulators use unique input methods and search interfaces. Regarding configurations, a prominent example is the key that signs the Android OS. The emulators use test keys while the real devices use release keys.

4.2.2 QEMU Detection Heuristics

QEMU. We found various observable artifacts that are part of QEMU. For example, we found a device node that accelerates the virtual graphics. In addition, there are several system properties set by QEMU and read by Android system services. An example is a property that stores the pixel density of virtual screens.

Goldfish virtual hardware. Most existing QEMU-based Android emulators are built upon a virtual hardware platform called "Goldfish." This platform introduces a set of virtual hardware for QEMU to run Android as its guest operating system. For instance, this set of virtual hardware includes a framebuffer, an audio device, and a battery. They are a must for QEMU-based emulators but never appear in real Android devices.

Bluetooth, NFC, and vibrator. The current QEMU-based emulators do not support these hardware. Their corresponding Android APIs return null if called from within the emulators. In particular, the driver of the vibrator is based on a Linux driver model called `timed_output`, which is also missing from the emulators.

4.2.3 VirtualBox Detection Heuristics

VirtualBox. Similar to QEMU-based emulators, we also found plenty of VirtualBox-specific artifacts. For example, we found 4 kernel modules that belong to VirtualBox Guest Additions. As stated in VirtualBox's documentation, these modules "optimize the guest operating system for better performance and usability." However, their presence also indicates VirtualBox.

PC hardware. As we have discussed, QEMU-based emulators lack support for some popular hardware, such as Bluetooth and NFC. On the contrary, VirtualBox-based emulators support many types of hardware that Android does not need. We found hundreds of artifacts that indicate PC hardware and obviously should not appear in mobile operating systems. For example, we found artifacts related to ACPI, CPU fans, thermal sensors, CD-ROM drives, AC97 audio codecs, and PCI Express.

5. MEASURING DETECTION HEURISTICS

As we have demonstrated the magnitude of the detection heuristic for Android emulators, we further measure their accuracies. To this end, we assembled a group of the top-ranked detection heuristics which are ranked by the heuristic selector. We then tested them against emulator-based malware analysis tools and real devices. In this section, we describe our experiments along with an empirical study on the average accuracies.

5.1 Experimental Setup

As the generated common detection heuristics were already ranked by the heuristic selector, we selected the top 10 heuristics out of the **F**ile, **A**PI, and **P**roperty detection heuristics, respectively. Table 2 lists the artifacts, tokens, and types of the 30 selected detection heuristics.

We created a synthetic application[3] to simulate the Android malware as we described in the threat model. Specifically, this application integrated the 30 heuristics with a heuristic matching engine based on Java's substring searching methods. It generated its decision using a majority vote among the 30 heuristics. In other words, an Android device is recognized as an emulator if more than half of the detection heuristics indicate so. Furthermore, it only needed four permissions: `READ_PHONE_STATE`, `ACCESS_NETWORK_STATE`, `ACCESS_WIFI_STATE`, and `INTERNET`, which are also frequently requested by benign applications in Google Play [29].

[3] `https://play.google.com/store/apps/details?id=edu.sefcom.devicetester` and `http://goo.gl/FXspKw`

Table 2: Top 10 File, API, and Property Heuristics

	Artifact	Token	Type
F1	/proc/misc	"network_throughput"	E
F2	/proc/ioports	"0ff\0:"	E
F3	/proc/uid_stat		D
F4	/sys/devices/virtual/misc/ cpu_dma_latency/uevent	"MINOR=5"	E
F5	/sys/devices/virtual/ppp		D
F6	/sys/devices/virtual/switch		D
F7	/sys/module/alarm/parameters		D
F8	/sys/devices/system/cpu/ cpu0/cpufreq		D
F9	/sys/devices/virtual/misc/ android_adb		D
F10	/proc/sys/net/ipv4/ tcp_syncookies		E
A1	isTetheringSupported()	"false"	E
A2	getAuthenticatorTypes()	"AuthenticatorDescription {type=com.g}"	D
A3	getSystemSharedLibraryNames()	"com.g"	D
A4	getGlobalSearchActivity()	".android.quicksearchbox/ com.android.quicksearchbox"	E
A5	getWebSearchActivity()	".android.quicksearchbox/ com.android.quicksearchbox"	E
A6	getTetherableWifiRegexs()	"wlan0\n"	D
A7	getTetherableUsbRegexs()	"rndis"	D
A8	getEnabledInputMethodList()	".android.inputmethod .latin/."	E
A9	getDeviceId() via Binder	"\0\0\03"	D
A10	getTetherableIfaces()	"wlan0"	D
P1	qemu.hw.mainkeys		E
P2	ro.build.description	"release-keys"	D
P3	ro.build.fingerprint	":user/release-keys"	D
P4	net.eth0.dns1		E
P5	rild.libpath	"/system/lib/libreference-ril.so"	E
P6	ro.radio.use-ppp		E
P7	gsm.version.baseband		D
P8	ro.build.tags	"release-key"	D
P9	ro.build.display.id	"test-"	E
P10	init.svc.console		E

Table 3: Evaluated Emulators and Real Devices

Emulators (9)	DroidBox [5] 2.3 and 4.1, Andrubis [6], CopperDroid [23], SandDroid [3], TraceDroid [4], Qihu 360, NVISO ApkScan, ForeSafe
Real Devices (128)	Samsung, HTC, LGE, Huawei, Motorola, Sony Ericsson, Lenovo, ZTE, Hisense, Asus, Acer, OPPO, BBK, Meizu, Gionee, DOOV, YuLong, Haier, AMOI

We ran this application in 9 emulator-based malware analysis tools and 128 distinct real devices. As shown in Table 3, the malware analysis tools covered two versions of an offline tool called DroidBox and 7 online services. Among the online services, 4 are derived from previous research work and 3 are security products. The 128 real devices were from AppThwack, TestObject, and Baidu MTC, all of which are online services that automatically test applications in real phones and tablets. Note that we did not run our artifact retriever on them due to their limited device minutes and bandwidth quota.

5.2 Results and Empirical Analysis

We deem emulators as *positive* and real devices as *negative*. Given the measured true positives (TP), false negatives (FN), false positives (FP), and true negatives (TN), we attempt to evaluate the detection heuristics with three metrics, namely *sensitivity*, *specificity*, and *accuracy*. For example, a Type E detection heuristic is sensitive if it matches all the emulator instances. And, it is specific if it does not match any non-emulator instances, *i.e.*, real devices. Simply put, we compute the values of the three metrics as follows:

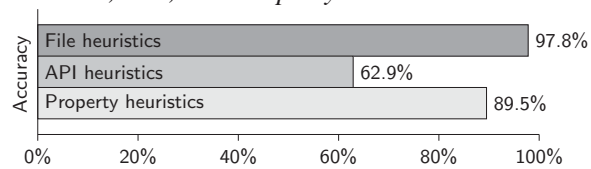
- $Sensitivity = TP/(TP + FN)$;
- $Specificity = TN/(FP + TN)$; and
- $Accuracy = (TP + TN)/(TP + FN + FP + TN)$.

Table 4: Evaluation Results of the 30 Heuristics

	TP	FN	FP	TN	Sens.(%)	Spec.(%)	Acc.(%)
F1	9	0	2	126	100.0	98.4	98.5
F2	9	0	0	128	100.0	100.0	100.0
F3	9	0	7	121	100.0	94.5	94.9
F4	9	0	4	124	100.0	96.9	97.1
F5	9	0	1	127	100.0	99.2	99.3
F6	9	0	1	127	100.0	99.2	99.3
F7	9	0	7	121	100.0	94.5	94.9
F8	9	0	0	128	100.0	100.0	100.0
F9	9	0	0	128	100.0	100.0	100.0
F10	9	0	8	120	100.0	93.8	94.2
A1	9	0	3	125	100.0	97.7	97.8
A2	7	2	82	46	77.8	35.9	38.7
A3	5	4	24	104	55.6	81.3	79.6
A4	7	2	48	80	77.8	62.5	63.5
A5	7	2	45	83	77.8	64.8	65.7
A6	9	0	37	91	100.0	71.1	73.0
A7	9	0	64	64	100.0	50.0	53.3
A8	9	0	37	91	90.0	71.1	72.5
A9	6	3	72	56	66.7	43.8	45.3
A10	9	0	82	46	100.0	35.9	40.1
P1	8	1	2	126	88.9	98.4	97.8
P2	8	1	17	111	88.9	86.7	86.9
P3	8	1	21	107	88.9	83.6	83.9
P4	9	0	5	123	100.0	96.1	96.4
P5	9	0	2	126	100.0	98.4	98.5
P6	9	0	11	117	100.0	91.4	92.0
P7	9	0	14	114	100.0	89.1	89.8
P8	8	1	10	118	88.9	92.2	92.0
P9	8	1	0	128	88.9	100.0	99.3
P10	9	0	20	108	100.0	84.4	85.4

Table 4 demonstrates the measured accuracies of the 30 detection heuristics. We next present our empirical analysis on the average accuracies from three aspects.

5.2.1 *File, API, and Property Heuristics*



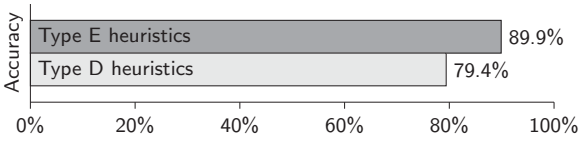
We first inspected the average accuracies of the heuristics according to the categories of their exploited observable artifacts. As shown in the above bar chart and Table 4, the file heuristics enjoyed both high sensitivities and specificities with an average accuracy of 97.8%. The API heuristics, despite of their acceptable sensitivities, suffered from significantly low specificities. For example, A2, A9, and A10 performed no better than 50/50 guesses as their accuracies were less than 50%. The property heuristics performed fairly good with an average accuracy of 89.5%.

One possible explanation for the API heuristics' low accuracies is that the Android APIs are designed to provide some sort of hardware/software abstraction. An evidence is the Android Compatibility Program[4], which precisely defines the behaviors of Android APIs to ensure that Android applications run in "a consistent and standard environment."

[4] `https://source.android.com/compatibility`

To build such an environment, the APIs that reveal the underlying details are not necessary, and they are subject to be removed or deprecated. However, we argue that this environment also requires a well-configured application sandbox to prevent applications from bypassing the APIs. Unfortunately, our discovered file and property heuristics imply that the current sandbox should be reinforced.

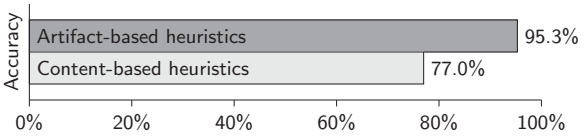
5.2.2 Type E and Type D Heuristics



We investigated the differences between the Type E and Type D heuristics. As we have discussed in Section 3.3, Type E and Type D detection heuristics respectively indicate emulators and real devices. In our experiments, the Type E heuristics outperformed the Type D ones.

We note that almost all of the heuristics in Table 4 with low specificities are of Type D. We believe that it is due to the diversified and fragmented nature of real devices. Type D heuristics expect the artifacts/tokens that are prevalent in real devices. However, device manufacturers inevitably customize devices and change artifacts, which makes it harder to find the artifacts that exist in every real device. On the contrary, emulators are much more unified in terms of customizations, which is possibly due to the difficulty in modifying and maintaining software-emulated hardware.

5.2.3 Artifact-based and Content-based Heuristics



Finally, we compared the artifact-based and content-based heuristics. The bar chart shows that the artifact-based heuristics had an average accuracy of 95.3%. The content-based heuristics fell behind with 77.0%. However, we stress that F1, F4, and P9 are also content-based heuristics and their accuracies were among the top of the 30 heuristics.

In addition to the factors of abstraction and customization that we discussed earlier, a possible explanation is that content-based heuristics are more subtle and vulnerable to intended or unintended changes. Content-based heuristics exploit artifacts' contents (*e.g.*, configurations), which are subject to change in a rapidly evolving system like Android. On the contrary, artifact-based heuristics rely on the presence of certain artifacts. Compared with the contents, the artifacts (*e.g.*, kernel modules) are relatively consistent in emulators and real devices, because developers are usually reluctant to remove them as to avoid unexpected problems.

5.3 Case Study: A9

Finally, we present a case study on heuristic A9 because it involves an Android API `getDeviceId`, which has been popular among the known detection heuristics. A9 exploits the same API but in a slightly different way. Specifically, it looks for a token "`\0\0\03`" in the Binder IPC message returned from the implementation of `getDeviceId` in `TelephonyManagerService`. In other words, A9 uses the IMEIs whose first character is "`3`" to indicate real devices. However, it turned out that A9 had a sensitivity of 66.7% and an unbearable specificity of 43.8%. We investigated the evaluated emulators and real devices to find out how they reacted to A9. The investigation led us to flaws in an existing anti-detection technique and improvements for A9.

5.3.1 A9 vs DroidBox 4.1

As we discussed in Section 4.1, our implementation of the artifact retriever employs a Java function caller and a Binder IPC caller to probe Android APIs. When we tested A9 against DroidBox 4.1, we found that these two callers returned disparate values, namely "`357242043237511`" and "`000000000000000`." We note that both callers should return the same value, because the application-side Binder proxies of Android APIs are not supposed to modify the IPC messages returned by the underlying Binder stubs in system services.

We inspected DroidBox's source code and found that DroidBox rewrites the Binder proxy of `getDeviceId` to return a dummy IMEI without involving the Binder stubs. Although such a countermeasure could neutralize the detection heuristics that call `getDeviceId` in a normal way, it is not effective against the Binder IPC caller, which bypasses the countermeasure and reads the actual full-zero IMEI. Therefore, we believe that the countermeasure implemented in DroidBox 4.1 is not complete. We note that ApkScan demonstrated the same issue, implying that ApkScan might have integrated DroidBox 4.1 for its dynamic analysis.

However, A9 was not effective against DroidBox 2.3. We found that DroidBox 2.3 opts for a similar countermeasure but implements it in the service-side Binder stub. In such a case, bypassing the stub and observing the actual IMEI would require root privileges, *i.e.*, the actual IMEI is not observable. Therefore, such a countermeasure is effective and the dummy IMEI would appear realistic.

5.3.2 A9 vs Non-U.S. Devices

A9 assumes that an Android device whose IMEI starts with "`3`" is a real device, otherwise it is an emulator. We checked the IMEIs of the 128 real devices and found that this assumption is incorrect.

According to IMEI Allocation and Approval Guidelines [7], the first digit of an IMEI is part of Reporting Body Identifier (RBI), which identifies the GSMA-approved authority that issues the IMEI. Typically, IMEIs of mobile devices are issued by the authorities in the same area where the devices are sold. For example, IMEIs of the devices sold in the U.S. are issued by the British Approvals Board for Telecommunications (BABT) and thus start with BABT's code "`35`." Similarly, IMEIs of the devices sold in China start with "`86`." We note that about half of the 128 evaluated real devices were from Baidu MTC that uses Android phones sold in China. Given that A9 was based on the devices in the U.S., A9 naturally got a low specificity, and it could be improved with wild cards that match multiple RBIs.

The lesson of A9 indeed illustrates the future of the armed race between emulator detection and anti-detection. First, Android malware could check the semantics of the observed artifacts. For example, the dummy IMEI in DroidBox 4.1 is invalid and could be noticed by a sophisticated adversary. Second, emulator-based malware analysis tools should consider the observability of actual artifacts and the semantics of dummy artifacts to be less distinguishable.

6. DISCUSSION

The evaluation results imply an imminent threat that Android malware may thwart existing emulator-based dynamic analysis systems. In this section, we suggest the potential countermeasures and discuss the limitations of our work.

6.1 Countermeasures

Provisional countermeasures. We suggest the methods that detect the usage of detection heuristics in Android malware as provisional countermeasures. Although they do not prevent Android malware from detecting Android emulators, they can raise alarms for analysts and thus thwart the malware's original purpose of evading analysis. For example, dynamic analysis systems could monitor accesses on files and properties seldom used by benign applications. API heuristics are much more stealthy because benign applications also frequently use them. In such a case, we suggest static data-flow analysis to locate branches that involve detection heuristics and lead to disparate code blocks.

Short-term countermeasures. Next, we discuss the countermeasures that allow an emulator to appear "realistic" to Android malware. First, we suggest a comprehensive deployment of dummy artifacts. Some existing works can be adapted to facilitate such countermeasures. For example, AirBag [26] supports a decoupled and isolated runtime environment based on OS-level virtualization. ASM [13] provides programmable interfaces that interpose Android APIs and return dummy values to applications. These works, if combined and extended, can enable a "brain in a vat" setup where an application runs in an emulator but gets dummy and valid data originated from real devices. Second, we suggest denying accesses on unnecessary observable artifacts with strict DAC and MAC policies. For example, artifacts in sysfs exploited by our file heuristics seem unnecessary for general Android applications. However, the usability impact of denying accesses still needs further verification.

Long-term countermeasures. The ideal countermeasure is to fix all the discrepancies in Android emulators. Although Garfinkel *et al.* [12] concludes its infeasibility in 2007 due to the inherent hardness of creating indistinguishable software-emulated hardware, hardware-assisted virtualization techniques (*e.g.*, Intel VT-x and VT-d) have evolved significantly to allow PC emulators to virtualize real hardware. Currently, ARM CPUs have integrated necessary virtualization extensions. Meanwhile, commodity ARM hypervisors are also in active development. We envision emerging Android emulators equipped with virtualized CPUs, sensors, and radios in the near future.

6.2 Limitations

Despite the robustness of Morpheus, the quality of the discovered detection heuristics is limited by the small number of real devices used in finding detection heuristics (Section 4). In general, Morpheus works like supervised learning, and its performance inevitably depends on the quality of the "training set," *i.e.*, the emulators and real devices observed by the artifact retriever. We note that the artifact retriever needs approximately 20 minutes to collect the artifacts on a single device. Unfortunately, online services like AppThwack (Section 5) do not allow the artifact retriever to run for such a long time or upload large bulks of data. As for future work, we plan to reach out to mobile carriers and device vendors to collect observable artifacts from more real devices.

Although Morpheus discovered more than 10,000 heuristics, we stress that they were derived only from 3 out of 33 sources of observable artifacts. To better understand the scope of detection heuristics for effective countermeasures, the artifact retriever could be enhanced to address more sources of artifacts as well as sophisticated usages of them. Examples include extended modules of the artifact retriever that can handle callbacks or construct valid input parameters for Android APIs. We did not cover them in this work because they require domain-specific knowledge of each Android system service.

Our heuristic generator produces relatively rigid heuristics, such as A9 that does not match multiple RBIs. This can be improved with more sophisticated and flexible heuristics. For example, a token-sequence heuristic matches an ordered set of tokens in the contents of an artifact. Moreover, a naïve Bayes heuristic enables probabilistic matching by aggregating the empirical probabilities of multiple artifact/token heuristics with the Bayes' law, assuming that the occurrences of artifacts/tokens are independent.

7. RELATED WORK

Behavior-based detection heuristics. Researchers have proposed several heuristics that exploit discrepancies in runtime behaviors rather than artifacts. For instance, a piece of specially crafted native code can identify QEMU-based emulators due to the discrepancies in QEMU's caching behaviors [19, 21, 24]. Low video frame rate indicates emulators because of the performance drawbacks in the SDK emulator's graphics rendering engine [25]. However, these heuristics are not evaluated against VirtualBox-based emulators and real devices. Thus, their sensitivities and specificities require further investigation. In addition, these heuristics do not return a decision until a sufficient number of events are observed, which tends to increase their footprints and attract analysis. Along these lines, Morpheus addresses artifact-based and content-based detection heuristics. More importantly, Morpheus generates detection heuristics automatically and systematically.

Dynamic analysis frameworks. Researchers have built several dynamic analysis frameworks to vet the runtime behaviors of Android malware. TaintDroid [10] tracks information flows that leak sensitive data to the Internet. VetDroid [28] further reveals information flows that involve permissions. AppIntent [27] helps determine if an information flow is user-intended. Some of these tools have been integrated into automated malware analysis systems such as DroidBox [5], Andrubis [6], CopperDroid [23], SandDroid [3], and TraceDroid [4]. They are vulnerable to be evaded using the detection heuristics in this work as long as they are deployed in Android emulators.

8. CONCLUSION

Recent Android malware demonstrates the capabilities of detecting Android emulators using detection heuristics. To convey the severity of this problem, we have presented Morpheus, a system that automatically and systematically generates detection heuristics. Morpheus analyzes artifacts observable by Android applications and discovers exploitable discrepancies in Android emulators. Moreover, we have described a proof-of-concept implementation of Morpheus, along with extensive experiments and findings.

Acknowledgements

We would like to thank Adam Doupé and the anonymous reviewers for their valuable comments that helped improve the presentation of this paper. This work was supported in part by the National Science Foundation and National Research Foundation. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not reflect the views of the funding agencies.

9. REFERENCES

[1] Android developers - using the emulator. http://developer.android.com/tools/devices/emulator.html. Accessed: May 2014.

[2] Genymotion, the fastest android emulator for app testing and presentation. http://genymotion.com. Accessed: May 2014.

[3] Sanddroid - an apk analysis sandbox. http://sanddroid.xjtu.edu.cn/. Accessed: May 2014.

[4] Tracedroid - dynamic android app analysis (by vu amsterdam). http://tracedroid.few.vu.nl/. Accessed: May 2014.

[5] Droidbox: An android application sandbox for dynamic analysis. https://code.google.com/p/droidbox/, 2011. Accessed: May 2014.

[6] Andrubis: A tool for analyzing unknown android applications. http://blog.iseclab.org/2012/06/04/andrubis-a-tool-for-analyzing-unknown-android-applications-2/, June 2012. Accessed: May 2014.

[7] G. Association et al. Imei allocation and approval guidelines. volume 10, 2010.

[8] D. Balzarotti, M. Cova, C. Karlberger, C. Kruegel, E. Kirda, and G. Vigna. Efficient detection of split personalities in malware. In *Proceedings of Network and Distributed System Security Symposium*, 2010.

[9] H. Dharmdasani. Android.hehe: Malware now disconnects phone calls. http://www.fireeye.com/blog/technical/2014/01/android-hehe-malware-now-disconnects-phone-calls.html, January 2014. Accessed: May 2014.

[10] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. In *Proceedings of the USENIX conference on Operating systems design and implementation*, pages 1–6. USENIX, 2010.

[11] F-Secure. Trojan:android/pincer.a. http://www.f-secure.com/weblog/archives/00002538.html, April 2013. Accessed: May 2014.

[12] T. Garfinkel, K. Adams, A. Warfield, and J. Franklin. Compatibility is not transparency: Vmm detection myths and realities. In *Proceedings of USENIX Workshop on Hot Topics in Operating Systems*, 2007.

[13] S. Heuser, A. Nadkarni, W. Enck, and A.-R. Sadeghi. Asm: A programmable interface for extending android security. In *Proceedings of the USENIX Security Symposium*, 2014.

[14] T. K. Ho. The random subspace method for constructing decision forests. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 20(8):832–844, 1998.

[15] C. Ionescu. Obfuscating embedded malware on android. http://www.symantec.com/connect/blogs/obfuscating-embedded-malware-android, June 2012. Accessed: May 2014.

[16] Z. Li, M. Sanghi, Y. Chen, M.-Y. Kao, and B. Chavez. Hamsa: Fast signature generation for zero-day polymorphic worms with provable attack resilience. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 15–pp. IEEE, 2006.

[17] H. Lockheimer. Android and security. http://googlemobile.blogspot.com/2012/02/android-and-security.html, February 2012. Accessed: May 2014.

[18] I. Lunden. Gartner: 102b app store downloads globally in 2013, 26b in sales, 17% from in-app purchases. http://techcrunch.com/2013/09/19/gartner-102b-app-store-downloads-globally-in-2013-26b-in-sales-17-from-in-app-purchases/, September 2013. Accessed: May 2014.

[19] F. Matenaar and P. Schulz. Detecting android sandboxes. http://dexlabs.org/blog/btdetect, August 2012. Accessed: May 2014.

[20] J. Oberheide and C. Miller. Dissecting the android bouncer. *SummerCon2012, New York*, 2012.

[21] T. Petsas, G. Voyatzis, E. Athanasopoulos, M. Polychronakis, and S. Ioannidis. Rage against the virtual machine: hindering dynamic analysis of android malware. In *Proceedings of the European Workshop on System Security*, page 5. ACM, 2014.

[22] S. Rasthofer, S. Arzt, and E. Bodden. A machine-learning approach for classifying and categorizing android sources and sinks. In *Proceedings of the Network and Distributed System Security Symposium*, 2014.

[23] A. Reina, A. Fattori, and L. Cavallaro. A system call-centric analysis and stimulation technique to automatically reconstruct android malware behaviors. In *Proceedings of the European Workshop on System Security*, April 2013.

[24] P. Schulz. Android emulator detection by observing low-level caching behavior. https://bluebox.com/technical/android-emulator-detection-by-observing-low-level-caching-behavior/, December 2013. Accessed: May 2014.

[25] T. Vidas and N. Christin. Evading android runtime analysis via sandbox detection. In *Proceedings of the ACM Symposium on Information, Computer and Communications Security*. ACM, 2014.

[26] C. Wu, Y. Zhou, K. Patel, Z. Liang, and X. Jiang. Airbag: Boosting smartphone resistance to malware infection. In *Proceedings of the Network and Distributed System Security Symposium*, 2014.

[27] Z. Yang, M. Yang, Y. Zhang, G. Gu, P. Ning, and X. S. Wang. Appintent: Analyzing sensitive data transmission in android for privacy leakage detection. In *Proceedings of the ACM conference on Computer and communications security*, pages 1043–1054. ACM.

[28] Y. Zhang, M. Yang, B. Xu, Z. Yang, G. Gu, P. Ning, X. S. Wang, and B. Zang. Vetting undesirable behaviors in android apps with permission use analysis. In *Proceedings of the ACM conference on Computer and communications security*, pages 611–622. ACM, 2013.

[29] Y. Zhou and X. Jiang. Dissecting android malware: Characterization and evolution. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, pages 95–109. IEEE, 2012.

[30] Y. Zhou, Z. Wang, W. Zhou, and X. Jiang. Hey, you, get off of my market: Detecting malicious apps in official and alternative android markets. In *Proceedings of the Network and Distributed System Security Symposium*, pages 5–8, 2012.

Design and Implementation of an Android Host-based Intrusion Prevention System

Mingshen Sun[*] [†], Min Zheng[†], John C.S. Lui[†], Xuxian Jiang[§]
[†] The Chinese University of Hong Kong, [§] Qihoo 360 & NC State University
{mssun, mzheng, cslui}@cse.cuhk.edu.hk, jiang@cs.ncsu.edu

ABSTRACT

Android has a dominating share in the mobile market and there is a significant rise of mobile malware targeting Android devices. Android malware accounted for 97% of all mobile threats in 2013 [26]. To protect smartphones and prevent privacy leakage, companies have implemented various *host-based intrusion prevention systems* (HIPS) on their Android devices. In this paper, we first analyze the implementations, strengths and weaknesses of three popular HIPS architectures. We demonstrate a severe loophole and weakness of an existing popular HIPS product in which hackers can readily exploit. Then we present a design and implementation of a secure and extensible HIPS platform — "*Patronus*". Patronus not only provides intrusion prevention without the need to modify the Android system, it can also dynamically detect existing malware based on runtime information. We propose a two-phase dynamic detection algorithm for detecting running malware. Our experiments show that Patronus can prevent the intrusive behaviors efficiently and detect malware accurately with a very low performance overhead and power consumption.

1. INTRODUCTION

We now live in a mobile digital era and smartphones are becoming indispensable as they are being used as personal communication and computing devices. Unfortunately, smartphones are also becoming hackers' biggest target [44]. Android is an open source operating system for mobile devices. Its market share has reached around 80 percent of all smartphone shipments in the second quarter of 2013 [35]. Due to its large market share, it becomes the major target for hackers. A recent report [26] indicated that 97 percent of these threats targeted Android devices. Furthermore McAfee [39] showed that 17,000 new Android malware were found in the second quarter of 2013. All these indicate that Android devices are facing an explosive growth of malware threat.

Permission mechanism is the primary security protection in Android. When an Android user installs a mobile application (app for short), the system will alert the user about the required permissions of that app. But this mechanism does not provide an adequate level of security. As shown in [28, 31, 38, 16], there are plenty of Android permission abuses. Due to the working mechanism and the open source nature of Android system, it is easy to write malware variants which can bypass the detection of anti-virus software. Recently, researchers [59, 48] have shown that some of the existing anti-virus software cannot effectively detect malware variants which were generated by simple transformations. Most of the existing anti-virus engines are using static analysis and malware signatures [62, 32] for detection. To make up for the limitations of static analysis, several systems [24, 55, 8] aim to detect the *runtime behaviors* by using dynamic analysis. However, these systems are mainly for post-mortem analysis and with a high likelihood, the malware under investigation may have already infected many devices. Furthermore, for these dynamic analysis systems, how to automatically trigger the malicious behaviors is still an ongoing research problem [58, 47]. Therefore, the security community has been advocating the use of a *host-based intrusion prevention system* (HIPS), which is an installed software on a mobile device to monitor suspicious activities, and to block and report malicious behaviors by analyzing events at realtime. HIPS can be installed on mobile devices to provide protection against apps which have runtime malicious behaviors. HIPS can also dynamically intercept the apps at runtime and notify users when the malware invoke some dangerous application program interfaces (APIs). In fact, the use of HIPS is gaining popularity and has been implemented in products like Jinshan Mobile Duba [9], LBE [10], 360 Mobile Safe [13], etc.

In general, there are three approaches to implement HIPS in Android, they are: (1) *system patching*, (2) *application repackaging* and (3) *API hooking*. System patching is to modify the Android operating system with new permission management functions. Application repackaging is to disassemble a mobile app, add new policy enforcement to it, and then repackage the mobile app as a new app. API hooking is to intercept mobile app's API calls at runtime so as to inspect malicious behaviors. Each of these approaches has its own limitations and may bring new vulnerabilities. In Section 3, we present in detail the implementation issues as well as weaknesses of these three popular HIPS implementations.

Given the limitations of static and/or dynamic analysis, as well as weaknesses of existing HIPS products, we propose

[*] Part of this work was done during his internship at Qihoo 360.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.
ACSAC '14, December 08–12 2014, New Orleans, LA, USA
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-3005-3/14/12 ...$15.00
http://dx.doi.org/10.1145/2664243.2664245

an enhanced HIPS called *Patronus*, which not only performs a secure policy enforcement, but can also dynamically detect existing malware using runtime information. To guarantee the ease of deployment, Patronus does *not* require any modifications on the Android firmware or mobile apps. Patronus performs runtime policy enforcement on the system level to inspect malicious behaviors. Moreover, Patronus provides a host-based runtime detection which can halt malicious behaviors execution. We make the following contributions:

- To the best of our knowledge, this is the first work which systematically analyzes three popular frameworks of HIPS and exposes various security vulnerabilities of these existing HIPS architectures. We also illustrate how to exploit the vulnerability of a popular HIPS product so as to bypass malware detection.
- We design and implement a secure architecture Patronus, which can prevent mobile malware intrusions and can detect malware at runtime. Patronus addresses the security issues we reveal in the current HIPS products and effectively prevent intrusions.
- We design and implement a two-phase detection algorithm based on the runtime intrusion information to detect existing malware. The algorithm can determine and prevent malware at runtime (online), while traditional signature-based static method can only achieve this in an offline manner.

The rest of the paper is organized as follows: Section 2 introduces the necessary background on Android. In Section 3, we present the strengths and weaknesses of three popular HIPS implementations. In Section 4, we propose our Patronus system, and describe the method to prevent intrusions as well as proposing a two-phase algorithm for dynamic malware detection. In Section 5, we present our experimental results to illustrate the effectiveness and performance of Patronus. Section 6 presents the related work and the conclusion is given in Section 7.

2. BACKGROUND

In this section, we introduce the architecture of Android system, its binder inter-process communication mechanism, as well as the Dalvik Virtual Machine and the Native Development Kit in Android. This background is essential to understand how one can create a HIPS for Android devices.

2.1 Android Architecture

Android is an open source operating system for mobile devices developed by Google. The system consists of five functional layers: (1) *kernel*, (2) *libraries*, (3) *runtime support*, (4) *application framework* and (5) *applications*. Figure 1 presents the Android architecture. Android utilizes Linux as the core kernel with various drivers for hardware communication. The libraries layer contains native libraries such as `libc` or `OpenGL` so as to support higher application layers. The runtime layer consists of the Dalvik Virtual Machine (DVM) with various runtime libraries. DVM is a special Java virtual machine to execute Android apps. Application framework layer contains basic services to provide activity management, SMS management, etc. Lastly, all Android apps are running on top of these layers.

Android apps are mostly written in Java using the Android SDK , and DVM is responsible for interpreting and executing these apps. Each app is running within its own DVM. The Android system assigns a unique ID for each

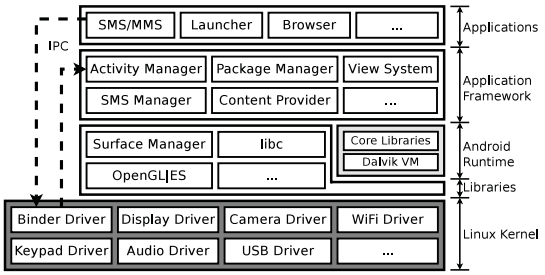


Figure 1: Architecture of Android system.

DVM to achieve process (or app) isolation. This sandboxing mechanism provides the basic security protection between apps. Android also provides a special inter-process communication mechanism called the *Binder*. Each app has to go through the Binder (which is in the kernel layer) so as to communicate with other apps or services.

Besides using DVM sandbox, permission is another security mechanism provided by Android. Only when an app has explicitly declared its permissions in its `AndroidManifest.xml` file, then the app can use the corresponding APIs. Moreover, when installing an app, the system will display a dialog window to alert users about the declared permissions of that app. However, most users usually accept these permission alerts without paying much attention at installation. Once the installation is completed, the system will not warn users in later invocations. Many Android malware take advantage that most users are negligent and simply accept the permission alerts, so the malware can often obtain enhanced permissions (e.g., sending premium SMS messages to subscribe chargeable service in the background).

2.2 Binder Mechanism

Binder is a specialized inter-process communication (IPC) mechanism in Android. Since apps are running in their own DVM sandboxes, they need to communicate through the Binder so as to utilize others' services. Figure 2 shows the basic flow of the Binder mechanism. For instance, if an app wants to send an SMS message, it should first (1) contact the `Service Manager` which contains the information of all registered services. The `Service Manager` will provide a handler to communicate with the `ISms Service` which is responsible for sending SMS messages. Once the app has the handler, it can (2) ask the `ISms Service` to send SMS messages, then (3) the `ISms Service` will process the request and send the message through the SMS Driver. Note that all communications have to go through the Binder by sending transactions with the required information (i.e., the parcel). Transaction is a communication procedure between two processes. In Android, the Binder transaction is used to send service requests (which is represented by a transaction code) to the corresponding processes. There are two stages to complete a transaction. First, the Binder will deliver a data parcel to the destination process containing the receiver information (i.e., transaction descriptor). Secondly, after completing the request, the received process will save the result in a reply parcel. In the above example of sending an SMS message, there are two transactions and they are completed in three steps. The first transaction requests for the ISms Service handler. The second transaction requests for sending an SMS message. In the second transaction, the data parcel contains the ISms Service descriptor (`com.android.internal.telephony.ISms`) and information

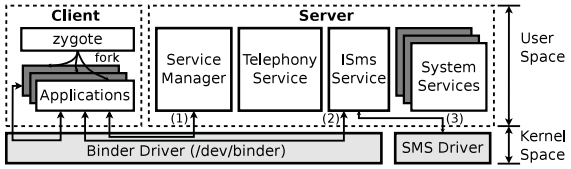


Figure 2: Binder IPC mechanism.

(e.g., destination address and text content) for sending an SMS message. In the third step, the `ISms Service` sends the request and saves the result in a reply parcel.

Note that this procedure is similar to the client-server communication model. Apps are on the client side and they are executing within their own sandboxes, while various services are on the server side and they are managed by the `Service Manager`. The `System Services` are a group of services related to the Android system (e.g., Account Manager Service, Battery Service and Network Management Service). Moreover, an app can export its service to the public by registering to the `Service Manager`. This special Binder mechanism not only provides a flexible inter-process communication framework, but also isolates apps and services so as to protect the system.

2.3 DVM & Native Development Kit

Dalvik Virtual Machine (DVM) is a special component in the Android system. The apps are written in Java and compiled into a bytecode file (i.e., `class` file) with Android SDK. These bytecodes will be converted to Dalvik executable files (i.e., `dex` file) before installation. To instantiate an app, the DVM will interpret and run the app in a *different* virtual machine. In order to integrate with other C/C++ native libraries, Android provides the Native Development Kit (NDK) for developers to implement parts of the app using native code such as C/C++. Moreover, Java Native Interface (JNI) provides a bridge between the Java code and C/C++ code so developers can invoke native code easily. Therefore, for the system, DVM can use native libraries (e.g., `OpenGL`, `libbinder`, `libc`) to support its execution.

3. HIPS

In this section, we discuss three popular frameworks to implement host-based intrusion prevention systems (HIPS): (1) *system patching*, (2) *application repackaging* and (3) *API hooking*. In particular, we present their implementations, strengths and weaknesses. We also demonstrate how to bypass a popular HIPS product.

3.1 System Patching

Before Android 4.3, there was no HIPS integrated in the Android OS, and users have experienced various forms of permission abuses since malware can take advantage of the system loopholes to achieve permission escalation. Because Android is an open source project, researchers and companies have developed patches for the Android system [53]. In fact, some third-party system images [7] have integrated with these patches and they can manage the permission usage for each app. Some mobile phone vendors (e.g., Huawei) also provide patched images for their phones and users can install these modified images on their devices. In Android 4.3, Google also provides system patching. For example, there is a hidden function called `App Ops` [45], which is a form of HIPS and by using `App Ops`, one can disable certain permissions of an app. However, this `App Ops` has been removed in Android 4.4.

To install these HIPS in Android, users need to update to Android 4.3, or install the patched system image from their vendors. However, as shown in the latest report [30], only 10% of Android phones are powered by Android 4.3 or 4.4. Furthermore, if users want to install any patched image, they have to find the appropriate third-party system images which match with their phone models. For this reason, many users often opt *not* to install these patches. These imply that there are a large percentage of smartphones which are vulnerable. Moreover, because one has to use `App Ops` to disable permission before launching the apps, systems like `App Ops` cannot prevent intrusions at runtime. Hence, even with system patches, many Android phones are still vulnerable to malware attack.

3.2 Application Repackaging

Android application package file (or `apk` file) is a zip file which contains Dalvik executable file (`classes.dex`) and other resource files (e.g., images or audio). An app needs to declare its permission usage in the `AndroidManifest.xml` file within the `apk` package so as to use the corresponding APIs. Thus, to prevent an app using sensitive APIs, one can delete certain permission declarations in the `AndroidManifest.xml` and repackage it as a new `apk` file. App Shield [5] is an app to manage app permission using this technique.

Since the Dalvik executable file is converted from the Java bytecode, it is easy to be reversed to readable codes. In fact, several tools [14, 4, 25] provide assembling and disassembling functions on `dex` files. Hence, one can disassemble the `dex` file into readable codes, modify the program logic and then assemble and repackage it into a new `apk` file. By using this repackaging method, HIPS can be implemented by inserting *stubs around sensitive Android APIs* so as to perform policy enforcement without modifying the Android system. For instance, if an app needs to get the current location by utilizing the GPS function, it has to use the `request LocationUpdates` API. So one can add code into the app to check the GPS permission before invoking this API. Aurasium [54] and RetroSkeleton [21] are two projects using this form of application repackaging to inspect sensitive APIs to reinforce the Android permission policy.

Although application repackaging does not need to modify the Android system, there are some major drawbacks. The first major drawback is on incomplete security coverage. Since a function can be implemented by different approaches (i.e., calling different APIs), so it is possible to miss policy enforcement unless we know *all possible APIs* in realizing a function. For example, using the Java Reflection, one can invoke APIs and bypass the stubs. Secondly, disassembling technique can only work on `dex` file. If application developers use native code (e.g., binary libraries written in C or C++) to implement some functionalities, application repackaging cannot monitor these native library calls. Hence, application repackaging cannot fully prevent malware intrusion. Furthermore, malware writers can exploit bugs in the disassembling tools [50, 3] to conceal their malicious functions. In summary, application repackaging *cannot* fully prevent malware intrusion.

3.3 API Hooking

Generally, API hooking is to intercept API calls in order

to inspect the behaviors of a calling app. For Android, apps must go through the Binder to call other services, so the hooking method can be implemented on the Binder communication. The basic flow of using API hooking to implement HIPS can be summarized as: (1) gaining root or system privilege; (2) injecting a shared library object file (i.e., `so` file) to the target process; (3) carrying out hooking on target APIs; (4) loading policy enforcement function. Let us describe in detail the Android API hooking process.

3.3.1 Overview of API Hooking

Due to the sandboxing protection, API hooking requires root or system privilege to hook on functions in the libraries of the target processes. There are several approaches [1, 29] to gain root or system privileges. Basically, these tools exploit the Android system and get root or system privileges. In order to ensure the security, the tools will also install an app called Superuser on the phones to manage the authorizations of higher privileges to certain apps. Furthermore, the system loopholes can be easily patched by using hooking methodology [41]. Hence, malware cannot exploit the loopholes to infect system.

The hooking operation has to be conducted in the native code. To achieve this, we inject a shared library file (`so` file) to achieve the hooking. Inline hooking [34] and `ptrace` are two methods to inject an `so` file. However, the implementation of inline hooking on ARM platform is more difficult than on the x86 platform. Therefore, we use `ptrace` to attach the target process so as to modify the registers in the target process to execute our shellcode. In the shellcode, we utilize `dlopen` and `dlsym` functions to inject an `so` file into the target memory. The shellcode will also invoke an entry function in the injected `so` file to carry out API hooking.

We can use the API hooking on Android to intercept any Java method. In particular, we can modify the global object (`gDvm`) in the `libdvm.so` library of DVM to intercept a target method. `gDvm` maintains the structure of every class and method at runtime. There is a variable called `insns` in `gDvm` for each method specifying the address of the corresponding Java method. So, we can find the target method in the loaded classes (`gDvm->loadedClasses`) and replace the method `insns` to point to the function in the injected `so`'s library. Hence, when one calls the intercepted APIs, the DVM will execute our own methods. After executing our methods, we can call back the original APIs. One can implement policy checking and enforcement in Java and compile them into a `jar` package. So the injected library can easily load the logics in the `jar` file.

Most of existing HIPS on mobile devices are implemented by "*hooking on the client side*". Let us elaborate on this approach and we will also illustrate its deficiency.

3.3.2 API Hooking on the Client Side

In Section 2, we explain the client-server model in the Binder communication. To realize HIPS via API hooking on the client side is to hook the shared libraries of the normal apps so as to perform policy enforcement. To do this, we need to first understand the workflow of the Binder transaction on the client side. To illustrate, consider that we use the following codes to send an SMS message:

```
SmsManager smsManager = SmsManager.getDefault();
smsManager.sendTextMessage(
    "phoneNumber", null, "message", null, null
);
```

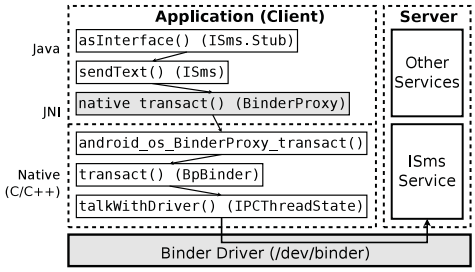


Figure 3: Call graph of ISms Binder transaction on the client side.

The first line is to query the ISms service (`ISms` handler) from the Service Manager. When calling the `sendTextMessage`, the system will call the `sendText` method of the `ISms` handler. Figure 3 depicts the partial function call graph of a Binder transaction for sending an SMS message. On the call path, there is a JNI method, `native transact`, which bridged with the `android_os_BinderProxy_transact` on the side of the native code. Finally, the transaction will end with the `talkWithDriver` function on the client side. Because the `native transact` method is the last method at the Java level, we can intercept it for hooking. Because it is a JNI method, the `insns` pointer in `gDvm` points to the address of the corresponding native function (`android_os_BinderProxy_transact`). Thus, we can store the original address, and replace the method pointer with our implementation inside the injected `so` file. After inspecting this transaction using our policy enforcement module, we can call back the original API, `android_os_BinderProxy_transact`. As shown in Figure 2, all apps are forked by a process called `zygote`. Therefore, instead of injecting an `so` file into any app, we can simply inject it to `zygote` at booting time. By using this approach, we can perform policy enforcement on any transactions which use the intercepted method.

Although the implementation of API hooking on the client side is straightforward and has been implemented in commercial systems [10, 9], this approach has a severe security limitation. This approach cannot ensure the security of the HIPS because the hooking operations are done in the *same sandbox* of the app. This implies that every operation can be modified and bypassed by the app itself. Therefore, it is dangerous if the malware writer discovers this HIPS implementation. Let us now demonstrate how to bypass this popular HIPS implementation.

3.4 Vulnerability of Existing HIPS Products

There are a number of existing HIPS products (e.g., [10, 9]) available in the market. In here, we show how to bypass these HIPS and elaborate on the vulnerability of the client-based hooking architecture (or *Client HIPS*). Client HIPS is one of the most popular Android anti-virus and protection architecture. For example, some HIPS products (e.g., LBE [10]) have more than ten million users. Moreover, Client HIPS has been pre-installed in a number of Android phones (e.g., Xiaomi smartphone). In our experiment, we take LBE (version: 5.4.7893) as an example to illustrate the vulnerability.

By examining the memory structure of the installed apps and the services, we can determine whether app processes have been injected with an `so` file and a `jar` file. Figure 4 illustrates the memory structure of a normal app. There

```
# cat /proc/1459/maps                          <-- Memory structure of pid: 1459
...
6b811000-6b813000 r--s 00015000 b3:1d 927            <-- Mapped memory region
/data/data/com.lbe.security/app_hips/client.jar      <-- File path
6b831000-6b860000 r--p 00000000 b3:1d 1185
/data/dalvik-cache/data@data@com.lbe.security@app_hips@client.jar@classes.dex
6cc47000-6cc52000 r-xp 00000000 b3:1d 262788
/data/data/com.lbe.security/app_hips/libclient.so
...
```

Figure 4: Memory structure of an app.

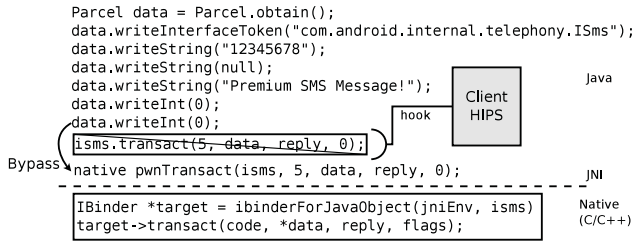


Figure 5: Workflow to bypass client HIPS.

are injected `so` file and `jar` file which do not belong to the app. This implies that the Client HIPS is using the client side API hooking to perform policy enforcement. By checking the `insns` value in the `gDvm`, the pointing address of `native transact` is `0x6CC4FED8`. In addition, from the mapped memory regions of apps, we find that the address of `native transact` resides in `0x6CC47000-0x6CC52000` region which mapped to `/data/data/com.client_hips/app_hips/libclient.so` file. Therefore, we can confirm that the system intercepts the `native transact` method in `android.os.Binder` class and points to the injected library.

Based on the discussion in Section 3.3, we know that the Client HIPS is in the same sandbox with the app. There are several ways to bypass the HIPS system. Firstly, we can modify the `insns` back to the original one to make sure that the transaction will not go through the policy enforcement. Secondly, because the `native transact` method is intercepted, malware writers can create their own transact implementation to bypass all the interceptions. Figure 5 demonstrates the workflow to bypass the Client HIPS. We implement our `native pwnTransact` method to send transactions to the target service directly. Therefore, Client HIPS cannot capture these transactions through intercepting `transact` method. In summary, malware writers can use the technique described above to determine the existence of Client HIPS (by examining the current mapped memory region `/proc/[pid]/maps` file). This shows that hooking on the client side *cannot* effectively prevent intrusions.

4. PATRONUS

In this section, we present our system, *Patronus*, a secure architecture which can prevent suspicious transactions and dynamically detect malware based on runtime transaction information. One advantage of Patronus is that we do not need to modify the Android system or repackage apps, and this facilitates easy deployment of our system. We present the system design of Patronus, and the methodologies used for intrusion prevention and dynamic detection.

4.1 API Hooking on the Server Side

As stated in Section 3.4, API hooking on the client side has a number of deficiencies. We also illustrated how to bypass detection if HIPS is implemented via API hooking on the client side. To overcome these security issues, Patronus

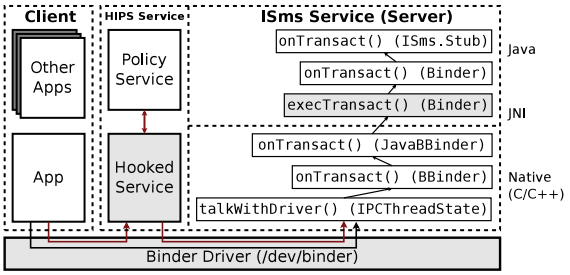


Figure 6: Call graph of ISms Binder transaction on the server side.

complements the API hooking by realizing it *both* on the client side and the server side. Let us first present our design and implementation of API hooking on the server side.

There are two approaches to conduct API hooking on the server side. The first approach is via the *Java API hooking*, and the second approach is to *hook the* `Service Manager`. Figure 6 shows that the transaction goes through the Binder driver from the client side to the server side and will be executed by the `ISms` service. There is a JNI between the native method and the Java method. Therefore, the first approach of Java API hooking is to replace the address of this Java method, `execTransact`, in the DVM global object to our policy enforcement functions. Afterward, the hooking function will call back the original function if our policy accepts this transaction. Note that using this approach, some of the services could not be protected by the policy enforcement. Because we only hook the Java method of JNI, the interception will be invalid when the service uses native codes (e.g., camera service in Android is only implemented by the native codes). Therefore, this hooking method can only intercept *subset* of API calls.

The second approach of API hooking provides a more comprehensive solution, which is to intercept the `Service Manager`. In Section 2, we explained that the `Service Manager` manages all registered services. When an app wants to utilize a service, it should first communicate with the `Service Manager` to query for the handler of the service. Therefore, we can intercept the `Service Manager` and reply back to the app with a handler which points to the service of HIPS for policy enforcement. Figure 6 shows the basic flow of hooking on the `Service Manager` (as illustrated in red lines). When an app queries the `ISms` Service from the intercepted `Service Manager`, it will get the handler of HIPS' hooked Service, and when an app uses the handler to send an SMS message, the hooked Service will first check with the Policy Service, then an accepted transaction will return to the original `ISms` service to complete the SMS message sending operation.

Note that many malware have the network capability. However, hooking the Binder cannot monitor the network APIs because network function in Android relies on socket IPC, and the network APIs of Android (e.g., Apache HTTP Client and `HTTPURLConnection`) use socket to setup HTTP connections with servers. Hence, any app can communicate with a remote server without using the Binder if they have declared the `android.permission.INTERNET` permission in the `AndroidManifest.xml`. One approach to monitor network behaviors is to intercept socket related APIs. For example, `java.net.Socket` is a class used by other internet APIs. In Patronus, we use the technique of hooking on the client side to intercept the `connect` method in the

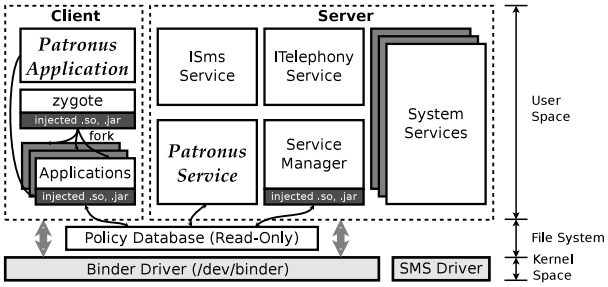


Figure 7: Architecture of Patronus.

`java.net.Socket` class to monitor the behaviors. Note that apps can also use network via the native `socket` function call. Although we can use `ptrace` to inspect the `socket` and `connect` functions to cover this case, this will bring much overhead. Because we want to create a HIPS on mobile phones which are resource constrained, we use the technique of hooking on the client side to intercept the `connect` method in the `java.net.Socket` class to monitor the behaviors.

4.2 System Design of Patronus

In Section 3, we presented and demonstrated the weaknesses of existing host-based intrusion prevention system on Android. In order to overcome these security problems and provide better functionalities, we set the following two design goals for Patronus: (1) It should dynamically prevent suspicious malware intrusions and can detect malware using runtime information (i.e., transaction information); (2) the architecture needs to be *extensible* and provides a flexible methodology to detect malware without modifying Android system or the need to repackage apps. Figure 7 depicts the architecture of Patronus. The system consists of four components: (a) *Patronus application*, (b) *Patronus service*, (c) *injected files* and (d) *policy database*.

• **Patronus Application:** A process which is responsible for displaying the user interface, initiates the Patronus Service and injects `so` or `jar` files. Specifically, Patronus Application uses the `ptrace` to inject a shellcode so as to load the `so` or `jar` files, which will be injected in the `zygote` on the client side and the `Service Manager` on the server side.

• **Patronus Service:** A process which is launched as the start of the system and it is a *sticky service*, which means that the service will not be killed by the system even in the condition with limited memory resources. Hence, this process is always available and we use it to detect any malware which registers at installation. Patronus Service will intercept all transactions on the server side by receiving the redirected transactions from the hooked Service Manager so as to perform policy enforcement and dynamic detection. Furthermore, Patronus Service is the owner of the policy database process and it has permission to read and write the database. For malware, they cannot modify the policy database due to the file system protection of Linux system. Patronus Service will also record all runtime information (i.e., transaction parcels) in a temporary file so that the system can perform the dynamic detection.

• **Injected files**: There are two types of injected files. Injected files on the client side and injected files on the server side. Because malware can access injected files on the client side, these injected files only have the read permission on the policy database so as to avoid modification by the malware. The injected files on the client side use the method explained in Section 3.3.2 to intercept the `native transact` method. These injected files will first check the transaction policy in the database which resides on the client side. By checking the transaction policy, Patronus can inspect and prevent intrusions before they arrive at the server side. Using the approach of API hooking on the server side which we explained in Section 4.1, the injected files on the server side will intercept `execTransaction` method of the `Service Manager`. So whenever there is any query from the client side, it will reply the Patronus service handler back to all queries. Therefore the transactions on the server side will first arrive at the Patronus Service, and they will be checked based on the policy database before dispatching these transactions to the corresponding servers. In our current implementation, we implement our policy checking and enforcement in API hooking, and use the Android SDK (e.g., `Popup Window` class to show a popup window on the apps).

• **Policy database:** This database contains the policy enforcement rules for each type of transaction of all apps. In our current implementation, we store these rules in an `XML` file. Using the Patronus Service, users can block transactions from some untrusted apps. Specifically, Patronus App communicates with Patronus Service to write the user-defined policy into the policy database. Users can enable some trusted apps (e.g., Google apps) or deny some transactions (e.g., `sendText`, `requestLocationUpdates`) if they do not intend to request these sensitive services. Due to the security consideration, this policy database belongs to the Patronus Service user and can only be modified by the trusted Patronus Service. Hence, with the protection of Linux file system, malicious app cannot destroy the policy database.

With these four components, Patronus can inspect the transactions either on the client side or on the server side. By analyzing each transaction, Patronus can prevent intrusions and detect malware.

4.3 Intrusion Prevention

Patronus can inspect transactions dynamically and will notify users if it encounters some intrusive transactions. In the Android system, there are hundreds of transaction types. The intrusive transactions are the transactions which can invoke some dangerous behaviors, for example, some malware subscribe premium service by sending an SMS message, and some spyware steal the location information using GPS service. Table 1 lists some intrusive transactions with the transaction descriptors, codes and codes name. For instance, the `requestLocationUpdates` transaction in the `android.location.ILocationManager` can gain the location information based on GPS. When this intrusive transaction is not initiated by users or without users' permissions, Patronus will alert the users using a popup window so that the users can pay attention to this transaction. Users can also deny this transaction at runtime. Patronus will monitor the transaction stream based on the intrusive transaction list. It is important to note that users or security analysts can add and/or delete rules in the Patronus policy database to ignore some intrusive transaction from some apps. For example, a third-party SMS mobile app can send, read and write SMS messages. If users are certain about the authenticity and security of this mobile app, users can delete rules for this SMS related intrusive transactions in order to eliminate the popup alerts. Also, the list is defined by the system (i.e., anti-virus software) which contains the sensitive transactions. Legiti-

Table 1: Intrusive Transaction List

TD	TC	TC Name
com.android.internal .telephony.ISms	4	sendData
	5	sendText
	...	...
com.android.internal .telephony.ITelephony	1	dail
	2	call
	28	getCellLocation
	...	...
android.location .ILocationManager	1	requestLocationUpdates
	5	getLastLocation
...	...	...

TD: Transaction Descriptor, TC: Transaction Code

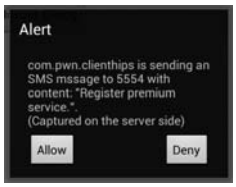


Figure 8: Alert of intrusive transaction (`sendText`).

mate transactions (e.g., opening window, changing Activity status) will not have any malicious behaviors.

Patronus uses two procedures to ensure the system security. The first is on the client side. Before the intrusive transaction goes through the Java layer and the native layer, Patronus will first inspect the transaction and check if there is any permission affiliated with this transaction. If there is no pre-defined policy, Patronus will pop up a window as an alert. The alert contains the transaction type, calling app of the transaction and transaction content (e.g., destination number and content of the SMS message) if needed. If the user allows this transaction, this transaction will proceed to the server side. The Patronus Service will not perform the second inspection. If the server finds that the transaction was not checked by the injected files on the client side, this implies that the app attempts to bypass the system or uses the native code. Then Patronus Service will perform policy enforcement to ensure the security of the transaction.

In Section 3.4, we illustrated several methods to bypass some existing HIPS. Patronus can easily detect and defend against these type of attacks. Firstly, if the malware attempts to bypass the transaction inspection on the Java layer, the system will perform the second inspection on the server side. Since the second verification is on the server side, the malware cannot bypass this step due to the process isolation feature in Android. Secondly, the Patronus policy database is also an important component of the system. The policy database can be read by any app but only Patronus has the write permission to the database. This mechanism guarantees that the policy database will not be compromised or contaminated by malware. Hence, Patronus provides a secured and complete intrusion prevention functionality. Consider the same mobile app (the one we discussed in Section 3.4) which bypassed the detection of a commercial HIPS product. When we execute this app on a mobile phone with Patronus, our system can intercept the intrusive transaction and the alert will popup, as shown in Figure 8.

4.4 Dynamic Detection

Besides intrusion prevention, Patronus can detect malware at runtime and block malicious behaviors before the malware infects the system. By using the runtime information to detect malware, Patronus can determine malware accurately and defend against obfuscation effectively. The dynamic detection consists of two steps: (1) *malware transaction forensics* and (2) *two-phase dynamic detection*.

4.4.1 Malware Transaction Forensics

Malware transaction forensics is a procedure to trigger the malicious behaviors and to collect the runtime transaction information. This can help the system to conduct dynamic detection. It includes two steps: *malware triggering* and *malware transaction tagging*.

Once the app is determined to be a potential malware (using existing static and dynamic analysis), the analysts can execute the malware on a test phone with the Patronus system. The analysts can manually trigger the malicious behaviors, and the system will record the transaction information (i.e., transaction descriptor and transaction code). If the transaction is an intrusive transaction, Patronus also records the content of the transaction parcel.

After collecting the transactions, the analysts can tag the malicious transactions in the set of suspicious intrusive transactions. The malicious transactions are used to invoke malicious behaviors by malware, such as sending premium SMS message, tracking location information and stealing contact information. Because the contents of transactions are evidence of malicious behaviors, these will be used to determine the type of malware we are analyzing. We want to emphasize that malware transaction tagging will not add more workload for analysts since we only tag malicious transactions and the system already filtered many other legitimate transactions. In our experiments, the number of intrusive transactions for a malware is less than five and most of them are tagged as malicious transactions. Let us now define *transaction footprint*, which we use for malware detection.

Definition 1. *A transaction footprint is a set of transaction information tuples $S = \{T_1, T_2, \ldots, T_n\}$ over runtime transactions where:*

- *The index represents the unique id of the combination of transaction descriptor and code.*
- *The transaction information tuple $T_i = (N_i, F_i, C_i)$, where N_i is the number of invocations of transaction i, F_i is the boolean flag tagging the malicious transaction i, C_i represents the content of the transaction parcel.*

We can use the transaction footprint for dynamic detection. Firstly, it describes the runtime behaviors of a group of similar malware. Secondly, it also contains evidence of malicious transactions. Due to these two factors, we propose a two-phase dynamic detection method to effectively detect the malware on an Android mobile phone at runtime.

4.4.2 Two-phase Dynamic Detection

The dynamic detection contains two phases: *correlation detection* and *transaction footprint comparison*. Because only the intrusive transactions can initiate malicious behaviors. For efficiency, the detection will be triggered when the system encounters an intrusive transaction. The system will use the transaction footprint collected at runtime and the malware footprint to detect the malware.

Phase one. Define V as the transaction vector over a transaction footprint S where $V = [v_1, v_2, \ldots, v_n]$, $(v_i = N_i)$. In phase one, we collect $V_{runtime}$ which is a transaction vector over the transaction footprint for the runtime app. Let

$V_{malware}$ be the transaction vector over the transaction footprint in malware transaction forensics. In Patronus, we use the *Pearson correlation* as the similarity score r so as to determine the correlation between the app and our malware samples. Formally, the similarity score r is:

$$r = \frac{\sum_{i=1}^{n}(X_i - \bar{X})(Y_i - \bar{Y})}{\sqrt{\sum_{i=1}^{n}(X_i - \bar{X})^2}\sqrt{\sum_{i=1}^{n}(Y_i - \bar{Y})^2}}. \quad (1)$$
$$where\ X = V_{runtime},\ Y = V_{malware}$$

Because transactions of an app represent the behaviors when interacting with other apps, the higher r implies that the known malware in our database and the mobile app under testing are very similar. The system will calculate r with the transaction footprint in the malware database.

Phase two. If there is any r higher than a pre-defined threshold (the threshold we use in our experiment was set as 0.9), we treat the detected app as a suspicious app. For the suspicious app, we conduct the phase-two detection to ensure the accuracy of our detection. In phase two, Patronus will use the content of malicious transactions $[C_1, C_2, \ldots, C_i]$, $(F_i = true)$ in the transaction footprint S to perform further verification. The system will compare the decisive fields in the content of each intrusive transaction from the suspicious app with the selected footprints. The decisive fields are values indicating the malicious behaviors. In general, the content of a transaction contains several fields, the semantics of the fields for each transaction can be different. For example, `sendText` transaction has four fields which indicate the destination address, source address, text and intent flag. The decisive fields for a malicious transaction are values which determine the malicious properties. For example, the destination field (number of premium SMS service) is the decisive field of a malicious sendText transaction.

By using the above two-phase detection, Patronus can detect malware based on the runtime information effectively and prevent the malware before it infects the system or steal users' privacy. Since the transaction information we obtain have more semantic information than the low-level system calls using the `ptrace`-based dynamic analysis system, this makes Patronus achieving a much higher accuracy in detecting intrusive apps. Furthermore, because one transaction procedure usually only contains tens of system calls, the performance overhead of transaction-based detection will be less than system-call-based detection. Moreover, traditional static analysis systems have to scan the apps offline, while Patronus can perform detection online.

5. LARGE SCALE EVALUATION

In this section, we analyze the capabilities of intrusion prevention and dynamic detection on large number of mobile apps. We also evaluate the performance overhead and power consumption introduced by Patronus.

5.1 Capability Evaluation

We downloaded the top 500 legitimate mobile apps from Google Play and use them as the base for our evaluation database. In addition, we also collected three malware families from [2] and [11] including 213 `BaseBridge` samples, 9 `FakeAV` samples and 15 `MobileTx` samples. We also define an intrusive transaction list (`https://www.cse.cuhk.edu.hk/~mssun/pub/intrusive_transaction_list.pdf`) including 49 transactions within eleven services.

Table 2: Top 10 Intrusive Transactions

Transaction Name	Total #
CALL_TRANSACTION	3,508
REGISTER_RECEIVER_TRANSACTION	2,960
START_ACTIVITY_TRANSACTION	1,734
TRANSACTION_getDeviceId	1,732
GET_CONTENT_PROVIDER_TRANSACTION	1,400
QUERY_TRANSACTION	1,303
TRANSACTION_getSubscriberId	333
TRANSACTION_requestLocationUpdates	228
INSERT_TRANSACTION	139
TRANSACTION_getCallState	90

Table 3: Transaction Statistics

Service	Top 500	BaseBridge	FakeAV	MobileTx
PackageManager	0	0	0	0
Telephony	90	0	0	0
TelephonyRegistry	52	1	0	17
ContentProvider	4,996	131	0	0
LocationManager	228	7	0	25
ActivityManager	6,156	295	0	98
AudioService	9,781	1,453	37	40
Sms	0	0	0	49
PhoneSubInfo	2,208	622	0	0
NotificationManager	639	336	161	0
PhoneStateListener	0	0	0	0
Total transaction	724,185	66,229	548	3,920
Intrusive transaction	24,150	2,845	198	229
Percentage	3.33%	4.30%	3.61%	5.84%

To evaluate Patronus, we automatically install the top 500 legitimate apps and malware samples into Android device (Nexus 5) with Patronus installed. Then we use `monkey` tool [12] to generate 500 pseudo-random user events such as clicks, touches, gestures or system-level events, into the device to trigger various behaviors. Patronus will record the transactions which are in the intrusive transaction list. Table 2 shows the top ten intrusive transactions and their total numbers recorded in our evaluation. Most of these transactions are dangerous and related with privacy operation. For example, `CALL_TRANSACTION`, `GET_CONTENT_PROVIDER_TRANSACTION`, `QUERY_TRANSACTION` and `INSERT_TRANSACTION` are operations (e.g., querying and updating) on databases which may contain users' personal information. While `TRANSACTION_requestLocationUpdates` can get current location information. Table 3 depicted detailed statistics of these intrusive transactions on all our downloaded apps. Note that the percentage of intrusive transaction is only around 4%. Therefore, the alert notifications of intrusive transaction will not annoy users in practice.

To evaluate the capability of dynamic detection, we first conduct transaction forensics on the malware families in our evaluation database. We automatically run these malware samples along with 500 legitimate apps to analyze the effectiveness of our detection algorithm. During the testing, we also generated 500 pseudo-random user events such as clicks, touches, or gestures for each app so to trigger different routines. We calculate the distribution of correlation scores and they are illustrated in Figure 9. From this distribution, we choose 0.9 as our *threshold* in the two-phase detection which effectively separate malware and legitimate app. Table 4 shows the detection results of malware samples within the legitimate apps. We compute the true positive (TP), true negative (TN), false positive (FP), false negative (FN), precision and F-score to illustrate our results. From these results, we observe that only few legitimate apps is reported as malware in the first phase of detection. We like to

Table 4: Detection Results

Malware	# of Samples	TP	TN	FP	FN	Precision	F-score
BaseBridge	213	186	495	5	27	0.87	0.92
FakeAV	9	9	500	0	0	1	1
MobileTx	15	11	494	6	4	0.65	0.69

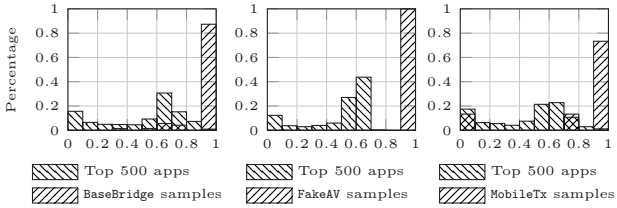


Figure 9: Distribution of correlation scores for BaseBridge, FakeAV and MobileTx samples.

note that for MobileTx families, some of the samples crashed in testing due to the bugs of the malware. Therefore, we obtained four false negative in our evaluation. In summary, Patronus can effectively prevent intrusions and detect existing malware.

5.2 Performance Evaluation

To measure the overhead introduced by Patronus, we use Quadrant Standard Edition v2.1.1 [6] to evaluate the general-purpose benchmark for CPU, memory, I/O, 2D and 3D graphics. The experiments are conducted on a LG Nexus 5 (Qualcomm Snapdragon 800 2.26GHz CPU, Adreno 330 450MHz GPU, 2300mAH battery) running Android 4.4.2 (KitKat) on a Linux 3.4.0 kernel. Table 5 shows the benchmark results. The "Baseline" means that the benchmarking was done on the device without Patronus. Compared with the baseline, Patronus only has a small impact on all but I/O intensive apps. However, we want to note that I/O operations are not the main operations for many mobile apps.

Because Patronus inspects transactions, we implement a transaction intensive app to evaluate its performance. The test app uses IPhoneSubInfo transaction to request device ID (transaction #: 1). The app will repeatedly send the transaction for one thousand times. We calculate the execution time of these transactions. The time on the device without Patronus is 890 milliseconds, while the execution time for a device with the Patronus installed increase to 988 milliseconds. This shows that the presence of Patronus will only bring about 11.1% performance overhead.

We also measure the battery overhead introduced by Patronus. We conduct two sets of experiments. In the first experiment, we check the battery usage of a fully-charged Nexus 5 in the standby mode for 24 hours. The device without Patronus has 94% battery left, and the device with Patronus installed shows 93%. This shows that Patronus only introduces a slight overhead. In the second experiment, we keep playing games on a fully-charged Nexus 5 for one hour. The results show that the device with Patronus uses 3% more battery than the original setting. In summary, Patronus not only provides a comprehensive intrusion protection, but also has a negligible impact on power consumption.

6. RELATED WORK

There is a growing interest on how to provide security and privacy on smartphones. For Apple's iOS, Wang *et al.* [51] discovers a severe security architecture problem. While most security issues are also observed on the Android platform.

Table 5: Benchmark Results

Test	Baseline	Patronus	Overhead
Total	8,914	8,285	7.1%
CPU	20,383	20,205	0.9%
Memory	14,354	13,211	8.0%
I/O	7,274	6,482	10.9%
2D	333	330	0.1%
3D	2,230	2,195	1.6%

Permission re-delegation problem is discussed in [20, 31] and some researchers propose possible solutions [28, 16, 22, 36]. Felt *et al.* [27, 15] systematically analyzes the permission abuse problem. AdRisk [33] reveals the potential privacy risks of advertisement libraries. Luo *et al.* [37] identifies the problem of Android WebView component. Zhou *et al.* [63] conducts a comprehensive study on the characterization and evolution of Android malware. Wu *et al.* [52] and Zheng *et al.* [60] analyze the security issues on the customized firmwares. Our study mainly focuses on the dynamic detection of Android malware and Android protection.

There are a number of works which propose to use HIPS on the PC platform [46, 43, 42]. However, the architectures of PC and mobile devices are very different. Therefore, few systems are proposed and targeted specifically for mobile operating systems. FireDroid [49] is a ptrace-based system call interception system. Because system calls are low level function calls interacting with kernel, one API call can generate a number of system calls. Therefore, ptrace-based system cannot accurately intercept all intrusive operations. TaintDroid [24] and PiOS [23] can track runtime privacy leaks on Android and iOS. DroidScope [55] provides a Dalvik semantic view for dynamic analysis. CrowDroid [18] is a detection system based on runtime system call. VetDroid [57] utilizes permission usage to study undesirable dynamic behaviors. AppIntent [56] aims to detect privacy leakage using symbolic execution to reduce the code search space. Android DDI [40] introduces the method to instrument the DVM. These systems are for malware dynamical analysis which are mainly used by malware analysts. However, our system aims to prevent intrusions on the user end. Aurasium [54] and RetroSkeleton [21] are two host-based systems which use repackaging technique. FlaskDroid [17] provides mandatory access control on SE Android. Compared with our system, they need to either repackage the apps or modify the system structure.

For static analysis, some systems aim to detect suspicious apps, zero-day malware and evaluate anti-virus software. DroidMOSS [62] and DroidRanger [65] can identify the potential repackaged apps or zero day malware in the third-party markets. DroidAnalytics [61] proposes a three-level signature to analyze malware to defend against obfuscation. ADAM and DroidChameleon [59, 48] discuss approaches to bypass anti-virus engines. MAST [19] analyzes the market-scale malware. Zhou *et al.* [64] proposes a system to find content leaks and pollution vulnerabilities. These systems can help the analysts to investigate the malicious behaviors at the code level.

7. CONCLUSION & FUTURE WORK

In this paper, we systematically analyze three popular frameworks of HIPS, and discuss their implementations, strengths and weaknesses. We also demonstrate how to bypass a popular HIPS software in the market. Moreover, we propose

a secure HIPS architecture and implement Patronus which can prevent intrusion and can dynamically detect malware. With the two-phase dynamic detection, Patronus utilizes runtime information to detect existing malware. We conduct extensive experiments to demonstrate the intrusion prevention and dynamic detection capabilities of Patronus, and demonstrate the Patronus only incurs small overhead in executing mobile apps. For future work, we plan to utilize cloud service along with uploaded runtime information collected by Patronus to improve the detection capability.

8. REFERENCES

[1] 360 one click root. http://shuaji.360.cn/root/index.html.

[2] Android Malware Genome Project. http://malgenomeproject.org.

[3] APKfuscator. https://github.com/strazzere/APKfuscator.

[4] Apktool. https://code.google.com/p/android-apktool/.

[5] App Shield. http://www.wandoujia.com/apps/com.gmail.exathink.appshield.

[6] Aurora Softworks quadrant standard edition. https://play.google.com/store/apps/details?id=com.aurorasoftworks.quadrant.ui.standard.

[7] cyanogenmod. http://www.cyanogenmod.org.

[8] DroidBox. https://code.google.com/p/droidbox/.

[9] Jinshan mobile duba. http://m.duba.net/.

[10] Lbe secrity guard. http://www.lbesec.com/.

[11] mobile malware mini dump. http://contagiominidump.blogspot.com/.

[12] monkey. http://developer.android.com/tools/help/monkey.html.

[13] Qihoo 360 mobile guard. http://shouji.360.cn/.

[14] smali/baksmali. https://code.google.com/p/smali/.

[15] K. W. Y. Au, Y. F. Zhou, Z. Huang, and D. Lie. Pscout: analyzing the android permission specification. In *CCS*, 2012.

[16] S. Bugiel, L. Davi, A. Dmitrienko, T. Fischer, A.-R. Sadeghi, and B. Shastry. Towards taming privilege-escalation attacks on android. In *NDSS*, 2012.

[17] S. Bugiel, S. Heuser, and A.-R. Sadeghi. Flexible and fine-grained mandatory access control on android for diverse security and privacy policies. In *USENIX Security*, 2013.

[18] I. Burguera, U. Zurutuza, and S. Nadjm-Tehrani. CrowDroid: behavior-based malware detection system for android. In *Proc. of the 1st ACM workshop on Security and privacy in smartphones and mobile devices*, 2011.

[19] S. Chakradeo, B. Reaves, P. Traynor, and W. Enck. MAST: triage for market-scale mobile malware analysis. In *ACM WiSec*, 2013.

[20] E. Chin, A. P. Felt, K. Greenwood, and D. Wagner. Analyzing inter-application communication in android. In *MobiSys*, 2011.

[21] B. Davis and H. Chen. RetroSkeleton: retrofitting android apps. In *MobiSys*, 2013.

[22] M. Dietz, S. Shekhar, Y. Pisetsky, A. Shu, and D. S. Wallach. QUIRE: Lightweight provenance for smart phone operating systems. In *USENIX Security*, 2011.

[23] M. Egele, C. Kruegel, E. Kirda, and G. Vigna. PiOS: Detecting privacy leaks in ios applications. In *NDSS*, 2011.

[24] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. TaintDroid: an information-flow tracking system for realtime privacy monitoring on smartphones. In *OSDI*, 2010.

[25] W. Enck, D. Octeau, P. McDaniel, and S. Chaudhuri. A study of android application security. In *USENIX Security*, 2011.

[26] F-Secure. Threat report h2 2013.

[27] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner. Android permissions demystified. In *CCS*, 2011.

[28] A. P. Felt, H. J. Wang, A. Moshchuk, S. Hanna, and E. Chin. Permission re-delegation: attacks and defenses. In *USENIX Security*, 2011.

[29] J. Forristal. Android: One root to own them all. In *Blackhat USA 2013*, 2013.

[30] Google. Platform versions. http://developer.android.com/about/dashboards/index.html.

[31] M. Grace, Y. Zhou, Z. Wang, and X. Jiang. Systematic detection of capability leaks in stock android smartphones. In *NDSS*, 2012.

[32] M. Grace, Y. Zhou, Q. Zhang, S. Zou, and X. Jiang. RiskRanker: scalable and accurate zero-day android malware detection. In *MobiSys*, 2012.

[33] M. C. Grace, W. Zhou, X. Jiang, and A.-R. Sadeghi. Unsafe exposure analysis of mobile in-app advertisements. In *WiSec*, 2012.

[34] N. Harbour. Win at reversing: Api tracing and sandboxing through inline hooking. In *DEFCON*, 2009.

[35] IDC. Apple cedes market share in smartphone operating system market as android surges and windows phone gains. http://www.idc.com/getdoc.jsp?containerId=prUS24257413.

[36] L. Lu, Z. Li, Z. Wu, W. Lee, and G. Jiang. CHEX: statically vetting android apps for component hijacking vulnerabilities. In *CCS*, 2012.

[37] T. Luo, H. Hao, W. Du, Y. Wang, and H. Yin. Attacks on webview in the android system. In *ACSAC*, 2011.

[38] W. Luo, S. Xu, and X. Jiang. Real-time detection and prevention of android sms permission abuses. In *Proc. of the first international workshop on Security in embedded systems and smartphones*, 2013.

[39] McAfee Labs. McAfee threats report: Second quarter 2013. Technical report, McAfee Labs, 2013.

[40] C. Mulliner. Android DDI: Introduction to dynamic dalvik instrumentation. In *The 11th Annual HITB Security Conference in ASIA*, 2013.

[41] C. Mulliner, J. Oberheide, W. Robertson, and E. Kirda. Patchdroid: scalable third-party security patches for android devices. In *ACSAC*, 2013.

[42] N. Nethercote and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. *ACM Sigplan Notices*.

[43] J. Newsome and D. Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. *Internet Society*, 2005.

[44] P. Olson. Your smartphone is hackers' next big target. http://edition.cnn.com/2013/08/26/opinion/olson-mobile-hackers, August 2013.

[45] A. Police. App Ops: Android 4.3's hidden app permission manager, control permissions for individual apps.

[46] N. Provos. Improving host security with system call policies. In *USENIX Security*, 2003.

[47] V. Rastogi, Y. Chen, and W. Enck. AppsPlayground: automatic security analysis of smartphone applications. In *CODASPY*, 2013.

[48] V. Rastogi, Y. Chen, and X. Jiang. DroidChameleon: evaluating android anti-malware against transformation attacks. In *ASIACCS*, 2013.

[49] G. Russello, A. B. Jimenez, H. Naderi, and W. van der Mark. Firedroid: hardening security in almost-stock android. In *ACSAC*, 2013.

[50] T. Strazzere. Dex education: Practicing safe dex. In *Blackhat USA 2012*, 2012.

[51] T. Wang, K. Lu, L. Lu, S. Chung, and W. Lee. Jekyll on iOS: when benign apps become evil. In *USENIX Security*, 2013.

[52] L. Wu, M. Grace, Y. Zhou, C. Wu, and X. Jiang. The impact of vendor customizations on android security. In *CCS*, 2013.

[53] xda-developers. PDroid. http://forum.xda-developers.com/showthread.php?t=1357056.

[54] R. Xu, H. Saïdi, and R. Anderson. Aurasium: practical policy enforcement for android applications. In *USENIX Security*, 2012.

[55] L. K. Yan and H. Yin. DroidScope: seamlessly reconstructing the os and dalvik semantic views for dynamic android malware analysis. In *USENIX Security*, 2012.

[56] Z. Yang, M. Yang, Y. Zhang, G. Gu, P. Ning, and X. S. Wang. AppIntent: Analyzing sensitive data transmission in android for privacy leakage detection. In *CCS*, 2013.

[57] Y. Zhang, M. Yang, B. Xu, Z. Yang, G. Gu, P. Ning, X. Wang, and B. Zang. Vetting undesirable behaviors in android apps with permission use analysis. In *CCS*, 2013.

[58] C. Zheng, S. Zhu, S. Dai, G. Gu, X. Gong, X. Han, and W. Zou. Smartdroid: an automatic system for revealing ui-based trigger conditions in android applications. In *Proc. of the second ACM workshop on Security and privacy in smartphones and mobile devices*, 2012.

[59] M. Zheng, P. P. C. Lee, and J. C. S. Lui. ADAM: an automatic and extensible platform to stress test android anti-virus systems. In *DIMVA*, 2013.

[60] M. Zheng, M. Sun, and J. Lui. DroidRay: a security evaluation system for customized android firmwares. In *ASIACCS*, 2014.

[61] M. Zheng, M. Sun, and J. C. S. Lui. DroidAnalytics: A signature based analytic system to collect, extract, analyze and associate android malware. In *TrustCom*, 2013.

[62] W. Zhou, Y. Zhou, X. Jiang, and P. Ning. Detecting repackaged smartphone applications in third-party android marketplaces. In *Proc. of the second ACM conference on Data and Application Security and Privacy*, 2012.

[63] Y. Zhou and X. Jiang. Dissecting android malware: Characterization and evolution. In *IEEE Symposium on Security and Privacy*, 2012.

[64] Y. Zhou and X. Jiang. Detecting passive content leaks and pollution in android applications. In *NDSS*, 2013.

[65] Y. Zhou, Z. Wang, W. Zhou, and X. Jiang. Hey, you, get off of my market: Detecting malicious apps in official and alternative android markets. In *NDSS*, 2012.

MoRePriv: Mobile OS Support for Application Personalization and Privacy

Drew Davidson
University of Wisconsin
davidson@cs.wisc.edu

Matt Fredrikson
University of Wisconsin
mfredrik@cs.wisc.edu

Benjamin Livshits
Microsoft Research
livshits@microsoft.com

ABSTRACT

Privacy and personalization of mobile experiences are inherently in conflict: better personalization demands knowing more about the user, potentially violating user privacy. A promising approach to mitigate this tension is to migrate personalization to the client, an approach dubbed *client-side personalization*. This paper advocates for *operating system support* for client-side personalization and describes MoRePriv, an operating system *service* implemented in the Windows Phone OS. We argue that personalization support should be as ubiquitous as location support, and should be provided by a unified system within the OS, instead of by individual apps.

We aim to provide a solution that will stoke innovation around mobile personalization. To enable easy application personalization, MoRePriv approximates users' interests using *personae* such as technophile or business executive. Using a number of case studies and crowd-sourced user studies, we illustrate how more complex personalization tasks can be achieved using MoRePriv.

For privacy protection, MoRePriv distills sensitive user information to a coarse-grained *profile*, which limits the potential damage from information leaks. We see MoRePriv as a way to increase end-user privacy by enabling client-side computing, thus minimizing the need to share user data with the server. As such, MoRePriv shepherds the ecosystem towards a better privacy stance by *nudging* developers away from today's privacy-violating practices. Furthermore, MoRePriv can be *combined* with privacy-enhancing technologies and is complimentary to recent advances in data leak detection.

Keywords

Security, privacy, personalization

1. INTRODUCTION

Mobile applications are becoming increasingly *personalized*, fluidly adapting themselves to the needs and preferences of their users. Today, personalization typically aggregates user data in the *cloud* and uses it for large-scale data mining. This approach has significant advantages for data aggregators, allowing them to present targeted services and ads. However, as the ongoing DoNotTrack debate illustrates, not all users are comfortable with sharing information with large companies such as Facebook or Apple to obtain personalized results. Recently, a series of papers has highlighted the severity of data leak problems in mobile apps [16, 32, 20]. However, existing work in this domain usually only *detects* when apps use personalized data (in which case the user's only remedy is to not install the app at all), or falsifies the data that apps access [25]. In both cases, the user forgoes the benefits of personalization.

To balance the goals of privacy and personalization, we implemented mobile OS support for rich personalization in the form of MoRePriv[1] Privacy-aware personalization is a rapidly expanding research area, often taking the form of client-side personalization. Encouragingly, a recent study has demonstrated users' generally positive attitudes towards client-side personalization [29]. Much of the state-of-the-art client-side personalization work focuses on solving a specific problem, such as privacy-conscious ad targeting [44, 22]. Similarly, we observe a trend towards platform-provided ad support in efforts such as Window 8's Ads in Apps[2] and Blackberry's integrated Advertising Service[3]. In contrast, the focus of MoRePriv is on creating a platform to unleash developer creativity across a variety of personalization scenarios.

Client-side personalization: MoRePriv encourages *leaving* user data on the mobile device, mediated by the operating system and under the control of the user. This approach has many benefits: the user retains control over data, and cloud providers do not have to worry about properly securing data, respecting the wide array of user privacy preferences, complying with local and international laws, running and powering expensive storage clouds, dealing with PR repercussions of data leaks and unauthorized tracking (such as the Apple location data scandal), etc.

OS support: When user preference information is collected from all apps and OS interactions, it is considerably more accurate compared to what any given app, even the mobile browser, can obtain individually. Furthermore, the user already implicitly trusts the OS, so performing personalization there does not expand the trusted computing base. In MoRePriv, personalization functionality draws from a single unified, trustworthy, and protected source provided by the OS and exposed to developers via a simple API.

Personae: To enable easy application personalization, MoRePriv approximates user's interests using *personae* such as technophile or business executive. Personae can be used for a basic level of personalization, which can be extended through the use of *custom classifiers*. We demonstrate how always-on user interest mining can effectively and accurately infer user interests in a mobile operating system. MoRePriv inference is based on parsing and classifying

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.
ACSAC '14, December 08 - 12 2014, New Orleans, LA, USA
Publication rights licensed to ACM.978-1-4503-3005-3/14/12 ...$15.00
http://dx.doi.org/10.1145/2664243.2664266 .

[1] for Mobile OS for Reclaiming Privacy
[2] `http://adsinapps.microsoft.com/en-us`
[3] `adservices.blackberry.com`

multiple streams of (sensitive) information about the user within the OS, such as their email, SMS, Facebook stream, and network communications. For privacy protection, this sensitive information is distilled to a coarse-grained profile (as opposed to fine-grained location data, whose leaks have caused much chagrin among users and privacy advocates), without being exposed to apps. The use of personae within the user profile limits the potential for user tracking: while persona information can be shared by apps to perform server-based personalization, for instance, it is not sufficient to link an individual user across multiple discrete interactions.

1.1 Contributions

In this paper, we

- propose OS-wide personalization to achieve personalization and privacy on mobile devices, illustrate key opportunities for mobile operating system designers, and suggest ways in which mobile applications can be personalized;
- describe our implementation of MoRePriv on the Windows Phone OS; and evaluate the use of personae, through use cases and end-user experiments;
- demonstrate automatic personalization of legacy applications through case studies and show how MoRePriv APIs can be used by new apps for personalization and skinning;
- show how to support custom classifiers, the integrity of which can be verified using the machinery of zero-knowledge proofs to provide assurances to 3rd parties when the user has incentive to fake personalization measurements;
- establish that using zero-knowledge proofs in our context is practical in terms of computational demands experimentally, finding that a classifier using 100 feature words requires only 11 seconds of prover time, less than a second of transfer time on a 3G connection, and negligible battery usage;
- present crowd-sourced user studies showing how MoRePriv personalization can be effective in a widely-used app such as Yelp.

1.2 Paper Organization

The rest of this paper is organized as follows: Section 2 provides the background for mobile privacy and personalization. Section 3 gives an overview of MoRePriv. Section 4 describes how we implemented MoRePriv by modifying the Windows Phone operating system and demonstrates the MoRePriv API. Section 5 discusses support for custom classifiers and describes how to provide integrity of computed classification results using zero-knowledge proofs. Section 6 illustrates the benefits of mobile personalization through case studies and describes the result of our user studies. Section 7 summarizes the related work and Section 8 concludes.

2. BACKGROUND

The prevalence of smart phones and other mobile computing devices has opened up new avenues for personalized applications. As users carry these devices with them wherever they go, they are subject to a level of user interaction never before seen on a personal computing platform. Furthermore, they are typically equipped with cameras, microphones, GPS, and several forms of wireless networking, providing a constant stream of data from both the physical world and the Internet. Common examples of personalization include Siri, Google Now, and personalized local search. All of these currently move a considerable amount of user data into the cloud for processing. We begin by discussing compelling representative examples of mobile personalization and then proceed to generalize the requirements for a successful personalization platform.

Personal Assistants: Google Now and Siri are familiar examples of mobile personalization. Both attempt to perform high-level tasks on behalf of the user, such as planning commutes, selecting movies, and making wine recommendations. Over time, these systems learn the user's preferences and personalize recommendations accordingly, although the mechanisms by which this happens is proprietary. If the user invokes the services frequently, they become privy to a sizable portion of the user's day-to-day activities, giving a valuable window into the user's life when targeting content and advertisements. However, this data is processed in the cloud and is potentially visible to third parties affiliates for analysis. As such, both the information given directly by the user such as "set a birthday reminder" and secondary information such as gender and age may be obtained in bulk.

Shopping: Shopkick (`shopkick.com`) integrates location awareness with user preference data. When a Shopkick user enters a brick-and-mortar store, the app consults the user's preferences to offer discounts and recommendations. Users can scan the barcodes of interesting items and are rewarded with discounts within the store. This personalization is interesting in that it allows traditional merchants to to target content, advertisements, and discounts towards users in much the same manner as online retailers do.

Summary: The apps above share the following set of personalization steps [28]: 1. Acquire personalization signal from user interactions with the app; 2. Address the *cold start* problem of not having data to base personalization on [38]; often, this is resolved using a secondary source. Many modern apps encourage the user to sign up with their Facebook credentials so they can "scrape" their Facebook data; 3. Refine personalization signal as a result of subsequent interactions.

3. OVERVIEW

The apps described in Section 2 are representative of the personalization needs of many apps: they benefit from an understanding of the user, but need not collect enough information about the user to personally identify them. In this Section, we describe how MoRePriv provides this information through the key notion of *personae*.

3.1 User Interest Profile: Personae

Personae are custom representations of various walks of life, offering easy-to-understand targets for personalization [40]. In addition to being easy to understand, the advantage of this technique is the degree of *pseudonymity* that it provides [30]. In other words, using a persona provides a way to declassify sensitive information; persona data is all that is released about the user to the application, as opposed to personally-identifiable information like their name or unique identifiers such as device IMEI.

In our prototype implementation of MoRePriv, we target *eight* personae. Each persona is represented by a Bayesian classifier C_p, trained on a manually curated list of keywords characteristic to profile p. For example, the business executive persona represents strong interest in business, finance, and national news. Thus, the corresponding classifier is trained on text from sites as the Financial Times. On the other hand, the technophile profile represents a strong interest in technology, so the corresponding classifier is populated with text from tech blogs.

Our prototype of MoRePriv supports the following personae: activist, bachelor, business executive, sports buff, retiree, homemaker, technophile, tween. Note, however, that we do not advocate for this specific list of personae. Our goal in choosing the set of personae is neither completeness nor exclusivity: arguably, it is very difficult to be comprehensive; similarly, there may be intersections between these personae. Instead we optimize for ease of understanding by

the user and the developer. We believe these profiles are a reasonable proof of concept but note that our system is modular with respect to the personae that are used, with updates amounting to simply re-training a Bayesian classifier on a new list of keywords.

In practice, no user is likely to have interests that match exactly one persona. As such, each persona is assigned a *persona weight* that indicates how closely that persona matches the user. MoRePriv keeps a vector of these weights, called the user's *profile*. Consider an example user who is very interested in technology news and somewhat interested in financial news. His or her profile would have a high persona weight for the technophile persona and a moderate persona weight the executive persona. We believe that many users understand the value of personae: we performed a brief anonymous survey of 179 crowd-sourced survey-takers, 39–45% of respondents are more willing to share *persona* data, whether in general or for the purposes of personalization, compared to the familiar baseline of location data.

3.2 Design Choices

The architectural of MoRePriv is divided into three parts: User interactions are observed by *personal preference miners*, that can be as diverse as location information miners that tell whether the user is home or on a trip, or transit miners that discover the activity of the user (walking, riding a train, in a car, etc.). Miners also pore over the user's email, SMS, and Facebook streams to establish the user's interests and preferences in news and entertainment. These interactions are distilled to a user profile by *persona classifiers* that come pre-installed (we discuss details of extending the system via custom classifiers in Section 5). Finally, the profile is used by a variety of *personalizers* for different forms of personalization. user's interests and preferences for news and entertainment.

A key strength of MoRePriv is that users remain in control of whether and to what extent to use it, in order to appeal to a wide spectrum of users. Previous studies have shown that users' attitudes towards mobile privacy vary greatly: conservative users may opt-out and turn off personalization or data collection *entirely* Section 4.4. Furthermore, conservative users will likely not to install *any* third-party classifiers. More adventurous users who want to experience extra functionality may install custom classifiers, but will be prompted for additional permissions at the time of installation (Section 5.2).

3.3 Data Protection and Privacy

So far, our discussion of MoRePriv has primarily focused on the personalization benefits of our approach. In the rest of this section, we focus on privacy-related advantages. In the past several years, numerous projects have examined data leakage potential on the mobile phone, including mobile information leakage [16, 20, 14], leading to privacy-enhancing technologies designed to guard against such data leaks, such as AppFence [25].

We want to draw a contrast between that line of research and MoRePriv, which is envisioned as a privacy-by-design approach. The goal of MoRePriv is to *incentivize* the different constituents of the mobile ecosystem into compliance. MoRePriv can and should be combined with privacy monitoring and privacy leak detection tools, but we see that as an orthogonal concern.

Use of personae limits information leaks: At the core of the MoRePriv design is the use of personae, which provides a degree of pseudonymity [30]. Sensitive data is distilled by MoRePriv to a restricted, deliberately coarse-grained persona, and is never given to user mode apps directly [37]. A natural direction to consider is implementing an information flow restriction approach to mitigate leaks of persona information. While many research efforts have argued for information flow and tainting approaches, in both static and runtime flavors at levels varying from hardware to the application runtime, practicable adoption has been slow. We argue that this is because it is very difficult to deploy these systems without causing numerous false positives or tolerating many false negatives. Consequently, we choose not to following this path in MoRePriv.

Data representation: Our implementation of MoRePriv maintains a vector of persona weights for the user. This vector is maintained in memory and it potentially serialized as part of the OS service that MoRePriv implements. To make data maintenance incremental, our implementation records two values for each persona p (out of eight possible personae):

1. *relevance*: sum of interest scores from classifier C_p, s_p
2. *support*: number of elements that have been scored by C_p, called n_p.

Intuitively, s_p indicates a raw score of how closely the persona matches the user, while n_p indicates the amount of evidence to support that score. The persona weight is given by the fraction s_p/n_p. Thus, only two data items per persona are stored. Note that these vectors are never directly shared with user mode apps.

While this is not encouraged, in our model, apps are allowed to leak the most relevant persona of the user if they so desire. However, we consider the consequences of that to be relatively benign, especially when compared to the current practice of user monitoring built into many of today's privacy-leaking apps [24]. MoRePriv is designed to use a limited number of personae such that each persona will be the most relevant for many users. Thus, leaking the top persona of a user has a negligible contribution towards identifying a user and does not identify any specific interests of the user. Those users that are particularly concerned about leaking their top persona can still disable apps from accessing this information.

Permissions: As we saw before, access to persona data in apps is guarded with a runtime permission prompt, similar to that used for obtaining location data. This is superior and more meaningful to the user compared to installation-time permissions [17]. We believe that this is part of responsible disclosure: the user is informed of persona data access and is given the opportunity to opt in. Furthermore, access to MoRePriv's persona data requires statically declared permissions at the level of application manifest. Just like with location data, we can envision extra scrutiny, code reviews, and testing being applied by the app store to apps that request persona data.

Data synchronization: While the default storage strategy is to keep the interest profiles local, on the current device, it is entirely possible to synchronize them — in an encrypted form — with the cloud. This is not unlike the approach used in Apple's iCloud for synchronizing application settings, etc. However, unlike application settings, persona information encroaches on user privacy considerably less. In addition to synchronization across multiple devices, cloud synchronization also serves as a backup. This approach is used in several domains, including bookmark synchronization, etc.

3.4 Value Proposition

Our goal with MoRePriv is to provide an attractive value proposition for all major categories of constituents within the mobile ecosystem: users, developers, and third party data and ad providers. We summarize the benefits below.

Users: Recent user studies indicate that users often prefer client-side personalization to alternatives [29]. Additionally, MoRePriv learns an accurate model of the user's tastes and preferences with-

out the user having to "teach" or configure it. To make it easier for users to understand, we model the way persona information is surfaced to the user after GPS location data; the figure to the right shows an example of a persona prompt within an app. Users can *audit* persona information released to every application, to understand when each release has taken place. Users can opt of of data collection, either partially or entirely. Lastly, for users who are strongly concerned about privacy, MoRePriv is designed to be combined with privacy-enhancing technologies.

Developers: The main advantage for developers is the MoRePriv API. It represents a single OS-level data source for personalization (as opposed to application-specific, ad-hoc information sources) allow seamless and uniform functionality for a single user across many applications, devices, and platforms. It is easy for developers to use for personalization and skinning, as illustrated in Section 6.1. The *cold start* problem [38] common in many personalization tasks is largely addressed, because of an OS-wide user personalization context, which apps can easily take advantage of. Finally, OS-wide data collection for a single repository allows richer and more accurate profile information about the user to be collected, compared to what any single application can accomplish.

Third-party and ad providers: Data providers are no longer responsible for storing "toxic" user data and cross-correlating it across a set of user interactions. Given the mounting pressure by legislators in the US and Europe to limit the impact of online tracking as well as mobile app tracking, this can be welcome news. Instead of trying to engage in user profiling, which is 1) costly, because it requires maintaining or buying data center capacity, developing custom software, and paying data mining and support personnel, and 2) cumbersome, because multiple laws of different localities must be respected, and there is always a danger of being limited later in time, a clear alternative is to use the information provided natively, by the mobile OS [2, 1]. While some third-party provides will certainly prefer to stick with the state of affairs, less entrenched ones will find the new approach attractive.

4. IMPLEMENTATION

In order to test the effectiveness of personal preference miners, we instrumented Windows Phone 7.5 (Mango) to capture several important *personalization signals*, sources of data that indicate likely user preferences. We then use these signals to locally classify the user with respect to the given personae. There are two facilities for personalizers in MoRePriv: a privileged service to perform automatic personalization within the OS, and a set of APIs that give third-party applications limited access to the user interest profile. We chose to implement MoRePriv on top of the Windows Phone operating system. Similar implementations on Android or iOS may be possible, and we point out areas in which we take advantage of features of the Windows Phone and possible analogues in other mobile operating systems. However, cross-platform results are not the focus of our work.

4.1 Personalization Signals

In order to assign relevance scores to each persona, MoRePriv needs data to classify. Here, we leverage our position at the OS level: all user information must pass through the operating system in order to be consumed or produced by the user. However, one must be careful of *which* data stream to collect: a poor choice can slow down the device or introduce noise. One must also consider *how* data is collected: the collection mechanism should be positioned at a level of abstraction such that the data has appropriate context. For example, one may be interested in mining the text of web pages that the user views, but if the miner interposes at a low level of the protocol stack, bytes of text will be indistinguishable from bytes of an image.

In our implementation, we capture five distinct personalization signals: Facebook data, the Twitter feed, SMS messages, emails, and HTTP traffic. We briefly discuss our approach to mining each of these signals on Windows Phone, and we mention how equivalent signals could be captured for an Android Device.

Facebook and Twitter: A unique feature of the Windows Phone is that several popular networking features are integrated directly into the operating system and organized into the *People Hub*. The intention of the People Hub is to organize social updates in a single, unified feed called the social feed, which is updated automatically. The social feed is a good target for mining because it is a rich source of structured user data. We implemented a miner for Facebook by reading social feed data from the Facebook service, consisting of "likes", posts that the user made, and posts made to the user's wall.

There is no direct analogue to the People Hub on Android. However, since the account credentials are stored in centralized Account Manager, an Android device could make separate queries through the APIs exposed by high-value services like Facebook and Twitter, and classify the results of those queries. Note that this approach loses an advantages of MoRePriv, namely that it does not consume additional network bandwidth.

SMS: due to the simplicity and inherent lack of structure in SMS messages, we implemented our miner by interposing on the SMS handler in native code underlying the application framework. Alternatively, similar modifications can be made to the C# core libraries to read SMS messages. To implement a similar miner in Android, one could periodically query the `ContentResolver` for SMS content, and classify each SMS message in turn.

Email: Sending email is exposed to third party users via the `Microsoft.Phone.Tasks` classes such as `EmailComposeTask`. However, in order to simplify capture of both outgoing and incoming email, we instead interpose on the internal implementation of SMTP. Although we do not treat fields such as *subject* and *sender* differently from text in the body, interposing on SMTP allows us to avoid classifying noise, e.g. attached images.

HTTP Traffic: Unlike SMS, HTTP traffic has structure that cannot be ignored. Fortunately, the Windows Phone passes information to an HTTP handler which parses the structure of the message. By interposing on the parser as it parses text, we can gather relevant web text without adding significant noise from non-textual traffic.

These signals demonstrate an advantage of performing signal capture at the Operating System level: since the OS and framework have a very high level of privilege, the user must already trust these components to handle personal data. As such, the signal capture mechanisms are already within the user's trusted computing base. Instrumentation at the OS level has the unique advantage of being able to integrate multiple data sources together. This is important for several reasons. Even very rich data sources can suffer from a cold-start problem, but are useful in aggregate.

To drive this point home, we conducted a study to see how strongly a user with a high interest in technology would be classified as a technophile. While we would expect this score to be high, the quality of different signals in isolation varies significantly. However, *combining* these signals together can boost the correct relevance score even in the face of irrelevant signal data.

4.2 Building MoRePriv Classifiers

Each MoRePriv persona is represented by a Naive Bayesian classifier. We trained our classifiers offline on manually curated lists of words obtained for web pages relevant to each profile, e.g., `techcrunch.com` for technophile and `espn.com` for sports buff. An

alternative source of such pages are taxonomies such as the Open Directory Project (ODP).

This resulted in thousands of words per persona, which were subsequently used to obtain the probabilities $P(w_i|C_j)$ for each attribute word w_i and each persona C_j. This classification data was then loaded into an OS-level service.

Although building each classifier can in principle be a difficult task, especially if a large training set is used, applying classification to a piece of text is very fast. For example, to find the most relevant profile for a piece of text, we tokenize it into words and perform a simple log-likelihood addition for each persona, maximizing over that value.

4.3 Universal Personalization

In the context of MoRePriv, we implemented *automatic* universal personalization within the OS. To accomplish this, we modified the Windows Phone C# framework upon which apps are built. We focused on reordering lists such that elements that are of the most interest to the user are displayed at the top, while items of less interest to the user are kept at the bottom. Widget classes like `System.Windows.Controls.ListBox`, are used in many third party apps to display lists, either directly or via a subclass. Our modification changed the way in which list items are displayed on screen, ordering them by their relevance to the user profile. In essence, this allows lists to be transparently personalized without any modification (or knowledge) of the app. An important consideration for universal personalization is not to personalize "too much". For example, if automatic personalization were to be applied to an alphabetized list, the alphabetic ordering would be lost. Thus, we detect if a list has been sorted, and if so we do not use the personalized list ordering to draw elements, instead relying on the ordering of the internal `ItemSource` list.

We have made a small number of changes to the Windows Phone core C# classes to inform our universal personalization mechanism that a list has been sorted, such as modifying the `List.Sort()` method to keep a `sorted` flag. When set, the flag causes UI classes like `ListBox` to display in their natural order. We emphasize that universal personalization is often effective, but recognize that automatically restructuring UI is not always guaranteed to work. For this reason, we allow universal personalization to be disabled through the MoRePriv management menu.

4.4 OS-level Service

Positioning MoRePriv within the OS provides an opportunity to collect a great deal of data to build a user interest profile. However, it also provides an opportunity to perform personalization on user-level apps without any modification of the app itself.

To explore automatic personalization, we altered the Windows Phone C# framework to reorder lists in the application UI, as described in Section 4.3, based on the persona weights. For legacy applications such as news readers, this has the effect of not only reordering the order in which stories are displayed (stories more relevant to the user's interests are shuffled to the top), but also reordering entire categories of subjects such that the "technology news" category page of a news reader app appears earlier in the menu than the "arts section" for a technophile. Finally, users can control how MoRePriv is used by toggling two independent facilities:

Toggling personalization: When personalization is off, the user will have the regular Windows Phone experience. When personalization is on, OS-level personalization is enabled and apps have access to the user's persona.

Toggling collection: This allows users to to freeze their profile scores. This indicates to MoRePriv that it should not track any behavior of the user until persona refinement is re-enabled. This is a form of a privacy mode, similar to those supported in modern browsers.

4.5 MoRePriv APIs

MoRePriv also exposes APIs to third party developers that allow application-specific personalization. We discuss three of these API functions:

- `IsMoRePrivEnabled()` returns true if personalization is enabled. We allow users to toggle personalization on and off as part of the MoRePriv configuration UI.
- `TopProfile()` return the most relevant profile to the user if personalization is enabled, and `null` otherwise. This is useful for application *skinning*, for example by changing the background of the app based on the top profile of the user.
- `Ignore(Object o)` Informs MoRePriv not to apply OS-level personalization to `o`. This allows developers to bypass the GUI features such as automatic list reordering.

We have also written a wrapper library that checks for these API functions and calls them if present. Thus, app developers can write a single app that will work on both MoRePriv-enabled and plain OSes (albeit without personalization).

5. INTEGRITY AND EXTENSIBILITY

There is growing evidence that the need for trusted user identities in advertising is growing[4]. These cases are emblematic of the commonly occurring case in which a 3rd party requires knowledge that the classification of users is accurate and trustworthy. A signature-based scheme may prevent benign users from loading hacked classifiers, but when it comes to malicious users, no such recourse is available. While the operating system itself is trusted by the user, the 3rd party may not trust the integrity of apps and classifiers: malicious users can install or "sideload" a hacked classifier to send fabricated data.

Instead, we extended MoRePriv to optionally use zero- knowledge proofs to provide a solution to the problem of *integrity* of user profiling, effectively guaranteeing to third parties that the personalization signal can be trusted through *checkable* proofs of computational integrity.

5.1 Design Choices

In some cases, the information provided by the default classifiers is not sufficient for personalization. However, while we want to support greater extensibility, we do not want to expose sensitive user data to untrusted 3rd parties.

In this paper, we consider two ways to architect the system we have described. The *monolithic* architecture provides only the built-in, default user classifier. However, we acknowledge that in some settings, more application-specific classifiers may be called for. Since classifiers may need to be upgraded as a result of retraining, the *customizable classifier* approach allows for custom classifiers to optionally be installed at the discretion of the user.

Monolithic: The model discussed in this paper thus far assumes a single monolithic set of MoRePriv classifiers which are implemented as an operating system service. As such, they are part of the TCB. A consumer of the user profile can trust the profile computation under the assumption that the operating system has not been tampered with. We feel that this is a sensible assumption,

[4] See `http://bit.ly/13hJpNk`, `http://tcrn.ch/PR5ZDL`, `http://bit.ly/1bT5mFD`, `http://on.wsj.com/1e57ms1`, `http://bit.ly/1bT5mFD`, and [8] for discussions of advertising fraud and its implications.

since tampering with the OS can invalidate the integrity of the observations on the basis of which the user profile is calculated. The main shortcomings of this monolithic approach is the difficulty in upgrading the classifiers and the fact that that classifiers are part of the TCB.

Customizable: An alternative model discussed in the rest of this section allows for custom classifiers to be installed in the same way that regular apps are installed, without affecting the OS and adding to the TCB. The user is responsible for checking installation- and runtime permissions to ensure these classifiers do not leak data inappropriately, just as they are for any other mobile app. Alternatively, as proposed in several recent projects [46, 26, 14, 16, 25, 31], classifiers can be subjected to static or runtime analysis to gain further assurance.

5.2 Deploying Custom Classifiers

In our model, custom classifiers are distributed as third-party applications in the same manner as other third-party software on mobile platforms. When the user installs a classifier application, he is prompted for permission to let the app interact with MoRePriv by monitoring keywords among the personalization signals discussed in Section 4.1; this permission is requested at the same time as the other platform-specific permissions supported by the operating system. When the classifier is installed, MoRePriv registers the needed keywords that are listed in the classifier's manifest.

To perform classification, the application queries MoRePriv via an exposed API that returns the number of times each feature word was observed in the signals as a vector $\mathbf{w} = [w_0, \ldots, w_n]$. After receiving an updated set of word counts, the classifier runs its custom algorithm, and uses the result as needed. Whether this classification step is performed *periodically* in the background, or *on-demand* when requested, is an application-specific design choice left up to the classifier developer. Similarly, the choice of when to provide a proof of integrity is left to the developer; it can happen each time a new classifier is constructed (for an absolute guarantee of integrity), or only when the classifier consumer decides to *challenge* the client (for a probabilistic guarantee).

5.3 Motivating Example

Suppose that a third-party service, such as AdMob, needs to determine a characteristic about the user that it cannot derive from the built-in personae. For example, it may want to determine if the user is a student, in order to determine whether to display advertisements about spring break vacation packages. In this situation, AdMob can utilize a classifier developed in-house, which is installed as a user-level application on the user's device much as a third-party ad library is on existing mobile platforms.

This is a compelling example of securely-managed cooperation between MoRePriv and a third-party that wishes to personalize, but it exposes an important concern for the third party. Namely, because AdMob's algorithm is running on the user's device in an unprotected mode, AdMob has no assurance that the results it obtains correspond to the true results of its algorithm. For example, a user who wishes to consume ad-supported content without providing information about themselves may simply install a "dummy" application in place of AdMob's, that returns random results.

In this section, we discuss a strategy for balancing this concern with our central focus of privacy based on recent advances in *Non-Interactive Zero-Knowledge (NIZK) Proofs of Knowledge* [7]. We provide integrity using the same practical assumptions as Danezis *et al.* [13] and Schnorr [41], which roots trust in the operating system, but does not require trusted hardware.

We discuss an implementation of the AdMob scenario given above that uses zero-knowledge proofs based on the construction of [39], and show how it can be efficiently parallelized to achieve acceptable performance. While our implementation is application-specific, the approach can be generalized without requiring custom zero-knowledge proof development using one of the many zero knowledge proof compilers [3, 4, 33].

5.4 Achieving Computational Integrity

In this section, we begin by giving a brief general overview of NIZK schemes, and then describe our realization in the context of MoRePriv.

Background and Overview: In a NIZK scheme, one party (the *prover*) attempts to convince another (the *verifier*) that it has correctly performed a computation that is known to both parties, without revealing the inputs that it provides to the computation. Oftentimes, the inputs are generated signed by a *data provider* that is trusted by the verifier, in order to establish an initial root of integrity. In our setting, the prover corresponds to a classifier, the verifier to a service that consumes the classifier's output, and the data provider to the MoRePriv core.

Our NIZK scheme makes use of a non-interactive commitment scheme, which consists of the algorithms *Commit* and *Open*[5]. Given a value x, $Commit(x)$ produces a *commitment* c_x as well as a piece of auxiliary data o_x called the *opening*. The commitment is *opened* by revealing (c_x, o_x), and checking that $Open(x, o_x, c_x)$ returns *true*. Commitments have two useful properties: *(1) binding*, meaning that *Open* only returns *true* when $(c_x, o_x) = Commit(x)$, and *(2) hiding*, meaning that a party who possesses only c_x should not be able to learn anything about x. Abusing notation somewhat, we use the phrase *signed commitment*, and write $Commit_{K_A}(x)$, to refer to the commitment of x signed with A's public key. MoRePriv provides inputs to the classifier via API calls, which can later be converted into signed commitments by the operating system when the consumer service requires a proof of integrity for some output of the classifier. The integrity of these inputs comes from the consumer service's trust in the integrity of the operating system, i.e., that it has not been compromised.

The scheme that we use requires the following entities:

- The computation $f(x_{\mathsf{pub}}, x_{\mathsf{priv}})$, which takes input known to both parties x_{pub} as well as input private to the prover x_{priv}.
- The inputs x_{pub} and x_{priv}, as well as signed commitments $Commit_{K_{TP}}(x_{\mathsf{priv}})$ to the private inputs, using the provider's public key K_{TP}.

As illustrated in Figure 1, it proceeds as follows:

- The data provider sends the prover its private input x_{priv}, as well as the signed commitment $S = \mathrm{Commit}_{K_{TP}}(x_{\mathsf{priv}})$.
- The prover runs the computation on both sets of inputs to learn $r = f(x_{\mathsf{pub}}, x_{\mathsf{priv}})$. It constructs a zero-knowledge proof of knowledge $P = \mathrm{ZKProof}(f, x_{\mathsf{pub}}, x_{\mathsf{priv}}, r, S)$ which demonstrates that $\exists\, x_{\mathsf{priv}}.r = f(x_{\mathsf{pub}}, x_{\mathsf{priv}})$ and $S = \mathrm{Commit}_{K_{TP}}(x_{\mathsf{priv}})$. Finally, the prover sends P and r to the verifier.
- The verifier checks P against S and r, which can be done in roughly the same amount of time taken by the prover to generate P and compute r.

We refer the reader to the explanation of these schemes given by [7] for more details.

[5] For simplicity, we omit certain details relating to security parameters and setup. For a detailed explanation, the reader should refer to [7, 39]

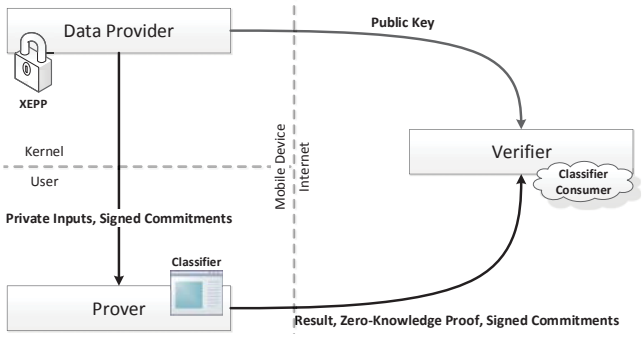


Figure 1: Zero-knowledge proofs for computational integrity.

5.5 Evaluation

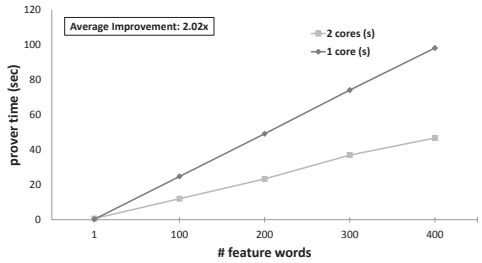
We evaluated our zero-knowledge classifier on a mobile device to determine its runtime, network utilization, and impact on battery life. In general, we found our implementation to be quite practical. Performance is parametrized by the number of feature words in the classifier.

For the student classifier in this section, based on our practical experience with Bayesian classification, we believe that using between 100 and 300 feature words gives sufficient precision. Note that we do not generally anticipate every single request requiring a proof; a scenario where a third party would periodically *audit* the result is more likely, making the numbers reported below a performance upper bound. Moreover, because the prover can run in idle time, when the phone is unused, and not on demand, we believe this represents more than acceptable performance.

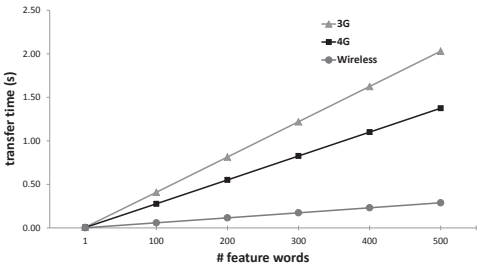
Setup: Unless stated otherwise, all experiments were performed on a Nokia Lumia 920 running Windows Phone 8. Experiments involving networking were performed on a small wireless network comprised of the mobile device, a laptop (with an Intel 4965AGN chipset), and a dedicated router not connected to an external network. The Zero-Knowledge protocol was implemented in F# and C#, using the .NET infinite-precision integer library. For commitments, we used a 1024-bit modulus; for signatures we used SHA1 with RSA, again with a 1024-bit modulus. All experiments used eight personae, unless otherwise noted.

Basic measurements: Figure 2(a) shows the time taken on the mobile device to compute the classifier result for all personae against the number of feature words in each persona's set of Bayesian classifier. Both cores of the mobile device were utilized without restriction. As shown, the time for a classifier size of 300 words is just under five minutes. Since our current personae classifiers are of approximately 300 words, we believe this to be a reasonable overhead. Profiling reveals that nearly all processing time is devoted to generating cryptographically-secure random noise (for commitment generation), and integer exponentiation (for the group operations). This means that while it is not feasible to produce a new, up-to-date zero-knowledge proof on-demand, one could perform these computations in the idle time after a period of user interaction with the phone.

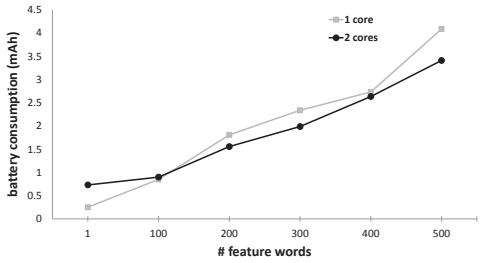
Figure 2(b) shows the network transfer time required to send the proof and input commitments against the number of feature words. To take these measurements, we simulated 3G and 4G cellular networks by modulating throughput, latency, and packet loss rate. Average throughput in each measured configuration (3G, 4G, and Wifi) was 134.4, 198.4, and 945.8 kilobytes per second, respectively. We observed proofs of less than 200 K that it takes 0.41–1.22 seconds to transmit the setup and proof for a 100 and 300-feature word classifier, respectively, on an "average" 3G network. We also



(a) Time to run the prover.



(b) Transmission time for the proof.



(c) Battery usage.

Figure 2: Custom classifier performance metrics

observed reasonably proof sizes of less than 200 KB for even the largest (300 feature-word classifier).

Figure 2(c) shows the amount of battery draw required by the classifier module, measured against the number of feature words in the classifier. This was computed based on CPU utilization, according to the model of Mittal *et al.* [34]. The power required to produce a proof goes down when multiple cores are available, and there is sufficient work available for the second core; this trend begins shortly after 100 feature words. To give these readings some perspective, we assume a 24-hour battery charge cycle in order to compute the amount of "battery time" consumed by the prover. Our test phone (Nokia Lumia 920) advertises a built-in 2,000 mAh battery, so producing a proof for a 300-word classifier consumes 0.099% of the available battery, about 40 seconds.

Summary: We believe our scheme to be practical with today's cryptographic primitives and mobile hardware in terms of computation and transmission time. With 100 feature words, the prover time is about 11 seconds. Transfer time of the proof for 100 feature words is under a second on an average 3G connection. Battery utilization (under 1 mAh) is negligible.

6. PERSONALIZATION

MoRePriv offers many opportunities for personalization. In this section, we focus on several personalization scenarios in depth, but note that this is only the tip of the iceberg. Other prominent examples include keying custom dictionaries for word completion,

News topic	activist	bachelor	business exec	sports buff	retiree	homemaker	technophile	tween
Health	3	3	4	4	10	7	2	1
Tech	4	6	5	5	3	4	10	8
US	9	4	7	6	6	4	3	2
Business	4	5	10	5	5	2	6	1
World	7	2	4	1	2	2	2	1
Entertainment	0	7	3	4	4	6	5	5
Science	2	3	3	1	3	2	6	4
Society	6	3	2	2	3	3	2	3
Politics	10	4	6	5	5	3	2	1
Sports	0	7	4	10	4	2	5	7

Figure 3: News personalization parameters.

spell-checking, and voice recognition on the persona.

6.1 Personalization Scenarios

RSS News Feed Personalization: To test the usefulness of the MoRePriv APIs, we built a custom RSS reader called MoRSS. This app pulls stories from 10 RSS news feeds, and samples from these feeds to display a list of stories to the user. MoRSS disables the OS-level GUI enhancements described in Section 4. Instead, MoRSS relies on the built-in table in Figure 3 to rate how interesting each feed will be to each persona.

MoRSS can operate with no personalization, in which case stories from each RSS feed will be sampled uniformly and displayed to the user in the order in which they are sampled. When personalization is enabled, MoRSS queries the MoRePriv API to determine the top persona of the user's profile, and then samples according to the column of Figure 3 for that persona.

MoRSS demonstrates the advantages of exposing limited information to third party applications: Developers have the flexibility to reinterpret the top profile in any way that they see fit. Apps such as MoRSS are free to sample tech stories for the homemaker, even though the built-in Bayesian classifier for that profile does not have tech keywords. Furthermore, the personalization can be done in a privacy-preserving way: MoRSS uses client-side personalization, so even the owner of the RSS feeds cannot learn the top profile of the user from the requests that MoRSS makes.

Application Skinning: OS-level personalization has broad applicability, even beyond networked device use. As a demonstration, we implemented a simple calculator that is re-skinned using the MoRePriv API for each persona. For instance, when the top profile is retiree, the calculator goes into a high contrast mode with large text and buttons.

We highlight this example to show that MoRePriv provides an alternative to forcing complicated configuration menus on users who nonetheless prefer different configurations. An app like the calculator could use profile data to provide an initial configuration that is likely to be close to what the user wants, and allow her to tweak configuration options from there.

6.2 Personalization Case Study: Yelp

The goal of the rest of the section is to demonstrate that persona-level personalization is useful and would be appreciated, were it provided by the operating system natively. However, effective personalization is not an easy task. It is both an art and a science, requiring a combination of technical and design skills, deep understanding of the target market, and the particulars of the application. Thus, we believe that a dedicated effort to create such a service would have even better results.

	Chicago	LA	Madison	NY	SF	Seattle
General	50%	33%	56%	50%	50%	39%
Executive	63%	76%	50%	53%	68%	47%
Boost	13%	43%	-6%	3%	18%	8%

Figure 4: Percentage of survey responders who prefer the executive personalization over the default one.

Personalizing Yelp: For our personalization target, we have chosen business recommendations using the Yelp API to get location-aware reviews and listings for businesses such as restaurants, hair salons, doctors, etc. popular in the US. The app includes a subset of the Yelp categories at `http://bit.ly/441TNj` relevant to a persona. When run, it pulls in a large number of Yelp listings, and prioritizes those in categories relevant to the top profile.

We have manually curated a list of 28 categories in the Yelp taxonomy (out of about 700) for business executive, intended to suggest an activity that an executive might bring business associates to. Categories fall into a more upscale niche, such as *steakhouses*, *wine bars*, or *auction houses*. To personalize, we sort the full set of Yelp results returned by reviewer rating "stars". We then search for a listing in a category relevant to the top persona not already selected for display. If there are no such listings, we select the highest ranked remaining listing in a category relevant to the top persona (even if we have already selected another listing in that category). We note that there are many additional features upon which to select listings, such as price, ambiance, availability of Wi-Fi, etc. Our experiment does not use these features, as doing so violates the Yelp API terms of use. Such restrictions could be ignored by an official Yelp app. In a general sense, we assume that the business executive persona applies at all times, which may not always be valid [12].

Methodology: Using the Instant.ly crowd-sourcing platform, we hired survey takers to answersix preference questions pertaining to the following cities: Chicago, LA, Madison, NY, SF, and Seattle. A sample survey can be seen at `http://www.instant.ly/s/B55pX`. One question was asked for every city, phrased as follows: *We are trying to understand which listing is more relevant for YOU. In other words, where are you more likely to go to and spend money at? Which listing below do you find more relevant and why?*

Of course, any such survey measures *expressed* preferences, not *actual* preferences, as noted in [36, 42]. However, short of deploying a modified app in the wild and watching users (who would have to opt into the study), this is a good initial assessment.

Selecting users: Using the Instant.ly built-in crowd, we created two survey groups *general* (21 respondents) and *executive* (41 respondents). Executives were selected based on the following criteria: age of 35–65, education of college graduate with a four-year degree or an advanced degree, employed full-time or self-employed, income in excess of $125,000, and married or living with a partner. This is a fairly simplistic way to reach that demographic, but we believe it suffices for our experiments. Given that these listings are somewhat location-specific, we tried to avoid location biases that might have been present in the recruiting process.

Results: Figure 4 presents the results of our study. We show preference for the executive treatment over the generic Yelp results for the general user group and the executive user group, reflecting the percentage of users who prefer the executive personalization. Overall, the *boost*, i.e. the difference in preference for executive in the executive demographics vs. the general population is most pronounced for Los Angeles (43%) and San Francisco (18%). The average boost across the cities is about 30%, which indicates that, on average, the executive personalization is quite effective.

7. RELATED WORK

Most efforts in the rapidly-growing space of mobile privacy research focus on detection and prevention techniques for information leaks, especially those in Android mobile apps [45, 26, 43, 15, 35]. TaintDroid [16] and Aquifer [35] are representative projects in this space, whose full overview we omit for space reasons. Our work takes a complementary position: leak detection tools prevent malicious use of private data, while MoRePriv offers devlopers a route to use private data in a way that does not damage the privacy of the user.

Resolving the tension between *extensible* personalization and privacy is not a new problem, which we attempt to address with our custom classifiers mechanism. Much work has been done on *verified extensions*, in contexts as diverse as OS drivers and browser extensions. The problem of checking extensions for privacy leaks can be addressed via static analysis [6], type systems [21], and runtime information flow tracking. This kind of validation can be integrated with MoRePriv or applied ahead of time by app store maintainers, but validating extensions is not the focus of our work. In contrast to many extensible systems, we do not require that custom classifiers be written in a special language, as, for example in [18].

MoRePriv falls into the category of efforts advocating client-side computing as a way to enhance user privacy. In the context of ad delivery, several researchers advocate remedying the problem by storing the necessary sensitive personal data on the client, along with all ads to be matched [23, 44, 27]. When an ad is displayed, it is matched to personal information locally, thus sidestepping the need to leak to the ad network. Accounting and click-fraud prevention are addressed using either additional semi-trusted parties, or homomorphic encryption. Although MoRePriv can be used in the context of personalized advertising, it has broader applicability than advertising alone.

RePriv [18] explored personalization opportunities in the context of a web browser by building a *user interest profile* based on the user's browsing history. RePriv profiles classify the sites the user visits based on the Open Directory Project (ODP) taxonomy of interests. Unlike RePriv, we focus on the mobile space. We go further by integrating our system directly into the operating system, drawing from more diverse data sources, and provide integrity through zero-knowledge proofs.

Both MoRePriv and RePriv are representative of an expanding line of work focusing on client-side computing for privacy. There are a number of notable differences between RePriv and MoRePriv: 1) MoRePriv user data is very coarse-grained, and its disclosure to apps is limited. This persona-based approach is substantially different than the taxonomy-based approach of RePriv; 2) MoRePriv does **not** impose a development language and an onerous-to-use type system on the developer; 3) MoRePriv alone provides automatic personalization for legacy apps. The focus on integrity and the use of Zero-Knowledge proof techniques is specific to MoRePriv as well; 4) Since MoRePriv is at the OS level, it has access to different, rich sources or data - personalization approaches for the browser and OS are substantially different.

Zero-knowledge proofs [5, 7, 13] have seen extensive use in the privacy and applied cryptography literature. Zero-Knowledge protocols have been developed for proving linear relations [9], equality and inequality [41], logical connectives [9], multiplication [10], division and modulo [11], and set membership [10]. Taken together, this work allows the construction of a zero-knowledge proof that expresses the functionality of an arbitrary circuit, as in the case of fully-homomorphic encryption [19].

8. CONCLUSIONS

This paper proposes operating system-level mechanisms that support automatic personalization of legacy applications, and simplify building new personalized applications. Our focus is primarily on mobile operating systems. Implementing a tool at this level allows us leverage a broad variety of user activity, and distill it into a trusted source of personalization information. We believe that at the level of the operating system these opportunities are largely untapped. We demonstrate that personalization can be done quickly and effectively using personae, which provide a degree of pseudonymity and are easy for users and developers to understand.

We show how both OS-wide universal personalization and custom personalization can be done with little effort by the developer, making us hope that persona and location information can become equally ubiquitous on mobile devices. We also demonstrate and experimentally evaluate how MoRePriv supports *integrity* of custom classifiers with the help of zero-knowledge proofs. With a classifier using 100 feature words, the prover time is about 11 seconds, the time to transfer the proof is under a second on an average 3G connection, and the battery utilization is negligible — under 1 mAh. Our results show that trusted classification is feasible with today's zero-knowledge techniques.

Finally, we performed a crowd-sourced user study showing that MoRePriv-style personalization improves the relevance of Yelp results, with business executives preferring MoRePriv personalized results most of the time (a boost of ~30%). In our experiments we have observed cases where the general population and users with a particular persona disagree on what is most relevant, highlighting the need for user profiling and personalization.

We believe MoRePriv is timely and can enable game-changing innovation around privacy through client-side computing.

9. REFERENCES

[1] Application privacy. `http://www.applicationprivacy.org/?page_id=39`.

[2] Mobile app privacy policies are now the law. `http://www.truste.com/blog/2012/03/02/mobile-app-privacy-policies-are-now-the-law`.

[3] J. B. Almeida, E. Bangerter, M. Barbosa, S. Krenn, A.-R. Sadeghi, and T. Schneider. A certifying compiler for zero-knowledge proofs of knowledge based on σ-protocols. In *Proceedings of the European Conference on Research in Computer Security*, 2010.

[4] M. Backes, M. Maffei, and K. Pecina. Automated synthesis of privacy-preserving distributed applications. In *Proceedings of the Network and Distributed System Security Symposium*, 2012.

[5] J. Balasch, A. Rial, C. Troncoso, B. Preneel, I. Verbauwhede, and C. Geuens. Pretp: privacy-preserving electronic toll pricing. In *Proceedings of the Usenix Conference on Security*, 2010.

[6] 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 European Conference on Computer Systems*, 2006.

[7] M. Bellare and O. Goldreich. On defining proofs of knowledge. In *Proceedings of the International Cryptology Conference on Advances in Cryptology*, 1993.

[8] R. Bhaskar, S. Guha, S. Laxman, and P. Naldurg. Verito: A Practical System for Transparency and Accountability in Virtual Economies. In *Proceedings of the 20th Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb 2013.

[9] S. Brands. Rapid demonstration of linear relations connected by boolean operators. In *Proceedings of the International Conference on Theory and Application of Cryptographic*

Techniques, 1997.

[10] J. Camenisch, R. Chaabouni, and A. Shelat. Efficient protocols for set membership and range proofs. In *Proceedings of the International Conference on the Theory and Application of Cryptology and Information Security: Advances in Cryptology*, 2008.

[11] J. Camenisch and M. Michels. Proving in zero-knowledge that a number is the product of two safe primes. In *Proceedings of the 17th international conference on Theory and application of cryptographic techniques*, 1999.

[12] L. F. Cranor. Designing personalized user experiences in ecommerce. In *Proceedings of the 2003 ACM Workshop on Privacy in the Electronic Society*, 2004.

[13] G. Danezis, M. Kohlweiss, B. Livshits, and A. Rial. Private client-side profiling with random forests and hidden Markov models. In *Proceedings of the International Conference on Privacy Enhancing Technologies*, 2012.

[14] M. Egele, C. Kruegel, E. Kirda, and G. Vigna. PiOS: Detecting privacy leaks in iOS applications. In *Proceedings of the Network and Distributed System Security Symposium*, Feb. 2011.

[15] S. Egelman, A. P. Felt, and D. Wagner. Choice architecture and smartphone privacy: There's a price for that. In *WEIS*, 2012.

[16] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. In *Proceedings of the Usenix Conference on Operating Systems Design and Implementation*, 2010.

[17] A. P. Felt, E. Ha, S. Egelman, A. Haney, E. Chin, and D. Wagner. Android permissions: User attention, comprehension, and behavior. In *Proceedings of SOUPS*, 2012.

[18] M. Fredrikson and B. Livshits. RePriv: Re-envisioning in-browser privacy. In *IEEE Symposium on Security and Privacy*, May 2011.

[19] C. Gentry. Fully homomorphic encryption using ideal lattices. In *Proceedings of the ACM Symposium on Theory of computing*, 2009.

[20] M. Grace, W. Zhou, X. Jiang, and A.-R. Sadeghi. Unsafe exposure analysis of mobile in-app advertisements. In *Proceedings of the Conference on Security and Privacy in Wireless and Mobile Networks*, Apr. 2012.

[21] A. Guha, M. Fredrikson, B. Livshits, and N. Swamy. Verified security for browser extensions. In *IEEE Symposium on Security and Privacy*, May 2011.

[22] S. Guha, B. Cheng, and P. Francis. Privad: practical privacy in online advertising. In *Proceedings of the Usenix Conference on Networked systems design and implementation*, 2011.

[23] S. Guha, A. Reznichenko, K. Tang, H. Haddadi, and P. Francis. Serving Ads from localhost for Performance, Privacy, and Profit. In *Proceedings of Hot Topics in Networking*, Nov. 2009.

[24] S. Han, J. Jung, and D. Wetherall. A study of third-party tracking by mobile apps in the wild. Technical report, University of Washington, Mar. 2012.

[25] P. Hornyack, S. Han, J. Jung, S. Schechter, and D. Wetherall. These aren't the droids you're looking for: Retrofitting Android to protect data from imperious applications. In *Proceedings of the International Symposium on Information, Computer, and Communications Security*, 2011.

[26] Y. Z. X. Jiang. Detecting passive content leaks and pollution in android applications. In *Proceedings of the 20th Network and Distributed System Security Symposium (NDSS)*, Feb 2013.

[27] A. Juels. Targeted advertising ... and privacy too. In *Proceedings of the Conference on Topics in Cryptology*, Apr. 2001.

[28] A. Kobsa. Privacy-enhanced personalization. *Commununications of the ACM*, 50(8), Aug. 2007.

[29] A. Kobsa, B. Knijnenburg, and B. Livshits. Let's do it at my place? attitudinal and behavioral study of privacy in client-side personalization. In *Proceedings of the ACM CHI Conference on Human Factors in Computing Systems Proceedings*, Apr. 2014.

[30] A. Kobsa and J. Schreck. Privacy through pseudonymity in user-adaptive systems. *ACM Transactions Internet Technologies*, 3(2), May 2003.

[31] B. Livshits and J. Jung. Automatic mediation of privacy-sensitive resource access in smartphone applications. In *Proceedings of the Usenix Conference on Security*, 2013.

[32] J. R. Mayer and J. C. Mitchell. Third-party Web tracking: Policy and technology. In *IEEE Symposium on Security and Privacy*, May 2012.

[33] S. Meiklejohn, C. C. Erway, A. Küpçü, T. Hinkle, and A. Lysyanskaya. Zkpdl: a language-based system for efficient zero-knowledge proofs and electronic cash. In *Proceedings of the Usenix Conference on Security*, 2010.

[34] R. Mittal, A. Kansal, and R. Chandra. Empowering developers to estimate app energy consumption. In *Proceedings of the International Conference on Mobile Computing and Networking*, 2012.

[35] A. Nadkarni and W. Enck. Preventing accidental data disclosure in modern operating systems. In *ACM Conference on Computer and Communications Security*, pages 1029–1042, 2013.

[36] P. Norberg, D. Horne, and D. Horne. The privacy paradox: Personal information disclosure intentions versus behaviors. *Journal of Consumer Affairs*, 41(1), 2007.

[37] L. Pareschi, D. Riboni, A. Agostini, and C. Bettini. C.: Composition and generalization of context data for privacy preservation. In *In: Sixth Annual IEEE International Conference on Pervasive Computing and Communications (PerCom 2008), Proceedings of the Workshops, IEEE Computer Society*, 2008.

[38] A. M. Rashid, G. Karypis, and J. Riedl. Learning preferences of new users in recommender systems: An information theoretic approach. *SIGKDD Explor. Newsl.*, 10(2):90–100, Dec. 2008.

[39] A. Rial and G. Danezis. Privacy-preserving smart metering. In *Proceedings of the Workshop on Privacy in the Electronic Society*, 2011.

[40] E. Rich. User modeling via stereotypes. *Cognitive Science*, 3, 1979.

[41] C.-P. Schnorr. Efficient signature generation by smart cards. *Journal of Cryptology*, 4, 1991.

[42] S. Spiekermann, J. Grossklags, and B. Berendt. E-privacy in 2nd generation e-commerce: privacy preferences versus actual behavior. In *Proceedings of the 3rd ACM conference on Electronic Commerce*, EC '01, 2001.

[43] R. Stevens, C. Gibler, J. Crussell, J. Erickson, and H. Chen. Investigating user privacy in android ad libraries. In *MoST 2012: Mobile Security Techologies (2012)*, 2009.

[44] V. Toubiana, A. Narayanan, D. Boneh, H. Nissenbaum, and S. Barocas. Adnostic: Privacy preserving targeted advertising. In *Proceedings of the Network and Distributed System Security Symposium*, Feb. 2010.

[45] N. Vallina-Rodriguez, J. Shah, A. Finamore, Y. Grunenberger, K. Papagiannaki, H. Haddadi, and J. Crowcroft. Breaking for commercials: characterizing mobile advertising. In *Proceedings of the 2012 ACM conference on Internet measurement conference*, 2012.

[46] Y. Zhou, X. Zhang, X. Jiang, and V. W. Freeh. Taming information-stealing smartphone applications (on android). In *Proceedings of the 4th international conference on Trust and trustworthy computing*, TRUST'11, pages 93–107, Berlin, Heidelberg, 2011. Springer-Verlag.

IMSI-Catch Me If You Can: IMSI-Catcher-Catchers

Adrian Dabrowski
SBA Research
Vienna, Austria
adabrowski@sba-research.org

Nicola Pianta
Università di Cagliari
Cagliari, Italy
nicola.pianta@gmail.com

Thomas Klepp
Vienna University of Technology
thomas.klepp@student.tuwien.ac.at

Martin Mulazzani
SBA Research
Vienna, Austria
mmulazzani@sba-research.org

Edgar Weippl
SBA Research
Vienna, Austria
eweippl@sba-research.org

ABSTRACT

IMSI Catchers are used in mobile networks to identify and eavesdrop on phones. When, the number of vendors increased and prices dropped, the device became available to much larger audiences. Self-made devices based on open source software are available for about US$ 1,500.

In this paper, we identify and describe multiple methods of detecting artifacts in the mobile network produced by such devices. We present two independent novel implementations of an *IMSI Catcher Catcher* (ICC) to detect this threat against everyone's privacy. The first one employs a network of stationary (sICC) measurement units installed in a geographical area and constantly scanning all frequency bands for cell announcements and fingerprinting the cell network parameters. These rooftop-mounted devices can cover large areas. The second implementation is an app for standard consumer grade mobile phones (mICC), without the need to *root* or *jailbreak* them. Its core principle is based upon geographical network topology correlation, facilitating the ubiquitous built-in GPS receiver in today's phones and a network cell capabilities fingerprinting technique. The latter works for the vicinity of the phone by first learning the cell landscape and than matching it against the learned data. We implemented and evaluated both solutions for digital self-defense and deployed several of the stationary units for a long term field-test. Finally, we describe how to detect recently published denial of service attacks.

1. INTRODUCTION

IMSI Catchers are MITM (man in the middle) devices for cellular networks [22]. Originally developed to steal IMSI (International Mobile Subscriber Identity) numbers from nearby phones (hence the name), later versions offered call- and message interception. Today, IMSI Catchers are used to track handsets, deliver geo-target spam [26], send operator messages that reconfigure the phone (e.g. installing a permanent MITM by setting a new APN, http-proxy, or attack the management interface [35]), directly attack SIM cards with encrypted SMS [28] that are filtered by most operators by now, and can potentially intercept mobile two-factor authentication schemes (mTAN). Pell and Soghoian [31] argue, that we are currently on the brink of age where almost everyone could eavesdrop phone calls, similar to the 1990es where cheap analog scanners where used to listen to mobile phones in the US and Europe.

In brief, these devices exploit the phone's behavior to prefer the strongest cell phone tower signal in vicinity to maximize the signal quality and minimize its own power consumption. Additionally, on GSM networks (2G), only the phone (via the SIM, Subscriber Identification Module) needs to authenticate to the network but not vice versa and therefore can easily be deluded to disable content data encryption. This enables an attacker to answer a phone's requests as if the phone was communicating with a legitimate cell phone network.

In contrast, the Universal Mobile Telecommunication System (UMTS, 3G) requires mutual two-way authentication, but can be circumvented using the GSM compatibility layer present in most networks [25], or mobiles can be forced to downgrade to a 2G connection by other means. Additionally, network operators use GSM as a fallback network where UMTS is not available. This makes GSM security still relevant and important in today's mobile network world.

The main contributions are structured as follows. A survey of network level artifacts caused by an IMSI Catcher are described in Section 4. In Section 5 we present a concept of a usable and customer grade warning system. Therefore, we determination which detection methods are available and implementable with what consumer grade hardware in Section 6. We present our implementation and the evaluation of these methods in Section 7. Finally, we describe and evaluate the detectability of large scale denial of service attacks such as [18] in Section 9.1 before we summarize our findings in Section 10.

2. MOTIVATION

The first IMSI Catchers date back as early as 1993 [34] and were big, heavy, and expensive. Only a few manufacturers existed and the economic barrier limited the device's use mostly to governmental agencies. However, in recent years, a number of smaller and cheaper as well as self-built projects appeared making cellular network snooping attacks feasible

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.

ACSAC'14, December 08 - 12 2014, New Orleans, LA, USA
Copyright is held by the authors. Publication rights licensed to ACM.
ACM 978-1-4503-3005-3/14/12 ...$15.00.
http://dx.doi.org/10.1145/2664243.2664272.

to much larger audiences.

Chris Paget built an IMSI Catcher for about US$1,500 [10] and presented it at DEFCON 2010. His setup consists of a *Software Defined Radio* [13] and free open source software such as GNU Radio, OpenBTS, and Asterisk. Several other (academic) projects also built such devices [32, 39] based on similar setups. Appropriate patches and configuration guides are publicly available.

In 2010, Nohl and Manaut [27, 29] presented practical snooping attacks on GSM's main cipher suite using custom firmware on modified mobile phones. However, such a solution can only monitor a very small number of frequencies at once and is likely to lose the intercepted phone on handovers to other cells. Therefore, a professional attacker will still use IMSI Catcher-like functionality to lock the radio channel.

As IMSI Catchers perform an active radio attack, we put forward multiple passive ways to detect such an attack, both stationary and mobile. We facilitated ordinary mobile phones or easily acquirable hardware. This allows for easy deployment of the described techniques for end users or interested hobbyists. We therefore intentionally chose to exclude expensive protocol analyzers or complex self-built solutions.

3. BACKGROUND

In general a mobile network consists of *base stations* (BS) that use one or more radio interfaces to create geographically limited radio *cells*. Multiple cells of an operator are grouped to *Location Areas*. After power-up or when a *mobile station* (MS, e.g. phone, data modem) lost connection to its network, it will perform a *full scan* to find the frequencies of nearby cells based on *beacon* signals sent out by every cell on a regular basis. The MS registers into the operators network using its worldwide unique *International Mobile Equipment Identity* (IMEI), its *International Mobile Subscriber Identity* (IMSI) number and a secret key stored on the *Subscriber Identity Module* (SIM). The network (in this example GSM) will assign a *Temporarily Mobile Subscriber Identity* (TMSI) number for addressing purposes. TMSIs are volatile and therefore reduce the risk of tracking individual subscribers. The more often a network changes the TMSI, the harder it is to passively track a specific user. Regardless, the network needs to know where its subscribers are at any given time to be able to communicate with them, e.g. forward incoming calls. In order to reduce position updates (saves network traffic and battery power on the mobile phone), updates are only performed when a phone moves from one group of cells (*Location*) to another, i.e. not on every individual cell. In case of an incoming message, the phone is *paged* in all cells of a *Location Area* (LA) and then assigned a specific logical channel of a cell. Based on the network's generation, this is either a frequency and a time slot (2G GSM) or an encoding scheme (3G UMTS).

To help the phone keep track of nearby cells, the network advertises them to the phone. Therefore, the scan overhead is reduced compared to full scans, saving time and battery. The phone maintains a short *neighbor list* based on signal strength and reports them back to the network on request. This data is the primary decision source for *handovers*, when the phone needs to change to another cell during an active call.

In GSM, a cell is uniquely identified by the mobile country code (MCC), network code (MNC), location area code (LAC) and the cell ID (CI). The neighbor list typically includes additional per cell attributes like the frequency (ARFCN) and channel quality metrics. Given that UMTS networks are organized differently, LAC and CI are replaced by PSC (primary scrambling code) and CPI (Cell Parameter ID). For the sake of simplicity, we will call any tuple that uniquely identifies a network cell a *Global Cell ID* or *Cell ID* for short.

IMSI Catchers blend into the mobile network operator's infrastructure impersonating a valid cell tower and therefore attracting nearby phones to register to it. Two main operating modes can be distinguished.

Identification Mode.

As a phone is lured into the fake cell, the worldwide unique identifiers such as IMSI and IMEI are retrieved and the phone is sent back to its original network via denying its original *Location Update Request* with an *Location Update Reject*-Message. This procedure typically takes less then two seconds, whereas attracting the phone can take minutes. No other information besides the identification numbers is retrieved.

A law enforcement agency can then apply for a warrant[1] and access the call- and meta information of a subject via the mobile network operator. This considerably saves the agency working hours, as no one has to operate the IMSI Catcher over the whole period of observation and follow the subject in its every move.

Other attackers can use this mode for user tracking purposes or to lookup the exact phone model based on the IMEI to better tailor future attacks.

Camping Mode.

The phone is held in the cell of the IMSI Catcher and content data is collected. Traffic is forwarded to the genuine network so that the victim stays unaware of the situation.

IMSI Catcher users that do not have time for for a warrant or can't acquire a warrant (e.g. because they operate outside the law) use this method. It will also gain importance as A5/3 and A5/4 are introduced into GSM networks, making passive snooping attacks on the broken A5/1 and A5/2 ciphers useless. In UMTS networks, phones are additionally downgraded to GSM and its less secure ciphers.

4. IMSI CATCHER ARTIFACTS AND DETECTABILITY

An IMSI Catcher has many detail problems to overcome; the respective solutions will typically introduce irregularities in the network layer that leave hints for an educated observer. Due to the secret nature of the operation of these devices, not much information is available. Nevertheless, we generated the list below based on the material available and our own research. Some of the traits can be mitigated but most are of structural nature. However, not every IMSI Catcher will produce all of the artifacts described below.

4.1 Choosing a Frequency

To increase signal quality, avoid radio interference, and thus trigger the mobile provider's own radio quality monitoring system, an attacker has to use an unused frequency

[1]May vary depending on the legislative system; in the U.S. also called *pen trap*

(i.e. ARFCN, Absolute Radio Frequency Channel Number) for its IMSI Catcher. A relatively safe choice for a frequency are unallocated radio channels (e.g. *guard channels* between different operators or reserved channels for testing). However, it is less likely to lure a mobile phone onto this channel, as the phone (MS) will preferably only look on the advertised neighbor frequencies. Another method is to use an advertised frequency that is actually not being used or is not receivable in the specific geographical area under attack.
Detectability: Off-band frequency usage can be detected using a current frequency band plan as assigned by the local authorities. Radio regulatory bodies and frequency plans are available for almost all countries.

4.2 Choosing a Cell ID

Typically, an attacker will introduce a new cell ID (preferable including a new LAC) previously unused in the specific geographical region for two reasons: First, to not provoke an accidental protocol mismatch when the MS should receive the corresponding genuine BS by accident. Secondly, to provoke a *Location Update Request*[2] from the phone to be able to lure it in the fake cell.
Detectability: Our data shows, that cell IDs are very static. Many mobile operating systems use them together with Cell ID databases to coarsely estimate the phone's location where either GPS is unavailable, rough estimations are detailed enough, or to aid the GPS receiver during initialization. Using such a database and correlating its information with the real geographic location could reveal unusual cell IDs and frequency usage in a specific area.

4.3 Base Station Capabilities Fingerprinting

Each beacon signal of a base station is accompanied by a list of supported features (e.g. packet radio services such as GPRS or EDGE). If the attacker does not copy the capabilities of the original network precisely, the simulated cell will not provide all services like the original network. For example GPRS and EDGE are services that need very complex emulation layers as they use a different modulation but share time slots with the rest of GSM. We do not expect many IMSI Catchers to support these protocols.
Detectability: A MS should denote such capabilities in the above Cell ID database (or a local one) and use them to find suspicious base stations not matching their previously known capabilities. Cell capabilities change very rarely, and if so, the network operator usually upgrades to new systems (e.g. GPRS to EDGE, HSDPA to HSUPA), but not vice versa.

4.4 Network Parameter Fingerprinting

Another information conveyed by the beacon signals to the mobile station are basic network parameters about the organization of the mobile network such as time slot organization, threshold values and timeout values. While they can differ from base station to base station, our research has shown that most of them tend to be uniform across a given network operator but vary between different operators. A IMSI Catcher operator might not always copy all of these parameters as they are not operationally important for an attack. Detection possible as described above (Section 4.3).

[2]A low *T3212 Periodic Location Update Timer* is another technique, but the smallest possible value is 6 minutes.

4.5 Forcing a MS to Register

Despite providing the better signal and simply waiting for a victim to voluntarily switch cells, an attacker can actively step in. An easy way to force a victim|s device to disconnect from the original network and register to a new (possible) fraudulent base station (as provided by the IMSI Catcher) is an RF jammer. After a fruitless scan of the advertised neighbor frequencies the phone eventually falls back to a full scan, therefore giving the IMSI Catcher the opportunity to attract the phone.

Several companies [6, 16] offer systems for targeted jamming of a specific phone.
Detectability: Jamming can be detected by a MS by watching channel noise levels (e.g. from the neighbor list).

4.6 Handling UMTS Clients

One possible way is to downgrade an UMTS capable MS to the less secure GSM network by rendering UMTS channels useless with an RF jammer (as above). Meyer and Wetzel [25] presented another way: a MITM attack for UMTS networks which facilitates its GSM compatibility layer. This layer is present in most deployed UMTS networks, as they use GSM for backward compatibility and to increase the coverage. Additionally, some companies [6, 16] claim, their equipment can transfer single targets from UMTS to GSM.
Detectability: Jamming can be detected as described above. A cell database can be used to spot unclaimed GSM usage where UMTS should be typically available.

4.7 Encryption

Older IMSI Catchers are likely to disable encryption (set cipher mode A5/0) in order to ease monitoring. However, current state-of-the-art attacks on GSM A5/1 and A5/2 cipher allow for a timely decryption and key recovery. Weaknesses found in the A5/2 cipher [9] have lead to its abolition by the GSM Association in 2006 [20]. However, the stronger variant A5/1 is also prone to precomputation attacks using rainbow tables. These are publicly available [5] and allow computers with a 2 TB hard disc and 2 GB RAM to recover the key in about two minutes [24]. While this makes completely passive eavesdropping on phone calls possible, phones can easily *get lost* by handing over to another cell (see next section). Furthermore, the newly introduced and currently rolled out [1] A5/3 and A5/4 ciphers (backported from UMTS) will force attackers back to active interception with IMSI Catchers to downgrade the encryption used. Known attacks on A5/3 are not yet feasible [8, 11, 23].
Detectability: The absence of a cipher alone is not a sufficient indicator: encryption might be unavailable in foreign roaming networks. However, once a phone had an encrypted session with a particular network and particular SIM card, it should assume that a sudden absence of any encryption is an alarming signal.

4.8 Cell Imprisonment

Once an attacker *caught* a phone, she/he will try to lock it in so it does not switch to another active cell. Therefore, it will either transmit an empty neighbor list to the phone or a list with solely unavailable neighbors. The base station can also manipulate the *receive gain* value [10]. This value is added to the actually measured signal levels by the MS to prefer a specific cell over another (hysteresis).

Detectability: A mobile station monitoring its neighbor list (e.g. together with a geographical database) is able to find such suspicious modifications.

4.9 Traffic Forwarding

The attacker needs to forward the calls, data and SMS to the public telephone system. There are multiple ways to achieve this. The simplest solution is to use another SIM card and a MS to relay calls into the mobile network. However, from the networks point of view these calls will be made under another identity. The attacker will most likely disable caller ID presentation to not immediately alarm the recipient. In this setup, the IMSI Catcher will not be able to handle any incoming calls for the surveyed station or any SMS.

Another setup could route these calls directly into a SS7 phone exchange network. Telecom operators usually trust their wholesale- and exchange partners with provider grade connections to set legitimate caller IDs. An attacker with access to such an interface could also spoof caller ID for outgoing phone calls and text messages. However, it is unlikely that the attacker can also manipulate the routing of incoming calls.

A third setup option (a full MITM attack) could facilitate a more advanced GSM frame relaying setup where data is handed over to the original network as if it where send by the victims phone.

Detectability: The first setup is detectable by making test calls and independently checking the caller ID (e.g. using an automated system).

4.10 Usage Pattern

IMSI Catcher in *identification mode* are operated for rather short periods of time to locate and verify an unknown phone such as prepaid phones or phones in an particular area. For tracking purposes and for eavesdropping the fake cell is active for the whole duration of the surveillance. Both operating times are considerably lower, than the average lifetime of a genuine cell.

Detectability: Cells that suddenly appear (with good signal quality) for a short period of time and cease to exists afterwards.

5. CATCHING AN IMSI CATCHER

Simple, cheap, and easily deployable *IMSI Catcher Catchers* (ICC) either need to run directly on a user's mobile phone or on affordable hardware (e.g. stationary device). While both concepts can be used to document IMSI Catcher use in a specific area, the former is also able to warn its user directly. In this section we describe both concepts, before we present our implementation in Section 6.

As Table 1 summarizes, the main detection method consists of a cell ID database. Commercial as well as free database projects exist. Most of them provide an online interface to their data. However, they neither guarantee to be complete nor correct, partly due to their croudsourcing nature. Also, they lack additional attributes needed for fingerprinting cell capabilities. Therefore, a IMSI Catcher Catcher (regardless if it is a mobile app or a dedicated stationary device) needs to be able to collect and maintain its own database regardless of any external databases (even when it is initially fed from another source). Furthermore, a mobile app can not assume online access is possible while being under attack.

Both types constantly collect all the data available about nearby cells. The mobile solution facilitates the almost ubiquitously built-in GPS receiver available in smart phones to correlate the data with its location. Therefore, from the phone's perspective the network topography is revealed similarly to *explorable maps* known from computer games, where the user only sees the areas of the map which he visited before (*Fog of War*). Visiting an already known area allows comparison of the current results with the stored data.

Additional tests include monitoring the noise levels of channels (RF jammer detection), network- and cell capabilities (e.g. cipher and GPRS availability), and sanity checks of network parameters (e.g. empty neighbor list might indicate a cell imprisoned phone). A caller ID test is implementable using an automated query system. However, regular calls to that system might result in non-negligible costs and have to be cryptographically authenticated.

The mobile app user (mICC) interface can be simplified to a user friendly four stage indicator:

Green No indicators of an IMSI Catcher attack found. Previously collected data matches the current network topography and all other tests completed negative.

Yellow Some indicators or tests show anomalies. However, these hints are not sufficient to postulate an IMSI Catcher attack. The user should avoid critical details in calls.

Red Indicators strongly suggest an IMSI Catcher attack or some other major network anomaly.

Grey Not enough data available (e.g. the user is in a previously unknown area).

An application with more intrusive access to the baseband might limit the phone's use to trusted cells only.

In contrast, a dedicated stationary IMSI Catcher Catcher (sICC) placed at a favorable position with a good antenna might receive a far greater radio cell neighborhood and allow to monitor a greater area non-stop (Figure 7). This is of great advantage when searching for a potentially transient event like the rather rare and short usage of an IMSI Catcher. Multiple devices can form a sensor network monitoring e.g. a whole city. As they don't move around, a GPS receiver is unnecessary. Most tests compare the collected data with the stations own history.

6. IMPLEMENTATION

Implementation poses some additional challenges: Only very limited baseband information is available to high level applications. In mobile operating systems, low level access is prohibited. System- and root applications can have access but are then limited to a very specific phone model (or chipset). This requires a *rooted* or *jail broke* phone. Additionally, only information is available that the chipset manufacturer has chosen to be disclosed. This also applies to commercial or industrial GSM/UMTS modules.

Among other baseband information, the neighbor cell list is an infamous example. Device support varies vastly, even for products of the same manufacturer. There is no identifiable pattern between low-end and high-end or older and newer products. Baseband information used to be called

Table 1: IMSI Catcher detection matrix

IMSI Catcher Artifact	Detection Method	Android API	iOS API‡	Telit [37]
Unusual Cell ID	Cell database	serving cell & neighbors†	serving cell only	yes
Unusual cell location		yes	yes	no
Unusual frequency usage		no	no	yes, ARFCN
Short living cells		yes	limited	yes
Unusual cell capabilities		serving cell & neighbors†	indirect	scan, neighbor
Guard channel usage	Band plan	no	no	yes
Network parameters	Network fingerprinting	no	no	limited (GPRS only)
RF jamming	Watching noise levels	limited	no	yes
Disabled cipher	Read cipher indicator	expected in future API [4]	no	no
Neighbor list manipulation	Cell DB & sanity check	limited†	no	limited
Receive gain	sanity check	no	no	no
Missing caller ID, SMS	Periodic test calls	yes	yes	yes

† Neighbor cells available via standard API, but not implemented in all phones.

‡ Only via iOS private API. See Section 6.2 on reasons why iOS is not considered in this paper.

engineering-, *field test-*, or *network monitor* functionality for a long time. However, a few years ago, access to information such as the *serving cell* or *neighbor cell list* became popular for (coarse) locating devices in combination with a geolocation cell ID database, where GPS is not available, a loose estimation is detailed enough, or to simply aid the GPS during initialization. Therefore, recent smart phone operating systems provide a direct or indirect API interface to this information - even when it is unreliable in some cases.

When available, the next challenge is just around the corner: A MS is not required to keep a list longer than six nearby cells. Thus, the neighbor list provides only a very limited geographical view into the nearby network structure of the currently selected operator, despite some potentially more receivable cells. This is especially true in very dense networks such as in urban centers.

To extend the view and collect more data than the neighbor list length, a MS could be switched to use just a specific network band, such as 900 or 1800 Mhz GSM band or the 2100 Mhz UMTS band (many older phones and some data modules allow for this). Collecting disjunctive neighbor cell information for all bands separately extends the view on the network. Additionally, a device with a foreign SIM might be able to register at multiple (roaming) networks to investigate each one separately. However, both techniques interfere with the normal operation of a hand set. A mobile device constantly performing these kinds of investigations is not able to provide services for the end user in commonly expected quality. It would require a dedicated device for such measurements.

6.1 GSM Modems and Modules

For the dedicated stationary type of the IMSI Catcher Catcher (sICC) we tested several USB modems from ZTE, Nokia, and Huawei as well as MiniPCI modems from Qualcomm, none of which supported neighbor cell listing. Nokia and Huawei seem to support it on older devices, but dropped support on more recent ones.

Additionally, we started to test industrial modems such as devices from Telit. Among others, the Telit GT864 allow network registration and neighbor list scanning even without an inserted SIM card, allowing to scan each network in a region on each frequency band separately (see above). This provides a much greater view on the network structure than a simple mobile phone can provide.

On top of it, many Telit modems implement a cell beacon monitoring mode [37] that can be easily facilitated into a frequency band sweep cell beacon scan. Thus, allowing a complete view over the receivable network cells by frequency including their ID, some capabilities, signal, and noise levels. The latter also allows a simple jamming detection.

Our Implementation.

Our dedicated stationary setup (Figure 1) consists of a Telit GT864 [36] and a Raspberry Pi embedded Linux computer. Internet up-link (to collect the captured data) is either provided by an Ethernet network, power LAN, via WIFI (USB-Dongle), or an UMTS modem. Data is collected locally in an sqlite3 database and periodically uploaded to a central server. The whole setup including mounting material costs less than €200. As the device is able to perform full frequency scans for all providers without the limitation of length-limited neighbor cell lists, we placed these devices on rooftops to extend their range.

As of August 2014, the network consists of four devices, the first one went online in July 2013. Our sICC is able to sweep through the whole 900 and 1800 Mhz GSM and EGSM bands within five to seven minutes. Besides the Cell ID, its

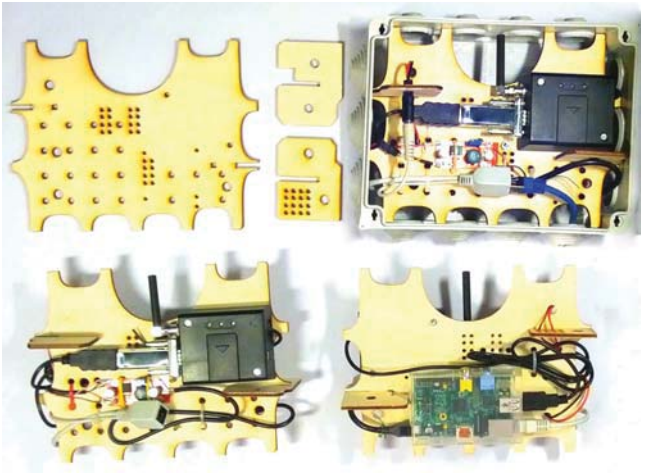


Figure 1: Construction of the dedicated stationary unit, using a laser-cut carrier (front and back)

main and auxiliary ARFCNs, it also records its receive levels and bit error rates as well as several GPRS configuration parameters (t3168 and t3192 timeouts, routing area codes, GPRS paging modes, etc).

6.2 iOS

iOS neither exposes high-level nor low-level baseband information (e.g. cell info) to applications through the official and public API. Methods such as `_CTServerConnectionCellMonitorGetCellInfo()` are available through a *private API*, whose documentation has leaked to the web. A field test App is available since iOS 5.1 by dialing `*3001#12345#*`. While the OS does not prevent the private API usage, it has been reported to be an immediate exclusion reason from the Apple App store. Applications using this API are only available to phones with a developer license or jail-broken phones and are therefore not of great use for a broader public.

Without a chance for widespread usage, we excluded iOS phones from further consideration.

6.3 Android OS

Android is a little more generous in providing access to baseband information. The `TelephonyManager` defines access to the important, but not all values on the wish list for the IMSI Catcher Catcher. Some values, such as the *cipher indication*, have been requested years ago and only recently got assigned for implementation [4].

The neighbor cell list problem described above also continues in the Android universe: The API defines the `TelephonyManager.getNeighboringCellInfo()` method. However, not even the long-time lead device *Google Galaxy Nexus* supports this method. Other devices only return meaningful values for this call for GSM type of networks, but not UMTS. It is not always clear if the underlying chipset does not provide this information or if the high level API lacks implementation by the phone manufacturer. A survey by the authors of the G-NetTrack application [17] reveals that this functionality is supported by less than half of the tested devices. Most devices report data only for the current serving cell. Recent devices have higher chances of implementing this method, most notably the *Google LG Nexus 4* and *Google LG Nexus 5*.

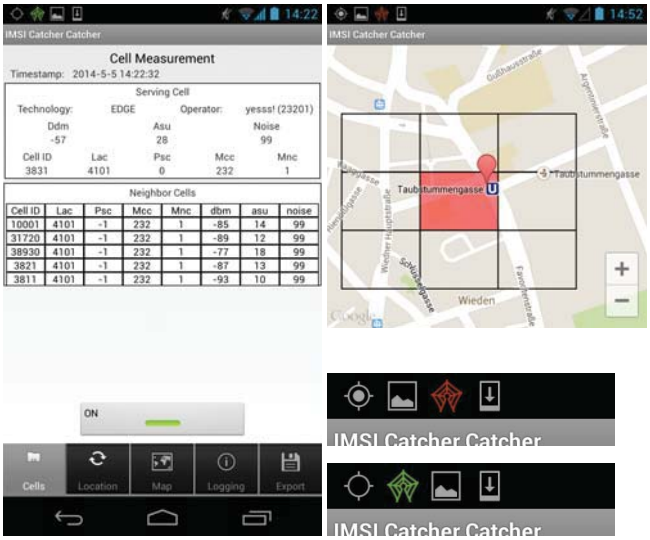


Figure 2: Screenshots of the mICC

In contrast, Samsung Galaxy S2 and S3 expose many parameters unavailable through the standard API (such as the cipher mode [4]) via a Service Mode Application [14]. Some HTC devices offer similar hidden *Field Test Applications* [33]. This applications run under elevated privileges and often directly communicate with the baseband chipset via an operating system level device. Copying their interface will limit the application use to a rooted phone of a very specific model.

The absence of a neighbor list feature does not make a mobile IMSI Catcher Catcher (mICC) application impossible, but much less effective. This especially effects the speed of the network structure learning phase and some sanity checks on the network structure (e.g. cell lock-in by not having any neighbors). Another value offered by the API but not implemented in all phones is the noise level.

Our Implementation.

In favor of keeping our implementation [2] root permission free, we intentionally renounce the use of low-level information. While this provides less details it enlarges the potential user base.

A background service collects GPS position and cell related data (serving cell, neighbor cell, supported packet data modes). Measurements are triggered by the `PhoneStateListener.onCellInfoChanged()` - Callback and a regular 10-second timer (whichever comes first). This way, brief redirection to and from a cell (Section 3, Identification Mode) can be detected. For the sake of simplicity we group measurements in rectangular geographical tiles of about 150 × 100 meters and store them in an sqlite3 database. Some tiles might be in the learning phase while others are used for evaluation at the same time. We consider a tile fit for evaluation if the user collected cell data in this cell and all of its 8-connected tile neighborhood. Otherwise, nearby cells might easily create false alarms. A cell is considered valid for a given tile, if it was received as serving- or neighbor cell in one of the 9 tiles.

The app also runs in the background and displays the current evaluation result in the notification bar, so that it is visible in the system dialer and phone application.

7. RESULTS AND DISCUSSION

The evaluations goal is to answer two main questions: (1) Are the two IMSI Catcher Catcher able to detect the presence of an IMSI Catcher? (2) Are IMSI Catchers used in our vicinity?

We evaluated both systems with lab tests as well as field tests. For our lab tests we used an USRP1 based IMSI Catcher running OpenBTS 2.6 in identification mode. Therefore, we patched OpenBTS to download the IMEI and IMSI of any phone and then reject the Location Update request - pushing the phone back into the genuine network based on [32]. Because of their very brief interaction with the phone, such IMSI Catchers are particularly hard to detect. Experiments were concluded in an controlled environment to not interfere with outside phones.

7.1 Stationary IMSI Catcher Catcher

In the lab experiment, the sICC was able to detect the new fake cell based on its cell id, parameters and capabilities.

For the field test, our first sICC was installed on a rooftop

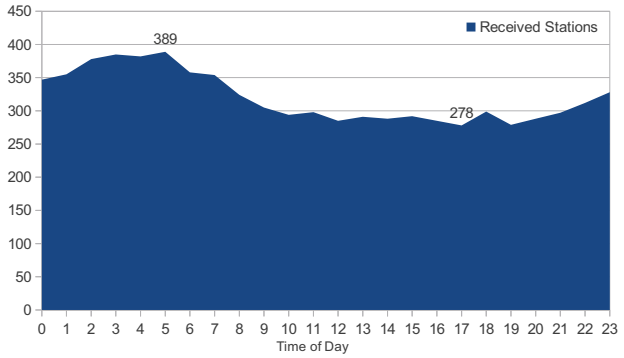


Figure 3: Maximum number of unique distinct cells received throughout the day (sICC)

in Viennese city center in July 2013. Three additional stations have been installed in the first months of 2014. We collected over 40 million datasets. The range of some installations is remarkable: Under rare conditions (*Inversions*) we receive single stations up to 90 km away. Radio conditions vary among the day and so does the number of received cells (Figure 3). A map based Google's geolocation database is shown in Figure 7. This external database is only used for visualization purposes and is not required for detection.

Regarding fingerprinting of cell parameters, we found many useful parameters[3]. In our test set of Austrian A1, T-Mobile, Orange/H3G, and Slovak O2 Telefonica network they all have the same value on all cells within a network, but distinct values between operators. Other values[4] displayed two distinct values within the Orange/H3G network.

CellIDs are very stable regarding their used ARFCN. However, on very received cells, one ARFCN can seem to have alternating different CellIDs. This can happen in situations, where the receiver sits in between two distant cells that are both using the same channel.

As Figure 4 shows, most cells remained static throughout the entire collection time. We attribute the bulk of very short-living cells to the following two effects: First, exceptional but transient weather and RF conditions that allowed the reception of cells very far away - often from foreign networks (Slovakia, Hungary, Czech Republic). We attribute this to *tropospheric scattering* and *ducting* caused by inversions [19, p.44]. These cell receptions are typically in the GSM 900 band and recorded as having very low signal levels and high bit error rates.

Second, we noticed a bigger cell reorganization at one of the operators (*A1 Telekom Austria AG*) in the night from November 16th 2013. During a period of several hours, many cells appeared for only a brief period of time. We have not yet received any explanation from the operator. Also in November 2013, Orange/H3G received previously unassigned frequencies in the GSM 1800 band.

We found two additional irregularities in our collected data: (1) Some cells seemed to operate outside the official assigned frequency ranges. A request at the Austrian Regulatory Authority for Broadcasting and Telecommunication (RTR) revealed an error on their side in the published frequency band plan. This was later corrected [7]. (2) We

[3] `PBCCH` existence, `SPGC`, `PAT`, `t3168`, `drmax`, `ctrlAck`, `alpha` and `pcMeasCh`

[4] `NMO` and `bsCVmax`

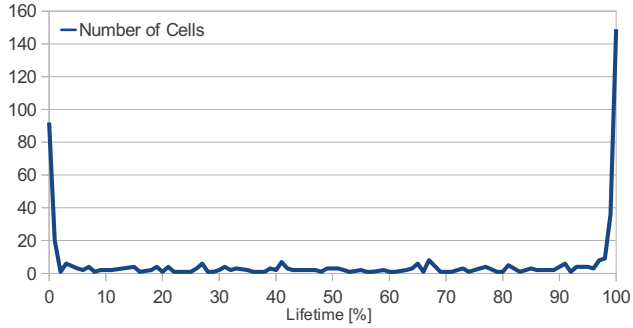


Figure 4: Cell ID lifetime throughout the experiment

received a cell with a valid looking Austrian MNC, LAC, and CI, but an unassigned network country code (NCC). We speculate that this could be either a transmission error or a base station in maintenance or test mode.

Under certain conditions it can make sense for an IMSI Catcher to emulate a foreign network to catch a roaming handset. However, in our case we are receiving different stations during nighttime over a span of multiple months. We therefore do not think these symptoms fits an IMSI Catcher and attribute them to natural effects (Section 9.2).

7.2 Mobile IMSI Catcher Catcher

For the prototype app we required at least 30 measurements and two re-entries into each map tile, before it finished the learning state. Additionally, the whole 8-neighborhood of the current tile must finish learning before it is considered for evaluation. The map view of our app supports the user in coloring tiles based on needed data. An always visible color coded icon in the notification bar indicates the warning level (Figure 2).

In our lab experiment, we were able to detect new and short living cells reliably, even when the *Location Update* was immediately rejected by our IMSI Catcher. Subtle differences exist in the implementation of the baseband to Android API interface. Some models report the new CellID and LAC for the ongoing but not completed cell change. Others only update the CellID immediately, while the LAC remains unchanged until the new base station accepts the *Location Update* request (e.g. Nexus 4).

For our biggest field test we chose a notoriously violent event in Vienna: a politically disputed ball taking place in the city center, and its counter-demonstrations. We anticipated that the authorities could use an IMSI Catcher to identify rowdies as suggested by media reports. We assembled a battery of three phones (Figure 5) for all three disjunctive GSM networks in Austria. We visited the demon-

Figure 5: Field test for all three GSM networks

stration route the day before and then attended the demonstration undercover. However, we could not find any indicators of an IMSI Catcher.

7.3 Limitations

Our geographical network topology correlation approach and the cell database in general assumes a rather static mobile network structure, as every change will be flagged as suspicious. In fact, network structure is very steady and this is actively utilized by mobile operating systems for coarse self-localization and commercial suppliers of geographical cell databases.

There are corner cases where the mobile IMSI Catcher Catcher needs refinement. One such case are tunnels and underground trains. In Vienna, the public metro enjoys an almost flawless GSM and UMTS coverage. However, without GPS reception these underground cells often get associated with the place of entrance into the underground structure, as the phone's GPS receiver needs some time to detect its failure.

Another problem are holes in the tile map. If a tile is entirely located within an inaccessible area (e.g. a large private property), the 8-connection neighborhood rule forces all nine cells to never advance from the learning state into the evaluation state. This could be mitigated by a hole filling algorithm (e.g. an interpolation). Additionally, setting appropriate warning thresholds needs extensive real world testing.

8. RELATED WORK

The osmocomBB Project [3] offers some IMSI Catcher indicators in their custom baseband firmware including cell fingerprinting and cipher indication. However, the project's target hardware platform are Texas Instruments' *Calypso* chipset based phones such as the (outdated) Motorola C123 or V171. This series of handsets appeared in 2005 and went out of production several years ago. Considering the fast production cycles and the non-disclosure policies in the mobile phone industry it is unlikely that such open source projects will develop similar custom firmware for recent phones any time soon.

Melette and Nohl, being aware of the latter, started investigating the possibilities to port at least a subset of this functionality to recent smart phone operating systems [24]. Problems include the limited access to baseband information. However, there has not been any activity on this project since January 2012. Another tool by Hummel and Neumann [21] works on a PC using an USB connection to a phone with an *Intel/Infineon X-Gold* baseband processor (Samsung Galaxy S2 and S3, but not S4).

Vallina-Rodriguez et al. [38] also faced the problem of acquiring internal baseband values and decided to require root privileges.

Unlike previous works, our approach works by recording the geographical topography of a mobile network and is therefore able to detect structural changes that an active IMSI Catcher will cause. It facilitates the almost ubiquitously built-in Global Positioning System (GPS) receiver in smart phones. By using only standard API without any special permissions it ensures compatibility with as many phones as possible and is fit for public use. Some similar approaches for Android are also employed by the AIMSICD-Project [15].

9. FUTURE WORK

We are currently experimenting with a new RTL2832U based stationary IMSI Catcher Catcher prototype. The RTL2832U [30] is used in many DVB-T/DAB television and radio receiver USB sticks in the US$25 range. The chipset offers a way to bypass the DVB decoder and directly download 8-bit I/Q-samples with typically 2.8 MS/s turning it into a *Software Defined Radio* (SDR). Different tuner types exist, where the Elonics E4000 is the only one covering all major mobile phone bands by ranging up to 2200 Mhz. However, their extreme low price is to blame for the bad quality of many secondary components used. The oscillator accuracy can be as low as 50 ppm, leading to huge frequency offsets and shifts during operation. 30 kHz up or down is not a big deal, when receiving a multi-Mhz broad DVB-T signal. However, on a 200 kHz GSM signal they are very disruptive and need extra compensation.

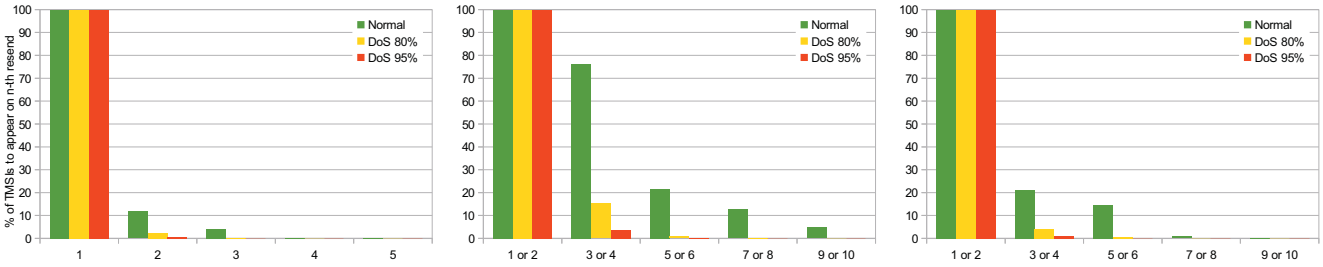
Directly decoding the broadcast and control channels (i.e. BCCH and CCCH) gives much more insight and material for fingerprinting base stations (e.g. more details about the organization of logical channels, broadcast traffic)[5]. It does also allow for detecting other types of attacks, such as the *Let me answer that for you* type of denial of service attack by Golde at al. [18]. In general this attack exploits a race condition, in which a fraudulent array of phones with a custom firmware answer a paging request before the genuine phone does. The following cipher handshake will almost certainly fail, leaving the GSM state machine no other option than to drop the call. As paging is broadcast over the whole Location Area (LA) this potentially affect a huge number of subscribers even when deployed only in one spot. A single LA can cover large portions of a multi-million inhabitants city [18, Fig. 8].

9.1 Exposing Large Scale Denial of Service Attacks

Based on paging statistic of over 470,000 paging requests of all three Austrian GSM networks we simulated how the distribution of paging broadcasts re-transmits will change in a network under attack based on the retry policies of the individual networks. A certain number of mobile stations does not answer on the first paging request (e.g. caused by a dead spot or interference) and has to be paged again. Some networks switch over from TMSI to paging by IMSI as a last resort. For our statistics we have to focus on TMSI paging, as there is no easy way to de-annonymize a large number of mobile stations at once. The distortions should be negligible for our purpose. We further conservatively assumed all paging requests within a 10 second window to belong to the original request. Only in very few cases (e.g. receiving many SMS messages in a brief period of time) this will not hold true.

Each paging request has a certain probability to not be answered by the target station on the first try and is therefore repeated. Based on the individual retry policy of each network, this produce a specific distribution on how many paging requests are tried a second, third, forth,... time. In our simulation we assumed a much less skillful attacker than in [18] with only 80% success rate and another one with 95%

[5]This data is mostly privacy neutral, as it contains public system information about the network and pseudonymized paging requests.



(a) A1-mobilkom (b) T-Mobile (c) Orange/H3G

Figure 6: Number of TMSIs to (re)appear in the n-th paging resend within a 10 second window.

success rate. In both cases, the distribution of paging retries is severely distorted. Interestingly enough, some networks (i.e. T-Mobile and Orange) almost always page in pairs with just a few hundred milliseconds in between, in which case we grouped these requests.

Figure 6 displays how the retry-statistics is distorted from the normal empirical data (green) by applying a DOS attack with 80% respectively 95% success rate. Watching this relation can reveal such an attack against a whole Location Area, however it will not detect attacks against single phones (once the TMSI - IMSI pseudonmization is broken).

9.2 Inversions and Tropospheric Ducting

Based on laser ceilometer [12] data from the Austrian central institution for meteorology and geodynamics (ZAMG) we have found a slight correlation ($\phi = 0.21$) on reception of selected far off cells and border layers between 1000 and 2200 meters. This suggests that a better weather model might help us to understand the occasional excessive range of our stations. Eventually, this will allow us to clean up received data as these effects can produce similar short term reception patters to briefly operated IMSI Catchers.

10. CONCLUSION

IMSI Catchers – as man-in-the-middle eavesdropping devices for mobile networks – became cheap and relatively easily available. Even in UMTS 3G networks, GSM 2G security is still important, as these networks are closely linked together, and therefore the *weakest link* principle applies.

Our goal was to survey, implement, and evaluate *IMSI Catcher Catchers* (i.e. devices that detect *IMSI Catchers*). We therefore identified structural artifacts thanks to which IMSI Catchers can be detected. Some of these can be mitigated, but not evaded completely.

Our first implementation is based on a network of stationary measurement devices with cheap and easily acquirable hardware. Data is collected in a central database for long time observations and then analyzed. We collected over 40 million datasets in 10 months. The second one is based on the Android platform and uses only publicly available APIs. Thus, ensuring its operability in future versions and on as many devices as possible. Furthermore, it neither requires special permissions nor rooting (or jail-breaking) of the phone. Because of its simple color-based warning system it is suitable for daily use.

Both solutions are not dependent on any external databases, as they collect all needed information by themselves. With an OpenBTS based IMSI Catcher, we validated the described methods. Both of our IMSI Catcher Catchers were able to detect the attack reliably, even in *identification mode* where the phone is captured for less then two seconds. In the future, we like to extend our tests to commercial available products. Our long term observation of real mobile networks with our fixed measurement devices was inconclusive at the time of writing.

Our results indicate that the detection of this kind of attack became feasible with standard hardware. Additionally, we described how to detect additional attacks on mobile networks, such as an recently published DOS attack.

Both implementations [2] have been released under an open source license.

Acknowledgments

The research was partly funded by the COMET K1 program by the *Austrian Research Funding Agency* (FFG). We like to thank *Technikum Wien* and *u'smile* (a Josef Ressel Center) for kind hardware support. Additional support by the *Institute of Telecommunications* (E389) at *Vienna University of Technology*, the *Austrian central institution for meteorology and geodynamics* (ZAMG), and *Funkfeuer Wien*. Special thanks for the valuable feedback from various telco employees and to *Manuel Leithner*.

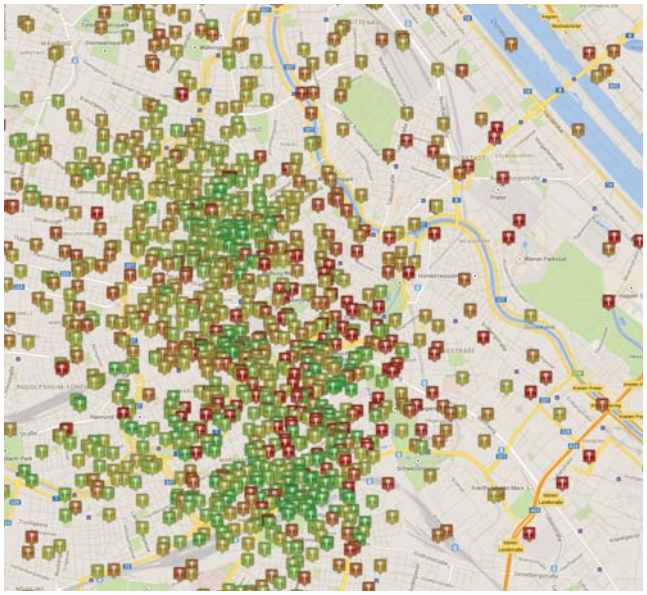


Figure 7: sICC: Color coded by signal strength of received cells. (Google Maps)

11. REFERENCES

[1] GSM security map. http://gsmmap.org/.

[2] IMSI Catcher Catcher source code. http://sourceforge.net/p/icc/.

[3] OsmocomBB open source GSM baseband software implementation. http://bb.osmocom.org.

[4] Android issue 5353: Ciphering indicator, 2009. https://code.google.com/p/android/issues/detail?id=5353, accessed July 14th 2013.

[5] A5/1 decryption rainbow tables. via Bittorent, 2010. opensource.srlabs.de/projects/a51-decrypt/files.

[6] Ability Computers and Software Industries Ltd. 3G Interception. Sales brochure. https://wikileaks.org/spyfiles/files/0/80_ABILITY-GSM_3G_Intercept.pdf, accessed Feb 25th 2014.

[7] Austrian Regulatory Authority for Broadcasting and Telecommunication RTR. Current utilization for GSM of the GSM 1800 frequency band. https://www.rtr.at/de/tk/1800MHzGSM.

[8] E. Biham, O. Dunkelman, and N. Keller. A related-key rectangle attack on the full KASUMI. In B. Roy, editor, *Advances in Cryptology - ASIACRYPT 2005*, volume 3788 of *Lecture Notes in Computer Science*, pages 443–461. Springer Berlin Heidelberg, 2005.

[9] M. Briceno, I. Goldberg, and D. Wagner. GSM voice-privacy algorithm A5/1, 1999. http://www.scard.org/gsm/, accessed July 17th 2013.

[10] Chris Paget aka Kristin Paget. Practical cellphone spying. In *DEFCON 19*, 2010.

[11] O. Dunkelman, N. Keller, and A. Shamir. A practical-time attack on the A5/3 cryptosystem used in third generation gsm telephony, 2010.

[12] S. Emeis, K. Schafer, and C. Munkel. Surface-based remote sensing of the mixing-layer height a review. *Meteorologische Zeitschrift*, 17(5):621–630, 2008.

[13] Ettus Research. Universal software radio peripheral. https://www.ettus.com/product.

[14] E:V:A et al. Galaxy S III - "secret codes" and hidden features, 2012. Pseudonymized online discussion forum, http://forum.xda-developers.com/showthread.php?t=1687249, accessed July 14th 2013.

[15] E:V:A et al. IMSI Catcher/Spy Detector, 2012-2014. Pseudonymized online discussion forum, http://forum.xda-developers.com/showthread.php?t=1422969, accessed September 7th 2014.

[16] Gamma Group. 3G-GSM Interctiopn & Target Location. Sales brochure. info.publicintelligence.net/Gamma-GSM.pdf, accessed Aug 27th 2013.

[17] G-NetTrack phone measurement capabilities. http://www.gyokovsolutions.com/survey/surveyresults.php, accessed July 15th 2013.

[18] N. Golde, K. Redon, and J.-P. Seifert. Let me answer that for you: Exploiting broadcast information in cellular networks. In *Proceedings of USENIX Security 2013*, pages 33–48. USENIX, 2013.

[19] A. W. Graham, N. C. Kirkman, and P. M. Paul. *Mobile Radio Network Design in the VHF and UHF Bands*. John Wiley & Sons Ltd, 2007.

[20] Prohibiting A5/2 in mobile stations and other clarifications regarding A5 algorithm support. http://www.3gpp.org/ftp/tsg_sa/TSG_SA/TSGS_37/Docs/SP-070671.zip.

[21] T. Hummel and L. Neumann. Xgoldscanner, 12 2013. https://opensource.srlabs.de/projects/mobile-network-assessment-tools/wiki/Xgoldscanner, accessed Feb 19th 2014.

[22] F. Joachim and B. Rainer. Method for identifying a mobile phone user or for eavesdropping on outgoing calls, 2000. Patent, Rohde & Schwarz, EP1051053.

[23] U. Kühn. Cryptanalysis of reduced-round MISTY. In *Advances in Cryptology – EUROCRYPT 2001*, pages 325–339. Springer Verlag, 2001.

[24] L. Malette. Catcher Catcher. opensource.srlabs.de/projects/catcher, accessed July 12th 2013.

[25] U. Meyer and S. Wetzel. A man-in-the-middle attack on UMTS. In *3rd ACM workshop on Wireless security*, pages 90–97, 2005.

[26] P. Muncaster. Chinese cops cuff 1,500 in fake base station spam raid. The Register, 26 Mar 2014. http://www.theregister.co.uk/2014/03/26/spam_text_china_clampdown_police/.

[27] K. Nohl. Breaking GSM phone privacy. Blackhat 2010.

[28] K. Nohl. Rooting SIM cards. Blackhat 2013.

[29] K. Nohl and S. Munaut. Wideband GSM sniffing. Chaos Communications Congress (27C3), 2010.

[30] osmocom Project. RTL-SDR - osmcomSDR. http://sdr.osmocom.org/trac/wiki/rtl-sdr, accessed March 5th 2014.

[31] S. K. Pell and C. Soghoian. Your secret stingray's no secret anymore: The vanishing government monopoly over cell phone surveillance and its impact on national security and consumer privacy. *Harvard Journal of Law and Technology*, 2014. http://ssrn.com/abstract=2437678.

[32] B. Postl. IMSI Catcher. Master's thesis, Technikum Wien, 2012.

[33] Richard's wireless blog. Hidden menus in android phone, 2009. http://rwireless.blogspot.co.at/2009_03_23_archive.html, accessed July 14th 2013.

[34] Rohde & Schwarz. Countering threats early on. www.idexuae.ae/ExhibitorLibrary/1328/Countering_threats_early_on_2.pdf, accessed July 14th 2013.

[35] M. Solnik and M. Blanchou. Cellular Exploitation on a Global Scale: The Rise and Fall of the Control Protocol. Blackhat 2014, Las Vegas.

[36] Telit Wireless Solutions. GT864-QUAD/PY - GSM/GPRS modules and terminals. http://www.telit.com/en/products/gsm-gprs.php?p_ac=show&p=3, accessed Feb 22th 2014.

[37] Telit Wireless Solutions. Easy Scan user guide, April 2013. http://www.telit.com/module/infopool/download.php?id=6004, accessed July 19th 2013.

[38] N. Vallina-Rodriguez, A. Auçinas, M. Almeida, Y. Grunenberger, K. Papagiannaki, and J. Crowcroft. RILAnalyzer: a Comprehensive 3G Monitor On Your Phone. In *Proceedings of the 2013 Internet Measurement Conference*, IMC '13, pages 257–264. ACM, October 2013.

[39] D. Wehrle. Open source IMSI-Catcher. Master's thesis, Albert-Ludwig-Universität Freiburg, 2009.

Advanced Wi-Fi Attacks Using Commodity Hardware

Mathy Vanhoef
iMinds-DistriNet
KU Leuven
Mathy.Vanhoef@cs.kuleuven.be

Frank Piessens
iMinds-DistriNet
KU Leuven
Frank.Piessens@cs.kuleuven.be

ABSTRACT

We show that low-layer attacks against Wi-Fi can be implemented using user-modifiable firmware. Hence cheap off-the-shelf Wi-Fi dongles can be used carry out advanced attacks. We demonstrate this by implementing five low-layer attacks using open source Atheros firmware. The first attack consists of unfair channel usage, giving the user a higher throughput while reducing that of others. The second attack defeats countermeasures designed to prevent unfair channel usage. The third attack performs continuous jamming, making the channel unusable for other devices. For the fourth attack we implemented a selective jammer, allowing one to jam specific frames already in the air. The fifth is a novel channel-based Man-in-the-Middle (MitM) attack, enabling reliable manipulation of encrypted traffic.

These low-layer attacks facilitate novel attacks against higher-layer protocols. To demonstrate this we show how our MitM attack facilitates attacks against the Temporal Key Integrity Protocol (TKIP) when used as a group cipher. Since a substantial number of networks still use TKIP as their group cipher, this shows that weaknesses in TKIP have a higher impact than previously thought.

1. INTRODUCTION

Wireless networks based on the 802.11 standard have had an enormous success, ranging from use in common households to large scale deployments in critical infrastructures. As new standards push the boundary of transmission speed and functionality, the capabilities of wireless chips have increased accordingly. This opens new possibilities where commodity devices can be used to implement state of the art attacks, previously thought only possible on expensive hardware such as Universal Software Radio Peripherals (USRPs). To demonstrate this we implement several low-layer attacks using off-the-shelf Wi-Fi dongles. In particular we modify the firmware of Atheros AR7010 and AR9271 chips. We conjecture that other devices, which also load user-modifiable firmware after power up, can execute similar attacks.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.
ACSAC '14, December 08–12, 2014, New Orleans, LA, USA.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-3005-3/14/12 ...$15.00.
http://dx.doi.org/10.1145/2664243.2664260 .

Our first modifications give the client an unfair share of the bandwidth. We use this to experimentally explore the behaviour of selfish stations. Based on this, to the best of our knowledge, we are the first to suggest that contending selfish stations will exploit the capture effect to increase throughput. This is the phenomenon where, in case of a collision, the frame with the strongest signal and lowest bitrate is received correctly [13]. Surprisingly, contending selfish stations end up lowering their bitrate to obtain a higher throughput. We also bypass systems designed to prevent selfish behaviour.

We continue by turning our dongle into a continuous and selective jammer. The former endlessly transmits noise, while the latter jams specific frames already in the air. Previously, selective jamming required either a costly USRP setup [7], or a WiFi adapter which allows modifications to the microcode handling medium access and de/encoding operations [4]. Though it shows selective jamming is possible, the URSP setup is non-trivial and costs more than $3600. Additionally, being able to change medium access algorithms in commodity devices is rare in practise. We show that these jamming attacks can be implemented using off-the-shelf USB dongles (costing only 15$) without requiring low-level microcode access. It is surprising this is possible by using only the default (and limited) API of the wireless chip (i.e., without modifying its medium access algorithms).

With as goal to enable (or facilitate) attacks on higher-layer protocols, we also present a channel-based MitM attack against any type of encrypted network. Our goal is not to decrypt traffic, but to be able to reliably intercept and manipulate it. Normally obtaining a MitM position in an encrypted network is not trivial, since the station and Access Point (AP) verify each other's MAC address. Therefore an attacker cannot set up a rogue access point with a different MAC address to subsequently intercept and forward all traffic. We demonstrate that cloning the AP on another channel bypasses the MAC address checks. Customizable firmware makes the implementation easier and more efficient, in particular when targeting multiple stations simultaneously.

Finally we use the MitM position to attack TKIP when used as a group cipher, and show how to decrease the execution time of the attack by targeting multiple stations simultaneously. Since most routers by default use TKIP as their group cipher for backwards compatibility, a significant amount of networks are vulnerable to our attack.

Our selective jammer, channel MitM, and TKIP attacks are available for download [1]. The unfair channel usage and continuous jammer can disrupt network activity without usable countermeasures being available. Therefore we do not

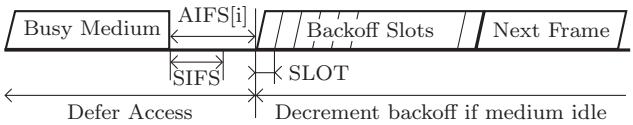


Figure 1: EDCA Timing Relations.

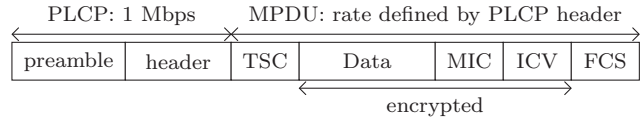


Figure 2: Simplified format of 802.11b TKIP frames.

release them. To summarize, our main contributions are:

- We study (contending) selfish stations using off-the-shelf Wi-Fi dongles, taking into account capture affect, and bypass systems designed to detect selfish users.
- We show how continuous and selective jamming can be implemented using off-the-shelf Wi-Fi dongles.
- We present a channel-based Man-in-the-Middle attack against encrypted networks, enabling reliable interception and manipulation of all encrypted traffic.
- We attack TKIP when used as a group cipher, and additionally devise a technique to decrease the execution time by targeting multiple stations simultaneously.

The remainder of this paper is organized as follows. Section 2 explains the relevant aspects of the 802.11 standard. In Section 3 we analyse, implement, and measure the impact of selfish stations. Section 4 shows how continuous and selective jamming can be implemented on commodity devices. A novel channel-based Man-in-the-Middle attack is presented in Section 5, and in Section 6 we use this to attack TKIP when used as a group cipher. Finally, we summarise related work in Section 7 and conclude in Section 8.

2. THE 802.11 STANDARD

In this section we present the basics of the 802.11 MAC and physical (PHY) layer [11], explain the TKIP encryption protocol, and introduce the `ath9k_htc` firmware.

2.1 Medium Access Control (MAC)

The MAC service is responsible for exchanging MAC Protocol Data Units (MPDUs) using carrier sense multiple access with collision avoidance (CSMA/CA). A station wishing to transmit first monitors the medium for a given Inter Frame Space (IFS), using a carrier sense mechanism. If the medium is busy the station defers its transmission using a random backoff procedure. The original standard offered the Distributed Coordination Function (DCF) to control this process. The 802.11e amendment extends DCF with Quality of Service (QoS) enhancements, resulting in enhanced distributed channel access (EDCA). With EDCA four different Access Classes (AC) are defined, allowing the prioritization of traffic. For example, the access class of voice traffic has a higher priority than that of background traffic. We use the terms QoS channel and priority as synonyms of AC.

The time a station must wait before it can transmit depends on several parameters and inter frame spaces (see Figure 1). The shortest is the Short Interframe Space (*SIFS*) and is the time between successfully receiving a frame and sending an acknowledgement (ACK). Other frames must wait longer than *SIFS*, giving ACKs the highest priority. When a station wants to transmit a data frame of access class AC it first monitors the medium for a period equal to

$$AIFS[AC] = SIFS + AIFSN[AC] \cdot SLOT \qquad (1)$$

where $SLOT$ is the duration of one slot interval, and with $AIFSN$ dependent on the access class. If the medium is idle during this time the station can transmit immediately. In case the medium was busy the contention window $CW[AC]$ is set to $CW_{min}[AC]$ and the random backoff procedure is initiated. This procedure initializes the backoff counter to a value uniformly distributed over the interval $[0, CW[AC]]$. Once the medium has been idle for $AIFS[AC]$, the backoff counter is decremented for each slot where the medium is idle. When the counter is zero the station transmits the frame. Since AC is clear from context it will no longer be explicitly included. The values assigned to $AIFSN$, CW_{min}, and CW_{max} determine the priority of an access class, and are advertised by the AP in the EDCA parameter set element in beacons and probe responses. Beacon frames are periodically transmitted to advertise the presence of a network, while probe requests and responses are used to actively seek networks. The value of $SIFS$ and $SLOT$ depend on the physical layer being used (see Section 2.2).

If the Frame Check Sequence (FCS) at the receiver is correct it will wait *SIFS* time and send an ACK. If the sender receives an ACK it resets CW to CW_{min} and enters a post-backoff stage by setting the backoff counter to a value uniformly distributed over the interval $[0, CW]$. When the FCS is wrong the frame is discarded. When the sender receives no ACK it retransmits the frame by updating CW to $2 \cdot CW + 1$ and repeating the backoff procedure. A frame is retransmitted at most 7 times, and CW is limited by CW_{max}.

Assuming all stations cooperate, analytical models of DCF show that it provides fairness [5]. Hence identical access classes in EDCA also provide this fairness. However, in practice stations can act selfishly and increase their throughput at the cost of others [25, 22]. In Sect. 3 we will study in detail how selfish stations can increase their throughput.

2.2 Physical Layer (PHY)

Wi-Fi can operate in the 2.4 and 5 GHz bands. A channel number defines the center frequency on which the radio operates. Senders can choose between multiple transmission rates: low rates are used for bad connections, and high for good connections. For backwards compatibility beacons and probes are generally sent using the lowest rate of 1 Mbps.

When an MPDU is transmitted a Physical Layer Convergence Protocol (PLCP) preamble and header is added (see Figure 2). The preamble allows the receiver to synchronize to the transmission, and the header defines the modulation used for the MPDU. We focus on 802.11g long preamble mode, where (slightly simplified) the overhead of the PLCP is 23 μs, $SIFS$ is 10 μs, and $SLOT$ is 9 μs [11, §18.3].

An important property of 802.11 radios is their susceptibility to the capture effect, the phenomenon where the strongest frame is received correctly in case of a collision. This goes against the common assumption that both frames are lost. More precisely, a collision results in one of three cases [13]: (1) both frames are lost; (2) if the stronger

frame is received first it will be decoded correctly; or (3) if the stronger frame is received second, yet within the PLCP preamble of the first one, the second frame will be decoded correctly. The capture effect has been observed on different chipsets [13, 8], occurs more frequently at low bitrates [14], and favours nearby stations [13]. In Section 3 we show that selfish stations can abuse it to contend with other stations.

2.3 Temporal Key Integrity Protocol

The Temporal Key Integrity Protocol (TKIP) is an improvement of the broken WEP protocol [27]. It was designed so old WEP-compatible hardware can support TKIP with only firmware updates. Nowadays TKIP is deprecated, and the more secure (AES-)CCMP is recommended. However, networks which support both TKIP and CCMP simultaneously, generally use TKIP as their group cipher. This is done for backward compatibility, so both old TKIP clients and newer CCMP clients can decrypt broadcast data. Since most routers by default allow both TKIP and CCMP, and thus use TKIP as their group cipher, TKIP is still widely used. For example, recently it has been shown that 66% of wireless networks in residential areas in Belgium use TKIP as their group cipher [29]. Note that WPA1 and WPA2 are certifications handed out by the WiFi Alliance. The difference between them is that WPA1 mandates TKIP support and optionally allows CCMP, while the reverse is true for WPA2. We use the term WPA1/2 to refer to both.

When a station connects to a secured network it begins by negotiating session keys using a 4-way handshake. This results in a pairwise transient key (PTK) and a group temporal key (GTK). These keys depend, among other things, on the MAC address of the station and AP. The PTK protects unicast traffic, while the GTK protects broadcast traffic. When using TKIP as the group cipher, the GTK is used to derive a 128-bit encryption key, and a 64-bit Message Integrity Code (MIC) key. Note that a client transmitting a broadcast frame first sends it to the AP (as a unicast frame), after which the AP broadcasts it to all stations using the group cipher. All keys are renewed after a user defined interval, commonly set to 1 hour.

When a station transmits a TKIP protected frame, it first calculates the MIC over the data field using its 64-bit MIC key (see Figure 2). Then it calculates a CRC over the data and MIC, called the Integrity Check Value (ICV). These fields are encrypted using a per packet key derived from the TKIP Sequence Counter (TSC) and 128-bit encryption key. The TSC is a replay counter which is incremented after successfully transmitting a frame. The receiver drops frames with an old TSC or bad ICV. However, if a frame has a valid TSC and ICV, but a wrong MIC, the TKIP countermeasures are activated:

- Clients send a MIC failure report to the AP. Additionally, when a client itself detects two failures within one minute, it will disconnect from the network.
- An AP silently logs the MIC failure. If two failures occur within one minute, it will halt all TKIP traffic for one minute. After this minute clients can reconnect.

Several weaknesses have been discovered in TKIP [27, 29, 18, 28, 10, 17, 21, 20]. The first known attack, and the one we built on, is the Beck and Tews attack [27]. It decrypts a packet byte by byte and requires that QoS is supported, which is the case for most modern devices [18, 29]. The attack works by guessing the last byte of an encrypted packet and then removing this byte. The ICV of the (encrypted) shortened packet can be corrected if the guess was correct. Because TKIP uses a unique TSC for each QoS channel, we can inject the corrected packet on a QoS channel with a lower TSC. If the guess was wrong, the ICV will be wrong, and the receiver drops the packet. However, if the guess was correct, the receiver will respond with a MIC failure. Hence the last byte of a packet can be found by trying all possible values and detecting a correct guess using the MIC failure report. This technique is then recursively applied to the shortened packet. To avoid the TKIP countermeasures at most one byte can be decrypted per minute. In the Beck and Tews attack this technique is used to decrypt an ARP packet. From the decrypted packet, the MIC key for that communication direction can be derived. This attack, and its variations, have previously only been designed to attack unicast traffic [27, 18, 28, 10, 29, 20].

For a more detailed background on TKIP, including practical attacks, we refer the reader to our earlier work [29].

Listing 1: Code which sets EDCA parameters for a specific access class in the ath9k_htc driver.

```
1 int ath_htc_txq_update(...):
2   qi.tqi_aifs = qinfo->tqi_aifs;
3   qi.tqi_cwmin = qinfo->tqi_cwmin / 2; /* XXX */
4   qi.tqi_cwmax = qinfo->tqi_cwmax;
```

2.4 Atheros Firmware Implementation

The open source `ath9k_htc` firmware runs on AR7010 and AR9271 chips, and is sent to the device after power up. The AR7010 is a system-on-chip that generally uses PCIe to connect to an external AR9280 wireless chip, while the AR9271 is a single-chip solution with onboard wireless chip. The AR7010 supports both the 2.4 GHz and 5 GHz band, but can only use its built-in 2x2 MIMO antenna. On the other hand, the AR9271 can use an external antenna, but only supports the 2.4 GHz band.

The wireless chip in both devices is controlled using memory mapped registers. For example, the `AR_QTXDP` register contains a pointer to the transmit queue, and writing to the `AR_Q_TXE` register will start the transmit process.

3. UNFAIR CHANNEL USAGE

In this section we study (contending) selfish stations using off-the-shelf Wi-Fi dongles. This is done by implementing several strategies and measuring the resulting throughput. Finally we bypass system designed to detect selfish users.

3.1 Experimental Setup

In order to implement selfish strategies we modified the `ath9k_htc` driver to get direct control over MAC layer parameters. While doing this we found that the original driver wrongly divided CW_{min} by two, while other parameters were properly configured (see Listing 1 line 3). This leads to an unfair advantage, and motivates other stations to act selfish as well. The driver was patched to properly set CW_{min}.

A Linksys WAG320N with firmware v1.00.08 is used as the AP. It is connected to a HP 8510p running Linux 3.7.2

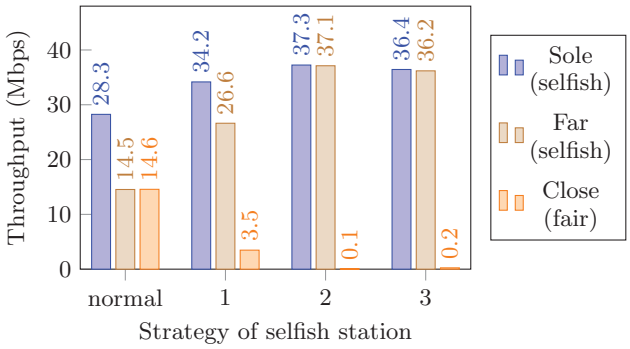


Figure 3: Throughput in case of one selfish station for the strategies in Sect. 3.2. Close and far denote contending stations, with far being the selfish one. Sole represents only one connected station. Standard deviation of all results were below 0.28 **Mbps.**

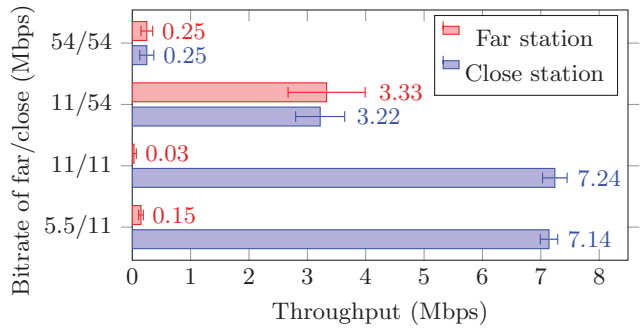


Figure 4: Impact of bitrate on contending selfish stations. Y-axis shows bitrate of far and close station, respectively. Error bars denote standard deviation.

using a gigabit ethernet connection. For the clients we created two identical VMWare Player instances, each having 2GB RAM and running Linux 3.7.2. One client uses a TL-WN722N placed 10 cm from the AP, while the other uses a AWUS036nha placed 1 meter from the AP (close distances reduce packet loss, making measurements more accurate). We refer to them as the close and far station, respectively. Reported results are over 10 runs of iperf 2.0.5 in UDP mode, using the maximum payload of an ethernet frame (1500 bytes minus 28 bytes for the IP and the UDP header), in the form bandwidth $\pm$ standard deviation. We use 802.11g, a channel width of 20 MHz, do not use the RTS or CTS protection mechanism, and do not use encryption. We assume the AP to be trusted and only consider selfish clients.

3.2 One Selfish Station

We study the behaviour a single selfish station in two phases. First when the selfish station is the only (sole) device connect to the network, and then when other fair stations are also connected to the network. For the first phase we let the close station be the only (sole) selfish station connected to the network. We implemented the following selfish strategies to determine whether they are effective or not:

1. Disabling backoff using the `AR_DMISC` register.
2. Setting *AIFSN* to zero in the `AR_DLCL_IFS` register.
3. Decreasing *SIFS* in the `AR_D_GBL_IFS_SIFS` register.

The result of these (incremental) strategies, including normal throughput, are shown in Figure 3 under "Sole (selfish)". Only results where *SIFS* was lowered to 6 are shown (our conclusion is the same for other values). We verify these results theoretically. It takes 218 μs to send 1472 bytes of UDP payload at 54 Mbps (the highest 802.11g bitrate), and 23 μs to send the long PLCP preamble and header. We have an overhead of a 34 bytes MAC header, a 20 byte IP header, a 8 byte UDP header, and a 4 byte FCS. At 54 Mbps this results in a overhead of 33 μs for each frame encapsulating a 1472 byte UDP payload. Since an ACK is 14 bytes and sent at 24 Mbps, it takes 38 μs to transmit when including the LPCP and *SIFS* timeout. Before sending the data frame, we wait on average $SIFS + (AIFSN + \frac{CW_{min}}{2}) \cdot SLOT$. Normally, when *SIFS* is 10 μs, *SLOT* is 9 μs, *AIFSN* is 3, and CW_{min} is 15, this is equal to 104.5μs. Hence it takes on average 393 μs to send 1472 bytes of data, resulting in a theoretical throughput of 29.96 Mbps. For the three selfish strategies we get 36.18 Mbps, 39.45 Mbps, and 40.53 Mbps, respectively. Our results are slightly lower because a beacon is transmitted every 102.4 ms and not all packets arrive successfully. Notice that reducing *SIFS* may negatively impact frame delivery. Hence a sole selfish station will disable backoff and set *AIFSN* to zero, but will not change *SIFS*.

The experiments were repeated when a fair station was also present. Since a malicious station may be unable to move arbitrarily close to the AP, we pick the far station as the selfish one. The results are shown in figure 3 under "Close (fair)" and "Far (selfish)". Again the optimal strategy of the selfish station is to keep the default value of *SIFS*, disable backoff, and set *AIFSN* to zero. We found this strategy works independent of the distances between stations.

3.3 Multiple Selfish Stations

We experimentally investigate the behaviour of contending selfish stations, by letting both the close and far station act selfishly. Because selfish stations generate so much traffic they hinder the arrival of beacons, it was necessary to modify the stations so they would not disconnect when too many beacons were lost. We found two strategies to increase throughput, both with as goal to exploit the capture effect:

1. Reducing the bitrate of transmissions.
2. Increasing transmit power.

These strategies work because, in case of a collision, the receiver tends to decode the frame having the lowest bitrate and the highest signal strength. Since collisions are now desired selfish stations will disable carrier sense. This can be done using the `AR_DIAG_SW` register. To avoid the station from lowering its bitrate after a failed transmission, we force both stations to use a fixed bitrate in each experiment.

First we performed experiments when contending selfish stations reduce their bitrate. We found that stations keep lowering their bitrate until this no longer provides any advantages (see Figure 4). In particular, both stations start with a 54 Mbps bitrate. Due to collisions at high bitrates most frames are lost. Assuming the far station lowers its bitrate, while the close station still uses 54 Mbps, the bitrate resulting in the maximum throughput for the far station was 11 Mbps. In response the close station will maximize its

own throughput by using 11 Mbps as well. This arms race continues until lowering the bitrate no longer provides any advantages. We conjecture that the final bitrates depend on the devices being used and on the environment. In our setting the close station ends up using 11 Mbps while the far station uses 5.5 Mbps. At this point neither station could increase throughput. Hence contending selfish stations will lower their bitrate in order to get a higher throughput.

A higher transmit power can also increase throughput, though it is more sensitive to distances. If the close station is 10 cm from the AP, the far station could not increase its throughput using a higher transmit power. However, when placing the close station 70 cm from the AP the situation changes. Assuming both stations act selfish and use 11 Mbps, the throughputs of the close and far station are 0.50 ± 0.52 and 2.10 ± 1.05 Mbps, respectively. When the far station uses an amplifier as external antenna, its throughput increases to 3.66 ± 2.08 Mbps while the close station gets 0.44 ± 0.28 Mbps. Hence selfish stations will use a higher transmit power in an attempt to increase their throughput.

At higher distances both strategies still work. However, as the distance increase, it gets harder for a selfish station to beat other stations and increase its own throughput.

3.4 Defeating Countermeasures

We now investigate mechanisms which are designed to detect (and prevent) selfish behaviour. Unfortunately we found that even advanced detection mechanisms such as DOMINO [25] have weaknesses. Though they can reliably detect manipulations of interframe spaces and the backoff procedure, defending against stations which jam others appears difficult. The goal of jamming others is to increase their contention window (backoff counter), increasing the chance the cheater can access the channel. Frames of other stations can be jammed by a selective jammer (see Sect. 4.2).

An attacker only jams frames of other clients. It detects such frames based on the addresses in the MAC header. Hence detection mechanisms assume the header of frames remain valid, and use this to count the number of failed (jammed) transmissions of stations [25]. If a station has a significantly lower failed transmission count than all others, it is assumed to be jamming the other stations, and is punished (e.g. thrown of the network). The flaw in this technique is that the authenticity of jammed (corrupted) frames cannot be guaranteed. An attacker can forge jammed frames and fool the mechanism into thinking a station is jamming others. The targeted station is then wrongly punished.

We created a tool to forge jammed frames. In particular we used the `AR_DIAG_SW` register to force the wireless chip to append a bad FCS to every transmitted frame. To target a victim we forge jammed frames which appear to come from all other stations. As a result the victim will appear to have a significant lower number of failed transmissions compared to the others, and will be wrongly punished.

We conclude that existing systems tend to underestimate the power of attackers, and hope our work gives a better overview of both the capabilities and behaviour of attackers.

4. JAMMING

In this section we show how to implement continuous and selective jamming on commodity Wi-Fi devices. To the best of our knowledge we are the first to accomplish this.

Register	Description
`AR_DIAG_SW`	Disable carrier sense, abort ongoing reception, append bad FCS
`AR_D_GBL_IFS_SIFS`	*SIFS*
`AR_D_GBL_IFS_SLOT`	*SLOT*
`AR_DLCL_IFS`	CW_{min}, CW_{max}, and *AIFSN*

Table 1: Important registers and their functionality.

Type	Wireless Chip	Device
USB	Atheros AR9271	Alfa AWUS036nha
	Atheros AR9271	TP-Link WN722N
	Atheros AR9280	Netgear WNDA3200
	Realtek RTL8187L	Alfa AWUS036h
	Ralink RT2720	Belkin F5D8053 v3
	Ralink RT3070	Digiflex aw-u150bb
PCI	Atheros AR5112	Wistron NeWeb CM9
	Broadcom bcm4334	Galaxy i9305
	Intel jc82535rde	Intel 4965AG
	Intel wg82541rde	Intel 5100AGN
Access Point (PCI)	Atheros AR2413	Sagem F@st 3464
	Broadcom bcm2050	WRT54g v2
	Broadcom bcm4322	Linksys WAG320N
	Broadcom bcm5356	Asus RT-N10

Table 2: Devices and their wireless chips. Monitor mode is supported by all USB and Intel PCI devices.

4.1 Continuous Jamming

A continuous jammer transmits noise for an indefinite amount of time. The noise can consist of random energy pulses or data resembling the protocol being jammed. Generally one of these options is preferred. For instance, injecting random Wi-Fi frames might trigger warnings from intrusion detection systems, while random noise will be seen as non-Wi-Fi devices using the same frequency.

To turn a commodity Wi-Fi device into a continuous jammer we need to be able to carry out the follow tasks:

1. Disable carrier sense.
2. Reset all interframe spaces and disable backoff.
3. Prevent the chip from waiting for an ACK.
4. Queue a large number of frames for transmission.

Achieving the first two points is explained in Sect. 3. The most important registers we used are summarized in Table 1.

To address the last two points we need to know how the wireless chip is instructed to transmit frames. The `ath9k_htc` firmware maintains a linked list of frames to be sent. Each frame is encapsulated by a transmit descriptor, which includes metadata describing how it should be transmitted. Among the metadata is the `HAL_TXDESC_NOACK` flag. When set, the wireless chip does not wait for an ACK and will not retransmit the frame. To queue an infinite number of frames we turn the linked list into one self-linked element.

We have implemented the continuous jammer by modifying the `ath9k_htc` firmware and created a tool to send commands to the firmware [1]. Since the AR7010 chip supports 2.4 and 5 GHz we can jam both bands on any channel. It is

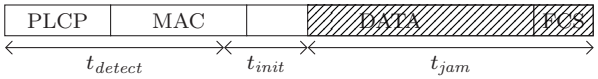


Figure 5: Timing requirements of selective jamming. After a frame header is detected and decoded, the jammer is initiated to jam the remaining content.

straightforward to turn our continuous jammer into a periodic jammer, which transmits only during certain intervals. An advantage of a periodic jammer is a lower consumption.

We tested our implementation using a WNDA3200 and AWUS036nha. Both devices are able to jam any channel in the 2.4 GHz band, with the WNDA3200 also able to jam channels in the 5 GHz band. All devices in Table 2 were susceptible to the attack: once the jammer was activated they all lost their connection to the AP. When monitoring the channel we observed that only the first frame injected by the jammer was visible. All devices in table 2 supporting monitor mode displayed this behaviour. This highlights an important difference between our jammer and the unfair channel usage described in Sect. 3. Though the high load of a selfish node could at times cause a fair station to disconnect, all its traffic would be visible in monitor mode.

The effectiveness of the jammer depends on how it is blocking frames. They can be blocked by triggering the carrier sense mechanism of the transmitter (preventing it from sending frames) or by mangling the frame at the receiver. Generally it is easier to silence a transmitter, since less power is required to trigger the carrier sense mechanism compared to mangling incoming frames. To establish an upper bound on the effectiveness of our jammer we tested the maximum distance for which the AWUS036nha could silence a TL-WN722n. This was done with a clear line-of-sight between the victim and the jammer. Testing whether a device was silenced was done by letting it inject a frame, and using a second device to monitor whether it was transmitted. With an AWUS036nha as the jammer, it was effective up to roughly 80 meters. When using an amplifier as external antenna the range was extended to roughly 120 meters.

Cheap and readily available jammers for the 2.4 and 5 GHz bands can have a high impact since many devices operate in these bands (e.g., security webcams, baby monitors, etc).

4.2 Selective Jamming

Selective jammers are arguably the most sophisticated and efficient. By targeting only selected frames an attacker can stealthy block specific frames from reaching their destination. In its simplest form a selective jammer emits noise whenever radio activity is detected. More sophisticated ones decode a prefix of a frame and then decide whether to jam the remaining content. In other words, they use higher-layer information to decide whether to jam it. The downside is that selective jammers can be difficult to implement, since they must adhere to strict timing requirements. This is illustrated in Figure 5 where t_{detect} is the time it takes to detect and decode the frame header, t_{init} the time needed to start the jammer, and t_{jam} the time when the frame is being jammed. To successfully jam a frame $t_{detect} + t_{init}$ must be lower than the time it takes to transmit the frame.

To implement selective jamming on commodity devices we must be able to accomplish the following tasks:

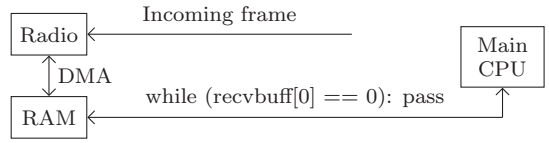


Figure 6: Using commodity devices to detect and process frames while they are still in the air.

1. Read the decoded header of a frame still in transit.
2. Abort receiving the current frame.
3. Immediately jam the channel.

Point 3 can be done by disabling carrier sense, setting all interframe spaces to zero, disabling the backoff procedure, and then injecting a dummy frame (see Sect. 4.1). Point 2 can be accomplished using the `AR_DIAG_SW` register.

There is no support to read the already decoded bytes of a frame still in transit. Nevertheless, one can still accomplish this if the wireless chip uses Direct Memory Access (DMA) to write already decoded bytes to memory. In that case we can monitor the receive buffer using the main CPU (see Figure 6). We first initialize the receive buffer with values that do not occur in valid 802.11 frames. Once these bytes are being modified, we know a frame is being received, and can read the already decoded bytes.

4.2.1 Implementation

We implemented the selective jammer by modifying the `ath9k_htc` firmware [1]. Currently it is designed to jam beacons and probe responses from a particular AP. The user can specify which SSID or MAC address to target and for how long to attack it. It is straightforward to modify our implementation to target different frames. In particular, the following fields can be used to target other frames: (1) MAC addresses in the header; (2) type and priority of frame; (3) sequence and fragment number. One can also use the first bytes of data in the packet, though this may make it difficult to jam the frame in time. Once decided to jam the frame, we inject a dummy frame transmitted at 1 Mbps to create a collision and mangle the frame. This will cause the FCS of the targeted frame to be invalid making the receiver drop the frame. To prevent the injected frame from being retransmitted we use the `HAL_TXDESC_NOACK` flag. When targeting unicast frames it is possible to inject an ACK immediately after jamming the frame. This prevents the sender from retransmitting the frame.

The selective jammer can also be modified to assure an attacker is the first to reply to certain frames. For example, an attacker can detect a probe request while it is still being transmitted, and immediately queue the transmission of a probe reply. To assure the reply is transmitted before any other station we disable backoff and set *AIFSN* to zero. A proof of concept of this attack has been implemented. The tool detects probe requests, jams them, and replies with a custom probe response. In Section 5 we use this to send probe responses with a modified channel number. Being able to reply faster than the legitimate source is known to facilitate other attacks as well. For example, it can also be used to forge DNS replies in an (unencrypted) network.

We conjecture that, with sufficient efforts, all operations can be implemented on other commodity devices as well.

4.2.2 Experiments

We selectively jammed beacons with the victim at 70 cm from the AP and the jammer at 1 m (all devices were located on a single line). The victim was put in monitor mode to track how many beacons were malformed. In each experiment we carried out the attack for 2 minutes. When an AWUS036nha jams a WNDA3200, on average 52% of the beacons are malformed. However, when an AWUS036nha is jamming a TL-WN722N, 99% of the beacons are malformed. Hence the effectiveness depends on the device of the victim. More precisely, the type of antenna, radio chip, internal noise filters, etc. can all influence how vulnerable a victim is. The device of the attacker also plays an important role. As mentioned, when an AWUS036nha uses its default antenna to jam a WNDA3200, on average 52% of the beacons are malformed. But when connecting an amplifier to the AWUS036nha and then jamming the WNDA3200, 92% of all beacons are jammed. The location of the victim and attacker also plays an important role. When a WNDA3200 jams another WNDA3200 in the 2.4 GHz band, 51% of beacons were jammed. If we switch the location of the attacker and victim, all beacons were jammed. This is expected, since the signal strength of the AP is now lower for the victim, meaning it becomes easier for the attacker to overpower it. More precisely, the attacker must transmit a signal powerful enough to overcome capture effect, otherwise the victim still correctly receives the original frame. Finally we tested our jammer in the 5 GHz band by using two WNDA3200's. In this case all beacons were jammed.

With beacons sent at 1 Mbps in the 2.4 GHz band, both the AWUS036nha and WNDA3200 start mangling bytes at position 52 of the beacon. When beacons are sent at 6 Mbps in the 5 GHz band, the WNDA3200 starts mangling bytes at position 88. This is sufficient to jam beacons, probe requests, probe responses, and other packets of similar size. Currently the jammer is limited by the time it takes the wireless chip to write the first decoded bytes to RAM. In particular we observed that the wireless chip writes to RAM only after it has decoded the first 48 bytes. Further reverse engineering may reveal a technique to force to chip to write to RAM earlier, resulting in faster reaction times.

An interesting observation was made when selectively jamming an AR5112 chip. When the signal power of the jammer was ± 16 dB higher than that of the AP, the AR5112 returned a prefix of the beacon, followed by the dummy frame injected by jammer. This was caused by the Message in Message (MiM) support of the AR5112. Devices that support MiM can resynchronise to a stronger frame while receiving a weaker frame, even if the stronger frame arrives after the preamble of the weaker frame [16]. Hence our selective jammer is an ideal tool to test whether a device supports MiM. For comparison, Lee et al. had to resort to more tedious experiments to show the AR5112 supports MiM [14].

4.2.3 Discussion

One limitation of our jammer is that it does not have access to the PLCP header of a frame still in transmit. In particular this means we are unable to access the length field in the PLCP header, and thus do not know how long the frame is. Normally the `AR_DataLen` field in the receive descriptor contains the length of the frame. However, for frames still in transmit, it contains the number of bytes written to RAM so far. We found no other data from which the length can be derived, though more analysis may overcome this obstacle.

Essentially our cheap selective jammer is capable of deterministically creating collisions. Apart from being used as a jammer, it can also be used to test new collision detection techniques, experiment with Message in Message radios, etc. In short, a cheap and easily available selective jammer not only shows that jamming poses a more serious threat than previously thought, it also facilitates research experiments.

5. CHANNEL-BASED MITM

In this section we present a man-in-the-middle attack on WPA1/2 secured networks. Our goal is not to decrypt traffic, but to reliable intercept and manipulate it. In Section 6 we use this to attack TKIP when used as a group cipher.

5.1 Background

If the goal of an attacker is to reliably sniff and manipulate traffic, merely monitoring the channel is insufficient. Inevitably packets will be missed, and it is difficult to block and manipulate traffic. Though a selective jammer can block certain packets, it is not reliable enough (see Sect. 4.2.2). Another problem is that selective jammers inherently do not have access to the complete frame, giving an attacker limited information to decide whether to block it. All these limitations disappear when having a MitM position.

Establishing a MitM position in a WPA1/2 secured network is difficult due to the 4-way handshake. This is because the generated session keys depend on the MAC address of the AP and client. Therefore, if we use a rogue AP with a different MAC address, the handshake will fail. Using the same MAC address as the real AP is not possible since the client and AP would simply communicate with each other.

5.2 Intercepting Encrypted Traffic

To intercept all traffic we will clone the AP on a different channel and forward all traffic to the real AP. This requires two Wi-Fi dongles: one operating on the channel of the real AP, and one cloning the AP on a different channel. Because both Wi-Fi dongles are physically close to each other they can receive each others frames, even though they operate on different channels. Hence, blindly forwarding packets between both channels may create an infinite loop. To avoid this we keep track of recently forwarded frames using their sequence numbers, and only forward new frames. To advertise our rogue AP we transmit a beacon every 102.4 ms and reply to probe requests using custom probe responses.

Once our rouge AP is started, we want to force clients to connect to it. Unfortunately, injecting probe responses containing the channel of the rogue AP does not cause clients to switch channels. Selectively jamming all beacons and probe responses is also not reliable, as some frames will inevitably be missed. Additionally we found that if a client was recently connected to an AP, it will simply assume it is still present, and immediately send an authentication frame. This technique allows a mobile device to quickly reconnect to a network. The authentication message is only 34 bytes long and too short to be selectively jammed. Hence selective jamming cannot be used to force clients connect to the rogue AP. To avoid all these issues we continuously jam the channel of the real AP (see Sect. 4.1). This forces the clients to switch to our channel and connect to our rogue AP. Once the clients have switched, we can stop the jammer, and start forwarding frames between the clients and the real AP.

5.3 Implementation

We implemented the attack in a command-line tool [1]. It allows a user to specify which AP to clone, whether to jam the channel of the real AP until clients connect to our AP, and whether to write all intercepted traffic to file.

The firmware was modified so injected frames are retransmitted in case of failure, and so ACKs are generated when frames are sent to us. Retransmitting frames is done by disabling the `HAL_TXDESC_NOACK` flag. Generating ACKs on the device cloning the AP is straightforward, as it only needs to listen on the MAC address of the AP. However, the device on the channel of the real AP must listen to the MAC addresses of all connected clients. To accomplish this we rely on virtual interface support, a hardware technology enabling a single device to listen on multiple MAC addresses. When a client is trying to connect to the rogue AP, our modified firmware simulates the addition of a new virtual interface.

We had to patch the driver and firmware to prevent modifications to forwarded frames. In particular we made the driver treat injected data frames as management frames. In the firmware we marked injected frames as CF-Poll frames to prevent the sequence number from being overwritten.

After establishing a MitM position an attacker will reliably capture all traffic. Additionally, packets can be blocked by not forwarding them. It is straightforward to update the tool so the attack can be executed even when the target is far away from the AP by forwarding frames over the internet.

5.4 Experiments

We performed several experiments to measure the reliability and impact of the attack. For the victim we used a Latitude E6500 running Linux 3.7.2, for the AP a Linksys WAG320N using firmware v1.00.08, and as attack machine a VMWare player instance having 2 GB RAM and running Linux 3.7.2 with our modified drivers and firmware. We used a TL-WN722N and WNDA3200 to intercept and forward traffic, and an AWUS036nha as the jammer. The AP was configured to operate in 802.11g mode.

When using continuous jamming to force clients to connect to our rogue AP, we consistently established a MitM position. Even clients already connected to the real AP switched to the rogue AP. We then measured the impact on latency, bandwidth, and web page loading times. When connected to the real AP the victim had a latency of 3.82 ± 10.4 ms. This increased to 7.45 ± 12.9 ms when connected to our rogue AP. Hence latency is doubled, which is expected since every packet must be forwarded by our rogue AP, and is thus transmitted twice. To test the impact on throughput we let the victim download a 100 MB file. Under normal conditions this takes place at 18.6 Mbps. When being attacked the speed is lowered to 8 Mbps. Finally we tested the impact on page load times when surfing the web. Under normal conditions a page was loaded in 0.76 ± 0.07 ms, which increased to 0.83 ± 0.08 ms when under attack. Since this is only a 9% slowdown users are unlikely to notice this and will not realise they are under attack.

These results can be optimized by implementing a rate adaptation algorithm when forwarding packets, and by using unfair MAC parameters as described in Sect. 3.

5.5 Countermeasures

There are legitimate reasons for an AP to change channel, for example to switch to a less occupied one. In the 5 GHz band, switching channels is even required to avoid interference with radars [11, §4.5.5.3]. Hence storing the channel of an AP you previously connected to is not feasible.

The attack can be detected by including the channel of the AP in the 4-way handshake. Unfortunately this requires a change in the existing protocol, making this difficult to realise in practise. Ideally the impact of the attack is reduced by using a secure encryption protocol such as CCMP.

6. TKIP AS A GROUP CIPHER

In this section we attack TKIP when used as a group cipher, which is the default security setting for most routers.

6.1 Attack Details

Previously TKIP was only investigated when used to protect unicast traffic (see Sect. 2.3). Our goal is to show that TKIP can also be attacked when used as a group cipher, i.e., when used to protect broadcast and multicast frames.

Directly applying the Beck and Tews attacks on broadcast packets fails when multiple clients are connected to the AP. In that case all clients will simultaneously send a MIC failure report (instead of only the targeted client as in the unicast scenario). Hence the AP immediately starts the TKIP countermeasures. Recall that the countermeasures disable all TKIP traffic for one minute, after which a new GTK is generated and clients can reconnect. Thus we would only be able to decrypt one byte. Our selective jammer cannot be used to block the MIC failure reports, because they are detected based on their unique length. More troublesome, MIC failures are generally sent at high bitrates, meaning even more advanced selective jammers are unable to reliably jam them. Instead we use the MitM attack from Sect. 5, allowing us to block MIC failure reports by not forwarding them. Note that we must still assure that individual clients do not send more than one MIC failure report every minute. Otherwise the client will disconnect from the network, even if the AP did not start the TKIP countermeasures.

Once a MitM position has been achieved we wait until a small broadcast packet is transmitted. We prefer small packets as these take less time to decrypt. Since ARP requests are small and sent when a client (re)connects to a network, they are an ideal candidate. After capturing an ARP request we decrypt its ICV and MIC using the Beck and Tews method, and guess the remaining content. Since we do not forward MIC failures to the AP, the TKIP countermeasures will not be actived when multiple clients send a MIC failure report. Once the packet has been decrypted we learn the keystream corresponding to the sequence counter (TSC) of the ARP request, and we can derive the MIC key for broadcast traffic [27]. With this we can inject 3 to 7 small packets to any client connected to the AP (the amount depends on the number of supported QoS channels). Furthermore, existing attacks relying on knowing the MIC key can be modified to work on broadcast packets. For example, by modifying the attacks presented in [29], we can abuse fragmentation to inject an arbitrary amount broadcast of packets, and we can efficiently decrypt arbitrary broadcast packets.

We continue by speeding up the attack. There are several techniques to accomplish this. First, if we do not detect a MIC failure after trying all possible 256 values, we know the target did not receive the packet with the correct guess. If it did, it would retransmit the MIC failure until we responded with an ACK. Hence we can immediately retry guessing all

values without fear of generating two MIC failures within one minute. In contrast, previous attacks on TKIP had to wait one minute because the attacker might have missed the MIC failure report while the AP did receive it.

If we are able to send broadcast packets to only one client we can further speed up the attack. We start the attack by sending guesses to one particular client. Once it sends a MIC failure report, we can immediately continue the attack by targeting another client. Each client will only see one MIC failure every minute, and the AP will see none, hence the TKIP countermeasures are never activated. We keep targeting different clients while assuring that an individual client will never send more than one MIC failure report every minute. The difficulty is in sending a broadcast packet to a specific client. Though it is possible by cloning the AP on multiple channels and spreading the clients out over these channels, such an approach is cumbersome and requires multiple Wi-Fi devices. Instead we rely on the observation that most devices do not process their own broadcast packets. That is, most devices will drop broadcast packets with as source address their own MAC address. Assuming we are attacking two clients simultaneously, we can target one client by using the source address of the other client. Finally we do not have to wait one minute when guessing the last remaining bytes. Though a client may send more than one MIC failure report within one minute, and therefore disconnect from the network, the AP will not see these MIC failures. Hence the AP will not activate its countermeasures.

6.2 Implementation and Experiments

We implemented the attack by extending the MitM implementation (see Sect. 5). To attack two clients we rely on the fact that a client will not process broadcast packets with as source its own MAC address. Once the clients are connected to our rogue AP, and completed the 4-way handshake, we wait for an ARP request. The ARP request is decrypted byte by byte, and finally the MIC key is derived.

Our experimental setup is the same as in Sect. 5.4, except that we now also use a Samsung Galaxy i9305 running Android 4.3 as a second victim. During the experiment we first let the two victims connect to the real AP, after which we started the attack. The execution time over 10 runs was 7.3 ± 0.6 minutes. This is the time from launching the tool to decrypting the APR reply and deriving the MIC key. Note that the execution time can be further reduced by targeting more stations simultaneously. Program output and network traces of the attack are available for download [1].

As a comparison we measured the execution time of the original Beck and Tews attack. The attacker used a TL-WN722N and the victim an AWUS036h. They were placed close to each other to reduce packet loss, an hence represents an optimal case. We used the tkiptun-ng tool from aircrack-ng. The execution time over 10 runs was 14.6 ± 1.1 minutes, more than twice the time of our attack.

6.3 Countermeasures

To prevent the interception of MIC failure reports, APs should securely acknowledge them. If the client does not receive the acknowledgement within time, it may be under attack, and should activate its TKIP countermeasures. This prevents an attacker from blocking MIC failure reports and attacking multiple stations simultaneously.

The attack can be mitigated by using a short rekeying timeout of 2 minutes or less. Ideally the attack is prevented by using a more secure encryption protocol such as CCMP.

7. RELATED WORK

Atheros drivers are popular for low-layer control over experiments, relevant examples are [25, 14, 8]. Custom firmware has also been used for low-layer control [3, 4, 9].

A significant amount of research on the behaviour of selfish stations rely only on analytic models and simulations [23, 6, 24]. Raya et al. perform experimental tests when a single node manipulates the contention window of the backoff procedure [25]. Pelechrinis et al. also perform experiments, but only consider stations which modify their CCA threshold [22]. However, an attacker can easily change additional parameters. We are not aware of any works doing a detailed experimental study on the behaviour of selfish stations, where multiple strategies are tested, and the capture effect is taken into account. The capture has been observed for 802.11 on Prism and Atheros chipsets [13, 8]. In [8] the authors attempt to reduce the unfairness caused by this effect. Lee et al. perform a detailed study on when the capture effect takes places, and found that the Atheros AR5112 not only exhibits capture effect but also supports Message in Message mode [14].

Bayraktaroglu et al. required a USRP to continuously jam the 2.4 GHz band [2]. Kim et al. modify the MadWifi driver of the Atheros 5212 to continuously emit meaningless frames in the 2.4 GHz band [12]. Noubir et al. used one USRP1 and two RFX2400 boards to build a selective jammer [19]. Unfortunately the reaction time of their setup was too slow to reliably jam frames [19, §6.1.4]. Cassola et al. required two USRP2s and two RFX2400 boards, totalling more than $3600, to selectively jam frames in the 2.4 GHz band [7]. Their jammer starts mangling frames sent using 1 Mbps at byte 38, while our jammer does this at byte 52. However, our dongle costs only $15, can jam both frequency bands, and is easier in use due to it's smaller size. In concurrent and independent work, Berger et al. used custom microcode to have control of medium access and de/encoding operations, and used this to implement selective jamming in the 2.4 GHz band [4]. The advantage of our approach is that we do not require such low-level control over medium access operations performed in the chip. Continuous and selective jammers have also been created to target other wireless protocols [30, 31]. Our selective Wi-Fi jammer is unique because it is both cheap and portable. Jamming can also be used as a defensive mechanism [6, 4, 30].

Beck and Tews found the first practically exploitable vulnerability in TKIP, which required QoS enhancements to be enabled [27]. Ohigashi and Morri used a MitM position to execute the attack even if QoS was not enabled [20]. In particular they assumed the client was not in range of the AP, and that the attacker acted as a repeater by forwarding all frames between the client and AP. In contrast, our MitM attack does not require that the client and AP are out of range. Over the years additional improvements of the Beck and Tews attack have been found [10, 18, 28, 29]. Several more theoretical attacks on TKIP have also been published [17, 26, 21]. Finally, the closest related work to our channel-based MitM attack we are aware of is the AirJack tool [15]. It clones an unencrypted network on a different channel and forwards all traffic over ethernet. Hence it cannot clone encrypted networks, whereas our attack can.

8. CONCLUSION

We were able to implement several low-layer attacks on Wi-Fi using open source Atheros firmware. This is surprising, since we only have access to a limited API to control the radio (e.g. we cannot transmit arbitrary signals, do not have access to raw signal data, don't modify medium access algorithms, etc). Additionally we bypassed systems designed to prevent some of these attacks, indicating that previous works have underestimated the capabilities of attackers. We also showed that our low-layer attacks facilitate attacks on higher-layer protocols, by attacking TKIP when used as a group cipher. Since TKIP is used significantly more as a group cipher than as a unicast cipher, this demonstrates that weaknesses in TKIP are still of high practical value.

Finally, the selective jammer, channel MitM, and TKIP attacks are available for download [1]. We hope these results aid in the creation of better countermeasures, and motivate people to only use the more secure (AES)-CCMP.

9. ACKNOWLEDGEMENTS

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).

10. REFERENCES

[1] `http://modwifi.bitbucket.org/`.

[2] E. Bayraktaroglu, C. King, X. Liu, G. Noubir, R. Rajaraman, and B. Thapa. On the performance of IEEE 802.11 under jamming. In *INFOCOM*, 2008.

[3] J. Bellardo and S. Savage. 802.11 denial-of-service attacks: real vulnerabilities and practical solutions. In *Proc. of the 12th USENIX Security Symp.*, 2003.

[4] D. S. Berger, F. Gringoli, N. Facchi, I. Martinovic, and J. Schmitt. Gaining insight on friendly jamming in a real-world IEEE 802.11 network. In *WiSec*, 2009.

[5] G. Berger-Sabbatel, A. Duda, O. Gaudouin, M. Heusse, and F. Rousseau. Fairness and its impact on delay in 802.11 networks. In *GLOBECOM*, 2004.

[6] M. Cagalj, S. Ganeriwal, I. Aad, and J.-P. Hubaux. On selfish behavior in CSMA/CA networks. In *INFOCOM*, 2005.

[7] A. Cassola, W. Robertson, E. Kirda, and G. Noubir. A practical, targeted, and stealthy attack against wpa enterprise authentication. In *NDSS Symp.*, Apr. 2013.

[8] S. Ganu, K. Ramachandran, M. Gruteser, I. Seskar, and J. Deng. Methods for restoring MAC layer fairness in IEEE 802.11 networks with physical layer capture. In *REALMAN*, 2006.

[9] D. Halperin, W. Hu, A. Sheth, and D. Wetherall. Tool release: Gathering 802.11n traces with channel state information. *ACM SIGCOMM CCR*, 2011.

[10] F. M. Halvorsen, O. Haugen, M. Eian, and S. F. Mjølsnes. An improved attack on TKIP. In *14th Nordic Conf. on Secure IT Systems (NordSec)*, 2009.

[11] IEEE Std 802.11-2012. *Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications*, 2012.

[12] Y. S. Kim, P. Tague, H. Lee, and H. Kim. Carving secure wi-fi zones with defensive jamming. In *ASIACCS*, 2012.

[13] A. Kochut, A. Vasan, A. U. Shankar, and A. Agrawala. Sniffing out the correct physical layer capture model in 802.11b. In *ICNP*, 2004.

[14] J. Lee, W. Kim, S.-J. Lee, D. Jo, J. Ryu, T. Kwon, and Y. Choi. An experimental study on the capture effect in 802.11a networks. In *Proc. of the 2nd ACM Intl. Workshop on Wireless Network Testbeds, Experimental Evaluation and Characterization*, 2007.

[15] M. Lynn and R. Baird. Advanced 802.11 attack. In *Black Hat Briefings*, 2002.

[16] J. Manweiler, N. Santhapuri, S. Sen, R. Roy Choudhury, S. Nelakuditi, and K. Munagala. Order matters: Transmission reordering in wireless networks. In *MobiCom*, 2009.

[17] V. Moen, H. Raddum, and K. J. Hole. Weaknesses in the temporal key hash of wpa. *Mobile Computing and Comm. Review*, 2004.

[18] M. Morii and Y. Todo. Cryptanalysis for rc4 and breaking wep/wpa-tkip. *IEICE Trans.*, 2011.

[19] G. Noubir, R. Rajaraman, B. Sheng, and B. Thapa. On the robustness of ieee 802.11 rate adaptation algorithms against smart jamming. In *WiSec*, 2011.

[20] T. Ohigashi and M. Morii. A practical message falsification attack on wpa. In *Joint Workshop on Information Security (JWIS)*, 2009.

[21] K. G. Paterson, B. Poettering, and J. C. Schuldt. Plaintext recovery attacks against wpa/tkip, 2013.

[22] K. Pelechrinis, G. Yan, S. Eidenbenz, and S. Krishnamurthy. Detecting selfish exploitation of carrier sensing in 802.11 networks. In *INFOCOM*, 2009.

[23] O. Queseth. The effect of selfish behavior in mobile networks using CSMA/CA. In *Proc. of the 61st IEEE Vehicular Technology Conf.*, 2005.

[24] S. Radosavac, J. S. Baras, and I. Koutsopoulos. A framework for MAC protocol misbehavior detection in wireless networks. In *Proc. of the 4th ACM workshop on Wireless security*, WiSe '05, 2005.

[25] M. Raya, J.-P. Hubaux, and I. Aad. DOMINO: a system to detect greedy behavior in EEE 802.11 hotspots. In *MobiSys*, 2004.

[26] P. Sepehrdad, S. Vaudenay, and M. Vuagnoux. Statistical attack on rc4 distinguishing wpa. In *EUROCRYPT*, 2011.

[27] E. Tews and M. Beck. Practical attacks against WEP and wpa. In *WiSec*, 2009.

[28] Y. Todo, Y. Ozawa, T. Ohigashi, and M. Morii. Falsification attacks against wpa-tkip in a realistic environment. *IEICE Trans.*, 2012.

[29] M. Vanhoef and F. Piessens. Practical verification of wpa-tkip vulnerabilities. In *ASIACCS*, 2013.

[30] M. Wilhelm, I. Martinovic, J. B. Schmitt, and V. Lenders. Wifire: A firewall for wireless networks. In *SIGCOMM*, 2011.

[31] W. Xu, W. Trappe, Y. Zhang, and T. Wood. The feasibility of launching and detecting jamming attacks in wireless networks. In *Proc. of ACM MobiHoc*, 2005.

Whitewash: Outsourcing Garbled Circuit Generation for Mobile Devices

Henry Carter
Georgia Institute of Technology
carterh@gatech.edu

Charles Lever
Georgia Institute of Technology
chazlever@gatech.edu

Patrick Traynor
University of Florida
traynor@cise.ufl.edu

ABSTRACT

Garbled circuits offer a powerful primitive for computation on a user's personal data while keeping that data private. Despite recent improvements, constructing and evaluating circuits of any useful size remains expensive on the limited hardware resources of a smartphone, the primary computational device available to most users around the world. In this work, we develop a new technique for securely outsourcing the generation of garbled circuits to a Cloud provider. By outsourcing the circuit generation, we are able to eliminate the most costly operations from the mobile device, including oblivious transfers. Our proofs of security show that this technique provides the best security guarantees of any existing garbled circuit outsourcing protocol. We also experimentally demonstrate that our new protocol, on average, decreases execution time by 75% and reduces network costs by 60% compared to previous outsourcing protocols. In so doing, we demonstrate that the use of garbled circuits on mobile devices can be made nearly as practical as it is becoming for server-class machines.

1. INTRODUCTION

Mobile devices have become one of the dominant computing platforms, with approximately 57% market penetration in the United States alone [9]. These devices are capable of gathering and storing all of a user's personal data, from current location and social contacts to banking and electronic payment information. Because of the personal nature of these devices, it is critical that a user's information be protected at all times. Unfortunately, many smartphone applications that require users to send data to application servers make preserving the privacy of this data difficult.

To resolve this issue, a variety of secure multiparty computation techniques exist that could be leveraged to perform computation over encrypted inputs [6, 11, 12, 28]. Currently, the most practically efficient two-party technique is the Yao Garbled Circuit [43]. Despite recent improvements in the efficiency of garbled circuits [27, 41], this technique still requires significant computation and communication resources, rendering it impractical for most smartphones. One possible solution to this imbalance of resources is to blindly outsource the heavy computation to the Cloud. However, because of the untrusted nature of Cloud providers [42], such a solution fails to provide measurable guarantees for applications requiring high assurance.

In this work, we develop a new protocol for securely outsourcing garbled circuit generation to an untrusted Cloud. We construct a protocol that offloads the role of generating the garbled circuit from the mobile device to the Cloud without exposing any private inputs or outputs. By choosing to outsource this portion of the protocol, we eliminate a significant number of expensive public-key cryptography operations and rounds of communication used in oblivious transfer. The result is a more computationally and bandwidth efficient outsourcing protocol with the strongest security guarantees of any outsourcing technique to date [8, 23].

In this paper, we make the following contributions:

- **Develop a new outsourcing protocol:** We develop the Whitewash[1] outsourcing protocol, which allows a mobile device participating in a two-party secure function evaluation to outsource the generation of the garbled circuit. Our protocol assigns the mobile device the role of circuit generator instead of circuit evaluator, outsourcing a completely different set of operations from previous outsourcing protocols [8, 23]. By reversing the functions of the two players, we fully eliminate the requirement for any oblivious transfers, outsourced or otherwise, to or from the mobile device. This "simple" role reversal requires fundamentally redesigning the outsourcing techniques used in previous work, as well as new security proof formulations.
- **Formal verification and analysis:** We formally prove the security of our outsourcing techniques in the malicious model defined by Kamara et al. [23]. Unlike previous work [8, 23], our protocol provides security when the mobile device is colluding with its Cloud provider against the application server. *This added security guarantee makes our protocol the most secure outsourcing protocol to date.* We then provide an analysis of the reduction in operations between our work and the outsourced oblivious transfer of Carter et al. [8], as well as the Salus framework by Kamara et al. [23]. Specifically, our protocol requires more executions of a pseudorandom number generator in exchange for fewer algebraic group operations and zero-knowledge proofs. Moreover, we significantly reduce the number of rounds of communication required to the mobile device.
- **Implement and evaluate the performance of our protocol:** In our performance evaluation, we demonstrate a maximum improvement of 98% in execution time and 92% improvement in bandwidth overhead compared to Carter et al. [8] (with 75% and 60% average improvement, respectively). For a different

[1] A reference to Tom Sawyer, who "outsourced" his chores to his friends without ever revealing the true nature of the work.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.
ACSAC '14 December 08 - 12 2014, New Orleans, LA, USA
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-3005-3/14/12:.$15.00
http://dx.doi.org/10.1145/2664243.2664255

test application, when compared to performing computation directly on the mobile device [28], we demonstrated a 96% and 90% improvement in execution time and bandwidth, respectively. These improvements allow for the largest circuits evaluated on any platform to be computed from a mobile device efficiently and with equivalent security parameters to non-mobile protocols.

The rest of this work is organized as follows: Section 2 provides detail on related research; Section 3 describes our threat model and security definition; Section 4 provides a description of the Whitewash protocol; Section 5 compares the operations required in our protocol to the protocols by Carter et al. and Kamara et al.; Section 6 describes our empirical performance analysis; and Section 7 provides concluding remarks.

2. RELATED WORK

Fairplay [35] provided the first practically efficient implementation of Yao's garbled circuit protocol [43], requiring only simple hash and symmetric key operations to securely evaluate an arbitrary function. Since then, a variety of garbled circuit-based secure function evaluation (SFE) protocols have been developed in the semi-honest adversarial model [5, 16, 20, 21, 29, 31, 34, 39]. The latest of these, developed by Huang et al. [16], allows garbled circuits to be evaluated in stages, which makes it the most efficient semi-honest garbled circuit evaluation technique, both in computation and memory requirements. In recent work, several garbled circuit SFE protocols have been developed in the malicious security model, which require significantly more computational resources than semi-honest protocols, but are secure against arbitrary polynomial-time adversaries [18, 26, 28, 30, 33, 37, 40]. The protocol by shelat and Shen [41] provides a two-party garbled circuit protocol which uses only symmetric-key constructions outside of the oblivious transfer. When combined with Huang's pipelining approach and the PCF compiler by Kreuter et al. [27], their protocol is among the most efficient maliciously-secure garbled circuit protocols implemented to date. Some efforts have been made to improve the efficiency of these protocols by slightly reducing the adversary model. Many schemes have been developed in the covert adversary model, which allows for some efficiency gains at the cost of security [2, 10, 15, 36]. Huang et al. [17] developed a protocol that leaks only one bit of input to a malicious adversary through dual execution, which was later implemented on GPUs by Husted et al. [19]. In order to further improve the efficiency of garbled circuit protocols, Gordon et al. [13] developed a protocol that combined Oblivious RAM with garbled circuits, allowing sub-linear amortized complexity. However, this protocol only allows this performance gain for functions that can be computed efficiently on a random-access machine. Other work has focused on making circuit compilation and garbling more efficient [4, 38]. These techniques improve the underlying operations found in all garbled circuit execution protocols, providing improvement in all existing techniques.

To further improve the speed of cryptographic protocols on devices with minimal computational resources, the idea of outsourcing cryptographic operations has been explored for many years in the field of Server-assisted cryptography [3]. More recently, Green et al. [14] developed a technique for outsourcing the costly decryption of attribute-based encryption schemes to the cloud without revealing the contents of the ciphertext. Atallah and Frikken [1] developed a set of special-purpose protocols for securely outsourcing Linear Algebra computations to a single cloud server. For data mining applications, Kerschbaum recently developed an outsourced set intersection protocol using homomorphic encryption techniques [24]. While all of these applications provide significant performance gains for specific cryptographic applications, none of them address outsourcing of general secure computation.

In their Salus protocol, Kamara et al. [22, 23] developed two protocols for securely outsourcing the computation of arbitrary functions to the cloud. Following Salus, Carter et al. [8] developed an outsourcing protocol based on the maliciously secure garbled circuit protocol by Kreuter et al. [28]. Carter's protocol outsources the evaluation of garbled circuits by adding in an Outsourced Oblivious Transfer primitive. Their participant configuration is the same configuration found in Kamara's maliciously secure protocol, where the cloud is made responsible for evaluating the garbled circuit. In this work, we choose to build on shelat and Shen's latest protocol [41] since the symmetric execution environment of Huang et al. [18] does not lend itself to outsourcing, and the bootstrapping technique used by Lindell [30] has not been implemented or evaluated in practice. Unlike previous work, we choose to fundamentally rearrange the roles of the participants, outsourcing the generation of the garbled circuits as in Kamara's covertly secure protocol.

3. OVERVIEW AND DEFINITIONS

3.1 Protocol Goals and Summary

The primary reason for developing an outsourcing protocol for secure function evaluation is to allow two parties of asymmetrical computing ability to securely compute some result. Current two-party computation protocols assume both parties are equipped with equivalent computing resources and so require both parties to perform comparable operations. However, when a mobile device is taking part in computation with an application server, some technique is necessary to reduce the complexity of the operation on the mobile device. Ideally, we can make the mobile device perform some small number of operations that is independent of the size of the circuit being evaluated.

In constructing such a protocol, there are four goals that we would like to satisfy. The first of these goals is correctness. It is necessary that an outsourcing protocol must produce correct output even in the face of malicious players attempting to corrupt the computation. The second desirable guarantee is security. SFE protocols frequently use a simulation-based approach to defining and proving security, which we outline in detail below. Essentially, the goal is to show that each party can learn the output of the computed function and nothing else. Third, an ideal protocol would provide some guarantee of fair release. This guarantee ensures that either both parties receive their outputs from the computation, or neither party receives their output. Our protocol achieves this in all but one corruption scenario by treating the Cloud as an arbiter, who will simultaneously and fairly release the outputs of the protocol using one-time pads. In the scenario where the mobile device and Cloud are colluding, it is possible for the Cloud to terminate the protocol after the mobile device receives output but before the application server receives output. However, this is inherently possible in most two-party garbled circuit protocols. The fourth goal of our protocol is efficiency. Our outsourcing protocol balances a minimal set of operations on the mobile device with efficient multiparty computation operations on the application server and Cloud.

Given these goals, we build our protocol in the two-party server-aided multiparty computation setting. This setting is composed of two parties, the mobile device and an application server, who wish to run a two-party secure computation while keeping both of their inputs private. To assist the mobile device, the server-aided setting adds a third party Cloud provider, which is independent and non-colluding with the application server (more on non-collusion in the following section). The Cloud performs cryptographic operations

for the mobile device, but is not allowed to see any party's input or output from the computation.

To achieve our goals in this setting, we first select the most efficient two-party garbled circuit computation protocol to date that provides guarantees of correctness and security in the malicious model. We assign the mobile device the role of circuit generator in this protocol, and the application server is assigned the role of circuit evaluator. To outsource the circuit generation operations from the mobile device, we allow the device to generate short random seeds and pass these values to a Cloud computation provider, which then generates the garbled circuits using these seeds to generate randomness. Thus, the mobile device's work is essentially reduced to (1) generating random strings on the order of a statistical security parameter, and (2) garbling and sending its input values to the evaluating party. In this way, we develop a secure computation protocol where the mobile device performs work that is independent of the size of the function being evaluated. We provide a formal proof of security for our protocol in our technical report [7].

3.2 Non-collusion

To maintain security, previous outsourcing protocols assume that neither party colludes with the Cloud [8, 23]. The theoretical intuition for this constraint, outlined by Kamara et al. [23], is that the existence of an outsourcing protocol where parties can arbitrarily collude would imply a two-party secure multiparty computation protocol where one party performs sub-linear work with respect to the size of the circuit. While this has been shown to be possible using fully homomorphic encryption and, in some cases, oblivious RAM [13], it is not clear that these techniques can be efficiently applied to a garbled circuit outsourcing scheme. Because of this, previous work has left the more complex security model for future study. However, while previous protocols restrict collusion between the Cloud and any party, the sub-linear work implication only applies to cases when the Cloud is generating circuits and colludes with the evaluating party, or vice versa. In the Whitewash protocol, we prove security when the mobile device colludes with the Cloud against the evaluating web application. While this collusion scenario removes the fair release guarantee of our protocol, it in no way compromises the security guarantees of confidentiality of participant's inputs and outputs. Essentially, it reduces to the two-party computation scenario that the underlying protocol is proven to be secure in. Since the mobile device is paying the Cloud for computation services, we believe it is a more realistic assumption to assume that a Cloud provider could collude maliciously with the paying customer, and note that our protocol is the first outsourcing protocol to provide any security guarantees in the face of collusion with the Cloud.

3.3 Security Constructions

In the two-party computation protocol underlying our work, shelat and Shen implement a number of new and efficient cryptographic checks to ensure that none of the parties participating in the computation behave maliciously. We provide an overview of these security checks in the following section. We refer the reader to shelat and Shen's work for more formal definitions and proofs [41].

3.3.1 k-probe-resistant input encoding

When working with garbled circuit protocols in the malicious model, the generator has the ability to learn information about the evaluator's input by corrupting the wire labels sent during the oblivious transfer. This attack, known as selective failure, was first proposed by Mohassel and Franklin [37] as well as Kiraz and Schoenmakers [25]. In the server-aided setting, it is possible that the mobile device and the Cloud could collude and carry out this attack to recover the application server's input. To prevent this attack, shelat and Shen [41] implement an improved version of the k-probe-resistant input encoding mechanism originally proposed by Lindell and Pinkas [32]. In their protocol, the evaluator does not input her real input y to the computation, but chooses her input $\overline{y}$ such that $\mathbf{M} \cdot \overline{y} = y$ for a k-probe resistant matrix $\mathbf{M}$. Intuitively, the idea is that the generator would have to probe the evaluator's input approximately 2^k times before learning anything about her input y.

3.3.2 2-Universal Hash Function

A second concern with garbled circuits in the malicious model is that the generator may send different input values for each of the evaluated circuits from the cut-&-choose. As in the two-party setting, it is possible for the mobile device to submit inconsistent inputs to the application server in the server-aided setting. To ensure that the generator's inputs are consistent across evaluation circuits, shelat and Shen implement an efficient witness-indistinguishable proof, which computes a randomized, 2-universal hash of the input value using only arithmetic operations on matrices. Because of the regularity guarantees of a 2-universal hash, the outputs of these hash operations can be seen by the evaluator without revealing any information about the generator's inputs. However, if any of the hashed input values is inconsistent across evaluation circuits, the evaluator can infer that the generator provided inconsistent inputs, and can terminate the protocol.

3.3.3 Output proof of consistency

When a function being evaluated using garbled circuits has separate, private outputs for the generating and evaluating parties, it is necessary to ensure that the evaluating party does not tamper with the generating party's output. Since the output must be decoded from the garbled output wires for the majority check at the end of the protocol, if the output is only blinded with a one-time pad, this allows the evaluator the opportunity to change bits of the generator's output. Our setting faces the same potential for attack from the application server, who is responsible for evaluating the circuit and distributing the blinded output. Several techniques for preventing this kind of tampering have been proposed, but shelat and Shen's latest protocol [41] implements a witness-indistinguishable proof that uses only symmetric key cryptographic operations. After the evaluator sends the blinded output of computation to the generator, the proof guarantees to the generator that the output value he received was actually generated by one of the garbled circuits he generated. However, it keeps the index of the circuit that produced the output hidden, as this could leak information to the generator.

3.4 Security Model and Definition

Our definition of security is based on the definition proposed by Kamara et al. [23], which we specify for the two-party setting as in Carter et al. [8]. We provide a brief description of the real/ideal world model here and direct readers to the previous work in this space for a more formal definition.

In the real world, both parties participating in the computation (the mobile device and application server) provide an input to the computation and an auxiliary input of random coins, while the single party designated as the outsourcing party (the Cloud) provides only random auxiliary input. The party evaluating the circuit in this computation is assumed to be non-colluding with the outsourcing party, as defined by Kamara et al. Some subset of these parties $A = (A_1, A_2, A_3)$ are corrupted and can deviate arbitrarily from the protocol. For the ith honest party, let OUT_i be its output, and for the ith corrupted party, let OUT_i be its view of the protocol

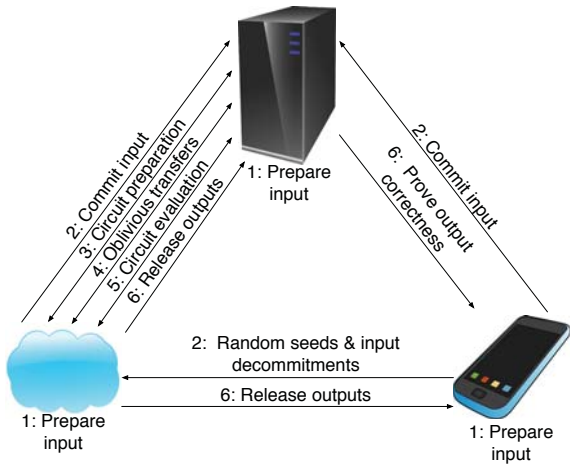


Figure 1: The complete Whitewash protocol. Note that MOBILE performs very little work compared to SERVER and CLOUD.

execution. Then the ith partial output of a real protocol execution with input x is defined as:

$$REAL^{(i)}(k, x; r) = \{OUT_j : j \in H\} \cup OUT_i$$

Where H is the set of honest parties, r is all random coins of all participants, and k is the security parameter.

In the ideal world, each party provides the same inputs as in the real world, however, they are sent to a trusted oracle which performs the secure computation. Once the trusted oracle completes the computation, it returns the output to the participating parties and no output to the outsourcing party. If any party aborts early or sends no input to the oracle, the oracle aborts and does not send the output to any party. For the ith honest party, let OUT_i be its output to the oracle, and for the ith corrupted party, let OUT_i be an arbitrary output value produced by the party. Then the ith partial output of an ideal protocol execution in the presence of independent malicious simulators $S = (S_1, S_2, S_3)$ is defined as:

$$IDEAL^{(i)}(k, x; r) = \{OUT_j : j \in H\} \cup OUT_i$$

Where H, r, and k are defined as before. Given this model, security is formally defined as:

DEFINITION 1. *An outsourcing protocol securely computes the function f if there exists a set of probabilistic polynomial-time (PPT) simulators $\{Sim_i\}_{i\in[3]}$ such that for all PPT adversaries (A_1, A_2, A_3), inputs x, auxiliary inputs z, and for all $i \in [3]$:*

$$\{REAL^{(i)}(k, x; r)\}_{k\in N} \overset{c}{\approx} \{IDEAL^{(i)}(k, x; r)\}_{k\in N}$$

Where $S = (S_1, S_2, S_3)$, $S_i = Sim_i(A_i)$, and r is uniformly random.

4. PROTOCOL

4.1 Participants

Given a mobile device and a web or application server who wish to jointly compute a function, there are three participating parties in the Whitewash protocol:

- **SERVER**: "SERVER" refers to the web or application server participating in the joint computation. She is assumed to have large computational resources and is responsible for evaluating the garbled circuits.
- **MOBILE**: "MOBILE" refers to the mobile device participating in the joint computation. He is assumed to have limited processing power, memory, and communication bandwidth. MOBILE is tasked with garbling the circuit to be evaluated by SERVER.
- **CLOUD**: The outsourcing party "CLOUD" is responsible for relieving MOBILE of the majority of his computational load, but is not trusted with knowing either party's input to or output from the joint computation.

4.2 Protocol

Common Inputs: Security parameters k (key length) and σ (the number of circuits generated for the cut-&-choose); a commitment scheme $com(x; c)$ with committed value x and commitment key c; and a function $f(x, y)$.
Private Inputs: MOBILE inputs x and SERVER inputs y.
Outputs: Two outputs f_s, f_m for SERVER and MOBILE, respectively.

Phase 1: Pre-computation

1. **Preparing inputs:** MOBILE randomly generates $r \in \{0,1\}^{2k+\log(k)}$ as his input to the 2-universal circuit. He also generates $e \in \{0,1\}^{|f_m|}$ as a one-time pad for his output. SERVER computes her k-probe-resistant matrix $\mathbf{M}$ and $\overline{y}$ such that $\mathbf{M} \cdot \overline{y} = y$. MOBILE'S input to the circuit will be $\overline{x} = x\|e\|r$ and SERVER'S input will be $\overline{y}$. We denote the set of indices $[m_s] = \{1, \cdots, |\overline{y}|\}$ and $[m_m] = \{1, \cdots, |\overline{x}|\}$.
2. **Preparing circuit randomness:** MOBILE generates random seeds $\{\rho^{(j)}\}_{j\in[\sigma]}$ for generating the circuits and sends them to CLOUD.

Phase 2: Input commitments

1. **Committing to MOBILE'S inputs:** For each circuit $j \in [\sigma]$, input bit $i \in [m_m]$, and $b \in \{0,1\}$ MOBILE uses $\rho^{(j)}$ to generate commitment keys $\theta^{(j)}_{i,b}$. Using the same random seeds, these keys will later be generated by CLOUD to commit to the input wire labels corresponding to MOBILE'S input. MOBILE then commits to his own inputs as $\{\Gamma^{(j)}\}_{j\in[\sigma]}$ as:

$$\Gamma^{(j)} = \{com(\theta^{(j)}_{i,\overline{x}_i}; \gamma^{(j)}_i)\}_{i\in[m_m]}$$

using independently generated random commitment keys $\gamma^{(j)}_i$. MOBILE sends $\{\Gamma^{(j)}\}_{j\in[\sigma]}$ to SERVER and the commitment keys $\{\gamma^{(j)}_i\}_{i\in[m_m],j\in[\sigma]}$ to CLOUD.
2. **Committing to CLOUD'S inputs:** To allow for a fair release of the outputs, CLOUD inputs one-time pads to blind both parties' outputs. CLOUD randomly generates $p_s \in \{0,1\}^{|f_s|}$ and $p_m \in \{0,1\}^{|f_m|}$, as well as $r_c \in \{0,1\}^{2k+\log(k)}$ as its input to the 2-universal circuit. We denote CLOUD'S input as $z = p_s\|p_m\|r_c$, and the indices of CLOUD'S input wires as $[m_c] = \{1, \cdots, |z|\}$.
For each circuit $j \in [\sigma]$ and input bit $i \in [m_c]$, CLOUD uses $\{\rho^{(j)}\}_{j\in[\sigma]}$ to generates the garbled input wire keys $(K^{(j)}_{i,0}, K^{(j)}_{i,1}, \pi^{(j)}_i)$, where $K^{(j)}_{i,0}, K^{(j)}_{i,1} \in \{0,1\}^k$ and the permutation bit $\pi^{(j)}_i \in \{0,1\}$. To locate the correct key for bit b on input wire w_i of circuit j, we designate the label $W^{(j)}_{i,b} = (K^{(j)}_{i,b}, b \oplus \pi^{(j)}_i)$.
Let $\{w_{m_s+i}\}_{i\in[m_c]}$ be the input wires for CLOUD. CLOUD then commits to the label pairs for its input wires as $\{\Psi^{(j)}\}_{j\in[\sigma]}$,

where

$$\Psi^{(j)} = \{com(W^{(j)}_{m_s+i,0\oplus\pi^{(j)}_i}; \psi^{(j)}_{i,0\oplus\pi^{(j)}_i}), \\ com(W^{(j)}_{m_s+i,1\oplus\pi^{(j)}_i}; \psi^{(j)}_{i,1\oplus\pi^{(j)}_i})\}_{i\in[m_c]}$$

using commitment keys $\psi^{(j)}_{i,b}$ generated with the random seed $\rho^{(j)}$. CLOUD then commits to its inputs as $\{\Xi^{(j)}\}_{j\in[\sigma]}$ as:

$$\Xi^{(j)} = \{com(\psi^{(j)}_{i,z_i}; \xi^{(j)}_i)\}_{i\in[m_c]}$$

using independently generated random commitment keys. CLOUD sends $\{\Psi^{(j)}\}_{j\in[\sigma]}$ and $\{\Xi^{(j)}\}_{j\in[\sigma]}$ to SERVER.

Phase 3: Circuit construction

1. **Constructing the objective circuit:** SERVER sends $\mathbf{M}$ to CLOUD, then SERVER and CLOUD run a coin flipping protocol to randomly determine the 2-universal hash matrix $\mathbf{H} \in \{0,1\}^{k\times m_m}$. These two matrices can be used to generate the new circuit C that computes the function $g : (\overline{x}, \overline{y}) \rightarrow (\perp, (h_m, h_c, c_s, c_m))$, where $h_m = \mathbf{H}\cdot\overline{x}, h_c = \mathbf{H}\cdot z, g_m = f_s(x, \mathbf{M}\cdot\overline{y}), c_m = g_m \oplus e \oplus p_m, g_s = f_s(x, \mathbf{M}\cdot\overline{y})$, and $c_s = g_s \oplus p_s$. MOBILE will need the values $h_c \| c_m$ to recover his output. We denote the set of indices corresponding to these values as $O_m = \{1, \cdots, |h_c| + |c_m|\}$.
2. **Committing to input and output wire label pairs:** Using the same method as in Phase 2, CLOUD uses $\{\rho^{(j)}\}_{j\in[\sigma]}$ to generate the input wire keys for both SERVER and MOBILE'S input as well as the output wire keys for MOBILE'S output (these output keys must be committed for the witness indistinguishable proof of MOBILE'S output correctness). Let $\{w_i\}_{i\in[m_m]}$ be the input wires for MOBILE, $\{w_{m_m+i}\}_{i\in[m_s]}$ be the input wires for SERVER, and $\{w_i\}_{i\in O_m}$. CLOUD then commits to the label pairs in MOBILE'S input, SERVER'S input, and MOBILE'S output as $\{\Theta^{(j)}, \Omega^{(j)}, \Phi^{(j)}\}_{j\in[\sigma]}$, where

$$\Theta^{(j)} = \{com(W^{(j)}_{i,0\oplus\pi^{(j)}_i}; \theta^{(j)}_{i,0\oplus\pi^{(j)}_i}), \\ com(W^{(j)}_{i,1\oplus\pi^{(j)}_i}; \theta^{(j)}_{i,1\oplus\pi^{(j)}_i})\}_{i\in[m_m]}$$

$$\Omega^{(j)} = \{com(W^{(j)}_{m_m+i,0}; \omega^{(j)}_i), com(W^{(j)}_{m_m+i,1}; \omega^{(j)}_i)\}_{i\in[m_s]}$$

$$\Phi^{(j)} = \{com(W^{(j)}_{i,0}; \phi^{(j)}_i), com(W^{(j)}_{i,1}; \phi^{(j)}_i)\}_{i\in O_m}$$

using commitment keys generated with the random seed $\rho^{(j)}$. CLOUD then sends these commitments to SERVER.

Phase 4: Oblivious transfers (OT)

1. **Oblivious transfers:** CLOUD and SERVER execute m_s input oblivious transfers and σ circuit oblivious transfers as follows:
   (a) **Input:** For each $i \in [m_s]$, both parties run a $\binom{2}{1}$-OT where CLOUD inputs

   $$\left(\{(W^{(j)}_{m_m+i,0}; \omega^{(j)}_i)\}_{j\in[\sigma]}, \{(W^{(j)}_{m_m+i,1}; \omega^{(j)}_i)\}_{j\in[\sigma]}\right)$$

   while SERVER inputs $\overline{y}_i$. Once SERVER receives all of her garbled input wire labels, she uses the decommitment keys obtained in the OTs to check the committed wire values in $\{\Omega^{(j)}\}_{j\in[\sigma]}$. If any of the labels received in the OT do not match the committed wire labels, SERVER terminates the protocol.
   (b) **Circuit:** SERVER selects a set of circuits to be evaluated $S \subset [\sigma]$ such that $|S| = \frac{2\sigma}{5}$, as in shelat and Shen's protocol [40]. She represents this set with a bit string $s \in \{0,1\}^\sigma$ such that the jth bit $s_j = 1$ if $j \in S$ and $s_j = 0$ otherwise. SERVER and CLOUD perform σ $\binom{2}{1}$-OTs where, for every $j \in [\sigma]$, CLOUD inputs $(\rho^{(j)}, (\{\gamma^{(j)}_i\}_{i\in[m_m]} \| \{\Xi^{(j)}_i\}_{i\in[m_c]}))$, while SERVER inputs s_j. This allows SERVER to learn either the randomness used to generate the check circuits or MOBILE and CLOUD'S inputs for the evaluation circuits without CLOUD knowing which circuits are being checked or evaluated.

Phase 5: Evaluation

1. **Circuit evaluation:** Using $\rho^{(j)}$, CLOUD garbles the objective circuit C as $G(C)^{(j)}$ for all $j \in [\sigma]$ and pipelines these circuits to SERVER using Huang's technique [16]. Depending on whether the circuit is a check circuit or an evaluation circuit, SERVER performs one of two actions:
   (a) **Check:** For each $j \in [\sigma]\backslash S$, SERVER checks to see if $\rho^{(j)}$ can correctly regenerate the committed wire values $\{\Theta^{(j)}, \Omega^{(j)}, \Phi^{(j)}, \Psi^{(j)}\}$ and the circuit $G(C)^{(j)}$.
   (b) **Evaluate:** For each $j \in S$, SERVER checks that she can correctly decommit MOBILE'S input by recovering half of $\Theta^{(j)}$ from the keys committed in $\Gamma^{(j)}$. She does the same for CLOUD'S input, recovering half of $\Psi^{(j)}$ from the keys committed in $\Xi^{(j)}$

   If any of the above checks fail, SERVER aborts the protocol. Otherwise, she evaluates the circuits $\{G(C)^{(j)}\}_{j\in[\sigma]\backslash S}$. Each circuit outputs the values $(h^{(j)}_m, h^{(j)}_c, c^{(j)}_s, c^{(j)}_m)$ for $j \in [\sigma]\backslash S$.
2. **Majority output selection and consistency check:** Let (h_m, h_c, c_s, c_m) be the output of the majority of the evaluated circuits. If no majority value exists, SERVER aborts the protocol. Otherwise, she checks that $h^{(j)}_m = h_m$ and $h^{(j)}_c = h_c$ for all $j \in [\sigma]\backslash S$. If any of MOBILE or CLOUD'S hashed input values do not match, SERVER aborts the protocol.

Phase 6: Output proof and release

1. **Proof of output authenticity:** SERVER and MOBILE perform the proof of output authenticity from shelat and Shen's protocol [41] using the commitments to MOBILE'S output wires $\{\Phi^{(j)}\}_{j\in[\sigma]\backslash S}$ and the values $h_c \| c_m$.
2. **Output release:** CLOUD simultaneously releases the input one-time pads p_s and p_m to SERVER and MOBILE. SERVER and MOBILE then hash the pads and check to see if the hash values output by the circuit $h_c = H \cdot p_s \| p_m$. If the hashes do not match, SERVER and MOBILE abort the protocol. Otherwise, SERVER receives $c_s \oplus p_s$ as her output and MOBILE receives $c_m \oplus p_m \oplus e$ as his output.

Protocol	SYM	GROUP	OT	CT
CMTB	$\|x\|$	$\frac{2\sigma}{5}(\|y\|+1)$	k	yes
Salus	$\frac{2\sigma}{5}(\|x\|+\|y\|+\|f(x,y)\|)$	-	-	yes
WW	$\sigma(\|x\|+\frac{2}{5}\|f_m(x,y)\|)$	-	-	no

Table 1: Operations required on the mobile device by three outsourcing protocols. Here, SYM is the symmetric cryptographic operations, GROUP is the group algebraic operations, OT is the oblivious transfers, and CT is whether the protocol requires a coin toss. Recall that k is the security parameter, σ is the number of circuits generated, x is the mobile device's input, and y is the application server's input.

5. COMPARISON WITH PREVIOUS OUTSOURCING PROTOCOLS

In this section, we compare the asymptotic complexity and security guarantees of the Whitewash protocol to two previous outsourcing techniques: the protocol developed by Carter et al. [8], which we call "CMTB" for the remainder of this work, and the Salus framework developed by Kamara et al. [23]. We refer to our

Whitewash protocol as WW.

When examining the complexity of each protocol, recall that one of our main goals is to optimize the efficiency on the mobile device. Thus, we examine the number of operations each protocol requires on the mobile device itself. When compared to the underlying shelat-Shen protocol, Whitewash adds extra input values from the Cloud, but does not add any steps to the computation that increase the complexity of operations performed on the application server or the Cloud. Thus, for a discussion of the application server and Cloud protocol complexity, we refer the reader to the original work by shelat and Shen [41].

5.1 Comparison to CMTB

The underlying two-party computation protocols of Whitewash and CMTB follow similar structures in terms of the security checks that are performed. However, Kreuter, shelat, and Shen's (KSS) protocol [28], which underlies CMTB, uses a number of algebraic operations to perform input consistency checks and output proofs of consistency. The protocol developed by shelat and Shen [41], which underlies Whitewash, removes these expensive cryptographic primitives in favor of constructions that use only efficient, symmetric-key operations. In addition to the improvements to the underlying protocol, Whitewash outsources the generation side of two-party computation, while CMTB outsources the evaluation side. In CMTB, since neither the mobile device or the Cloud could garble inputs before computation, a specially designed Outsourced Oblivious Transfer (OOT) protocol is necessary to deliver the mobile device's inputs to the evaluating Cloud in a secure, privacy-preserving manner. By swapping roles in the Whitewash protocol, we allow the mobile device to garble its own inputs, removing the need for an OT protocol to be performed from the mobile device. While Whitewash still requires OTs between the Cloud and the evaluating party, these operations can be parallelized, while the OOT protocol acts as a non-parallelizable bottleneck in computation.

5.1.1 Asymptotic Complexity

Table 1 shows this complexity for both Whitewash and CMTB. Note that for the mobile device, Whitewash requires significantly more symmetric key operations for garbling its own input and verifying the correctness of its output. By contrast, the OOT protocol in CMTB requires very few symmetric key operations, but requires several instantiations of an oblivious transfer. In addition, CMTB requires that the mobile device check the application server's input consistency and verify the correctness of the output using algebraic operations (e.g., modular exponentiations and homomorphic operations). Considering the fact that modular exponentiation is significantly more costly than symmetric key operations, removing these public key operations from the phone is a significant efficiency improvement for Whitewash. We also note that CMTB requires a two-party fair coin toss at the mobile device, which is not required by Whitewash.

5.1.2 Security Guarantees

The removal of the OOT protocol in Whitewash not only increases its efficiency when compared to CMTB, it also allows for stronger security guarantees. In CMTB, security was only possible if none of the parties collude, since the mobile device possessed information that would allow the Cloud to recover both input wire labels for all of the mobile input wires after the OOT. If the mobile device and Cloud collude in the Whitewash protocol, it simply removes the guarantee of fair release and makes the protocol equivalent to the underlying two-party computation protocol. Thus, the only guarantee lost is that of fair release at the end of the protocol, since a colluding mobile device and Cloud may not release the one-time pad used to blind the evaluating party's output. We believe that this represents a more realistic security setting, since the mobile device is paying for the assistance of the Cloud and may collude.

5.2 Comparison to Salus

When considering the operations performed on the mobile device, the Salus protocol and the Whitewash protocol both make the mobile device responsible for generating circuit randomness and garbling its own inputs. However, the Whitewash protocol requires an added proof of output consistency that is not included in Salus. While this proof adds some complexity to the protocol, it allows Whitewash to handle functions where both parties get different output values, while Salus is designed to handle functions with a single, shared output value. In addition, the Whitewash protocol outsources the generation of the garbled circuit, while the malicious secure Salus protocol outsources the evaluation. By swapping the roles of the outsourced task and adding in consistency checks at the evaluating party, the Whitewash protocol guarantees security in a stronger adversarial model.

5.2.1 Asymptotic Complexity

Both the Whitewash and Salus protocols use only efficient, symmetric key operations, but there is a slight tradeoff in the number of operations required (Table 1). Salus only requires operations for the $\frac{2\sigma}{5}$ evaluated circuits, but requires those operations for each bit of both party's inputs and the shared output. By contrast, Whitewash requires that the mobile device's input be committed for all σ circuits generated, but then only requires correctness proof of the output wires on the $\frac{2\sigma}{5}$ evaluated circuits. When the application server's input is significantly longer than the mobile device's, this will cause the Salus protocol to be less efficient than Whitewash. However, in the average case where both inputs are approximately the same length, this will mean that Whitewash requires more operations. This small tradeoff in efficiency is justified by the fact that Whitewash provides security in a stronger adversarial model than Salus. We also note that Salus requires a two-party fair coin toss before the protocol begins, which is not required by Whitewash.

5.2.2 Security Guarantees

The Salus protocol provides equivalent security guarantees to CMTB, guaranteeing security when none of the parties are colluding. This is a result of outsourcing the evaluation to the Cloud while allowing the mobile device to generate circuit randomness. If the mobile device colludes with the Cloud, they can trivially recover all of the other party's inputs. By outsourcing the generation of the garbled circuit and adding in additional consistency checks at the evaluating party, Whitewash guarantees security under this type of collusion. As stated above, the only guarantee lost is that of fair output release, which ultimately reduces Whitewash to the security of the underlying two-party computation protocol.

6. PERFORMANCE EVALUATION

Our protocol significantly expands upon the implementations of the PCF garbled circuit generation technique [27] and shelat and Shen's garbled circuit evaluation protocol [41]. For experimental comparison to previous protocols, we acquired the code implementation of the outsourcing protocol by Carter et al. [8] directly from the authors, as well as an Android port of the two-party garbled circuit protocol developed by Kreuter, shelat, and Shen [28]. We would like to thank the authors of [8, 27, 28, 41] for making their

Circuit	Input Size (Bits)	Total Gates		Non-XOR Gates	
		KSS	PCF	KSS	PCF
HAM (1600)	1,600	24,379	32,912	17,234	6,375
HAM (16384)	16,384	262,771	376,176	186,326	101,083
MAT (3x3)	288	424,748	92,961	263,511	27,369
MAT (5x5)	800	1,968,452	433,475	1,221,475	127,225
MAT (8x8)	2,048	8,067,458	1,782,656	5,006,656	522,304
MAT (16x16)	8,192	64,570,969	14,308,864	40,076,631	4,186,368
DIJK 10	112/1,040	259,232	530,354	118,357	291,490
DIJK 20	192/2,080	1,653,380	2,171,088	757,197	1,192,704
DIJK 50	432/5,200	22,109,330	13,741,514	10,170,407	7,549,370
RSA-256	256/512	934,092,960	673,105,990	602,006,981	235,925,023

Table 2: Input size and circuit size for all test circuits evaluated.

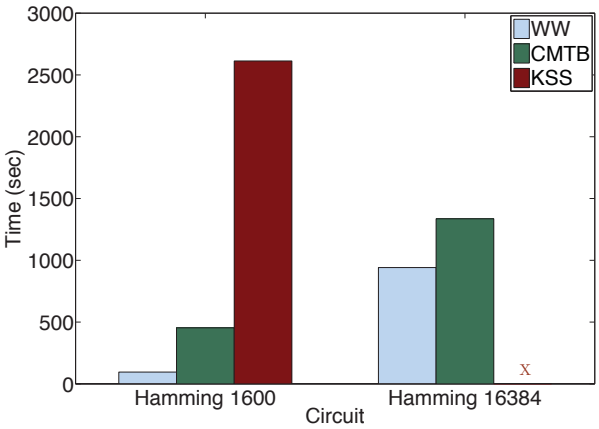


Figure 2: Execution time (ms) for Hamming Distance with input sizes of 1,600 and 16,384 bits for $\sigma = 256$ (note: log scale). Note that without outsourcing, only very small inputs can be computed over. Additionally, even for a large number of input bits, performing OTs on the servers still produces a faster execution time.

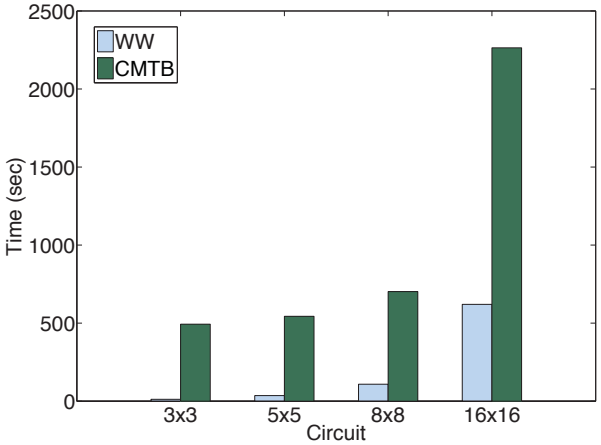


Figure 3: Execution time (ms) for the Matrix-Multiplication problem with input size varying between 3×3 matrices and 16×16 matrices for $\sigma = 256$ (note: log scale). This figure clearly shows that the oblivious transfers, consistency checks, and larger circuit representations of CMTB add up to a significant overhead as input size and gate count increase. By contrast, Whitewash requires less overhead and increases more slowly in execution time as gate counts and input size grow.

code available and for assisting us with our evaluation[2].

6.1 Test Environment

For evaluating our test circuits, we perform our experiments with a single server performing the role of Cloud and Application server, communicating with a mobile device over an 802.11g wireless connection. The server is equipped with 64 cores and 1TB of memory, and we partition the work between cores into parallel processing nodes using MPI. The mobile device used is a Samsung Galaxy Nexus with a 1.2 GHz dual-core ARM Cortex-A9 processor and 1 GB of RAM, running Android 4.0.

The large input sizes examined in the Hamming Distance trials required us to use a different testbed. For inputs as large as 16,384 bits, the phone provided by the above computing facility would overheat and fail to complete computation. Because the gate counts for Hamming Distance are significantly smaller than the other test circuits, we were able to run these experiments on a local testbed. We used two servers with Dual Intel Xeon E5620 processors, each with 4 hyper-threaded cores at 2.4 GHz each for the Cloud and the Application server. Each server is running the Linux kernel version 2.6, and is connected by a VLAN through a 1 Gbps switch. Our mobile device is a Samsung Galaxy Note II with a 1.6 GHz quad-core processor with Arm Cortex A9 cores and 2 GB of RAM, running the Android operating system at version 4.1. The phone connects to the two servers through a Linksys 802.11g wireless router with a maximum data rate of 54 Mbps. While this test environment represents optimistic connection speeds that may not always be available in practice, it allows us to consider the performance of the protocol without interference from variable network conditions, and mirrors the test environments used in previous work [8, 28, 41]. For all experiments except RSA-256, we take the average execution time over ten test runs, with a confidence interval of 95%. For RSA-256, we ran 3 executions.

6.2 Experimental Circuits

To evaluate the performance of our protocol, we run tests over the following functions. We selected the following test circuits because they exercise a range of the two major variables that affect the speed of garbled circuit protocols: input size and gate counts.

[2]We contacted the authors of the Salus protocol [23] in an attempt to acquire their framework to compare the actual performance of their scheme with ours. Because they were unable to release their code, no sound comparison to their work beyond an asymptotic analysis was possible. Our code is available at `http://www.foryourphoneonly.org`.

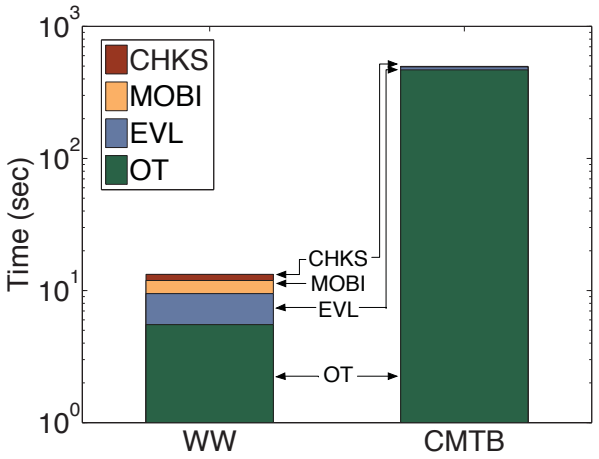


Figure 4: Microbenchmarking execution times (ms) for Whitewash and CMTB over the Matrix-Multiplication problem. We denote the total time spent in computation for Whitewash as "MOBI". Since the mobile device is linked with "CHKS" and "OT" in CMTB, we do not separate out the mobile time for that protocol. Notice the dominating amount of time required to perform oblivious transfers. Moving these operations off the mobile device removes a significant computation bottleneck.

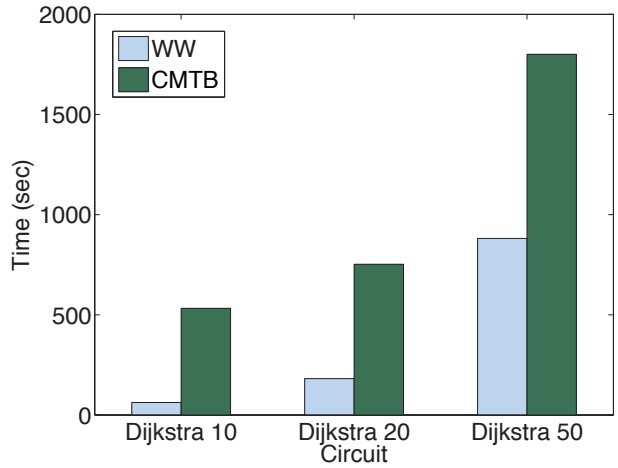


Figure 5: Execution time (s) for Dijkstra's algorithm with input sizes of $10, 20,$ and 50 node graphs for $\sigma = 256$. This figure shows that the Whitewash protocol allows for computation that was only feasible to be executed in a close to practically useful time frame.

In addition, these programs are becoming somewhat standard test applications, having been used as benchmarks in a large amount of the related literature [8, 27, 28, 41]. All of the programs are implemented with the algorithms used by Kreuter et al. [27] except for Dijkstra's algorithm, which matches the implementation used by Carter et al. [8]:

- **Hamming Distance (HAM):** The Hamming Distance circuit accepts binary string inputs from both parties and outputs the number of locations at which those strings differ. This circuit demonstrates performance for a small number of gates over a wide range of input sizes. We consider input strings of length 1,600 bits and 16,384 bits.
- **Matrix Multiplication (MAT):** Matrix multiplication takes an $n \times n$ matrix of 32-bit integer entries from each party and outputs the result of multiplying the matrices together. This circuit demonstrates performance when both input size and gate count vary widely. We consider square matrix inputs where $n = 3, 5, 8,$ and 16.
- **Dijkstra's Algorithm (DIJK):** This version of Dijkstra's algorithm takes an undirected weighted graph with a grid structure and a maximum node degree of four from the first party, and a start and end node from the second party. The circuit outputs the shortest path from the start node to the end node to the second party, and nothing to the first. For an n node graph, the graph description from the first party requires $104n$ input bits, while the start and end node descriptions require $8n + 32$ bits. We consider graphs with $n = 10, 20,$ and 50 nodes. Due to an error in the PCF compiler, we were unable to compile a program for graphs larger than 50 nodes.
- **RSA Function (RSA):** The RSA function (i.e., modular exponentiation) accepts an RSA message from one party and an RSA public key from the other party and outputs the encryption of the input plaintext under the input public key. Specifically, one party inputs the modulus $n = pq$ for primes p and q, as well as the encryption key $e \in \mathbb{Z}_{\phi(n)}$. The other party inputs a message $x \in \mathbb{Z}_n^*$, and the circuit computes $x^e \pmod{n}$. This circuit demonstrates performance for small input sizes over very large gate counts. We consider the case where the input values $x, n,$ and e are 256 bits each. While these are not secure parameters in practice, the function itself provides a complex circuit that is scalable on input size and useful for benchmarking our protocol.

For each test circuit, we consider the time required to execute and the bandwidth overhead to the mobile device. Table 2 shows the input size and gate counts for each test circuit, showing the exact range of values tested for these two circuit variables.

6.3 Execution Time

In all experiments, the efficiency gains of removing oblivious transfers and public key operations are immediately apparent. To examine how Whitewash compares to generating garbled circuits directly on the mobile device, we considered Hamming Distance as a simple problem (Figure 2). Even with a relatively small gate count, garbling the circuit directly on the mobile device is only possible for the small input size of 1,600 bits. Whitewash is capable of executing this protocol in 96 seconds, while running the computation directly on the mobile device takes 2,613 seconds, representing a 96% performance improvement through our outsourcing scheme. For the very large input size of 16,384 bits, computation directly on the mobile device ceases to be possible. When comparing to CMTB, this circuit further illustrates the cost of oblivious transfers on the mobile device. Even with the significantly reduced number of OTs allowed by the OOT protocol in CMTB (80 OTs), performing 16,384 malicious secure oblivious transfers between two servers in Whitewash still runs 30% faster than CMTB.

The matrix-multiplication circuit provides a good overview of average-case garbled circuit performance, as it represents a large range of both gate counts and size of inputs. For the input size of a 3×3 matrix, the Whitewash protocol runs in an average of 12 seconds, while CMTB requires 493 seconds, representing a 98% improvement (see Figure 3). Upon inspecting the micro benchmarking breakdown of each protocol's execution in Figure 4, we observe a significant speedup simply by moving oblivious transfers off of the mobile device. Even though the number of OTs required by CMTB is essentially constant based on their application of the

Ishai OT extension, performing standard malicious secure oblivious transfers in parallel between the servers is much more efficient than requiring that the phone perform these costly operations. In addition, if we examine the amount of execution time where the phone participates in Whitewash, we see that the mobile device ("MOBI" in Figure 4), takes around 1 second, and is idle during the majority of computation. By contrast, both the OT and consistency check phases of CMTB require the mobile device to participate in a significant capacity, totaling to almost 8 minutes of the computation. Having the phone perform as little work as possible means that the Whitewash protocol performance is nearly equivalent to running the same computation between two server-class machines.

To examine the performance of Whitewash for a more practical application, we considered the Dijkstra's algorithm circuit used to implement privacy-preserving navigation by Carter et al. [8]. They point out that this application, which has uses from military convoys to industrial shipping routes, is a significant first step in providing privacy for the growing genre of location-based mobile applications. Unfortunately, the PCF compiler does not optimize the Dijkstra's circuit as well as the previous experimental programs, which is evident in Table 2. In the 10 and 20 node graphs, the PCF compiler even produces a *larger* circuit than the compiler used by KSS. However, despite evaluating larger circuits, the Whitewash protocol still outperforms CMTB in execution time, running 88%,76%, and 51% faster in the 10, 20, and 50 node cases respectively (shown in Figure 5). As circuit compilers continue to improve and produce smaller circuits, the performance gains of the Whitewash protocol will be even larger. In this experiment, we also noticed that because Whitewash evaluates and checks circuits simultaneously, it created contention for the network stack in our test server. In a truly distributed environment where each server node has dedicated network resources, the highly parallelizable structure of shelat and Shen's protocol would allow Whitewash to execute faster. Given that Whitewash can execute Dijkstra's algorithm obliviously on the order of minutes, it allows computation considered only feasible for previous schemes to be performed in a nearly practical execution time.

The previous experiments clearly show that outsourcing is necessary to run circuits of any practical size. For our final test circuit, we consider an extremely complex problem to demonstrate the ability of outsourcing protocols in the worst-case. The RSA-256 circuit evaluated by Kreuter et al. in [27] and shelat and Shen in [41] represents one of the largest garbled circuits ever evaluated by a malicious secure protocol. For the RSA-256 problem, Whitewash completed the computation in 515 minutes. CMTB was unable to complete one execution of the protocol. A large part of this efficiency improvement results from the underlying protocol of Whitewash, which uses only symmetric-key operations outside of the oblivious transfers between the servers. The reduced non-XOR gate counts and more compact circuit representation of the PCF compiler also contribute to this improvement. Ultimately, because Whitewash ensures that the phone participates minimally in the protocol, it no longer acts as a bottleneck on computation. We essentially reduce performance of our outsourcing protocol to that of the underlying two-party protocol, allowing this technique for outsourcing to benefit as more improvements are made in non-outsourced garbled circuit protocols. In addition, this minimal level of interactivity allows us to run these protocols with 256 circuits, equivalent to a security parameter of approximately 80-bit security, which is agreed by the research community to be an adequate security parameter. Finally, the phone is only active for a few seconds during this large computation, keeping its system resources free for other user applications (or to preserve battery power) while the servers complete the computation. *This shows that Whitewash is capable of evaluating the same circuits as the most efficient desktop-based garbled circuit protocols with a minimal overhead cost*. Exact execution times are shown in our technical report [7].

	Bandwidth (MB)			Reduction Over	
Circuit	WW	CMTB	KSS	CMTB	KSS
HAM (1600)	23.56	41.05	240.33	42.62%	90.20%
HAM (16384)	241.02	374.03	x	35.56%	x
MAT (3x3)	4.26	11.50	x	62.97%	x
MAT (5x5)	11.79	23.04	x	48.82%	x
MAT (8x8)	30.15	51.14	x	41.05%	x
MAT (16x16)	120.52	189.52	x	36.41%	x
DIJK 10	1.67	20.21	x	91.73%	x
DIJK 20	2.85	35.28	x	91.93%	x
DIJK 50	6.38	80.49	x	92.08%	x
RSA-256	3.97	x	x	x	x

Table 3: Bandwidth measures for all experiment circuits. Note that there is as much as a 84% reduction in bandwidth when using the Whitewash protocol.

6.4 Network Bandwidth

The Whitewash protocol not only improves the speed of execution when outsourcing garbled circuit computation, it also significantly reduces the amount of bandwidth required by the mobile device to participate in the computation. Table 3 shows the bandwidth used by the mobile device for each test circuit. In the best case, for Dijkstra's algorithm over 50 node graphs, we observed a 92% reduction in bandwidth between Whitewash and CMTB. This is a result of the mobile device not performing OTs and only sending relatively small symmetric-key values instead of algebraic elements for consistency checks. For all test circuits, we observed a small decrease in the amount of improvement between the two protocols as the input size increased. This is because the number of commitments sent by the phone in Whitewash increases as the size of the input grows, while CMTB performs a fixed number of OTs as the input size increases. However, the oblivious transfers still require a significant enough amount of bandwidth to make removing them the most efficient option. When comparing to not outsourcing garbled circuit generation, the cost of oblivious transfers and sending several copies of the garbled circuit to the evaluator quickly adds up to a significant bandwidth cost. For the smallest circuit evaluated, outsourcing the circuit garbling reduces the required amount of bandwidth by 90%. The importance of these bandwidth reductions is further highlighted when considering mobile power savings. With data transmission costing roughly 100 times as much power as computation on the same amount of data, any reduction in the bandwidth required by a protocol implies a critical improvement in practicality.

One challenge encountered during the implementation of the Whitewash protocol was the extensive use of hardware-specific functions used to implement commitment schemes in shelat and Shen's code. Rather than try to port this code over to Android, which would require significant development of hardware-specific libraries, we chose to implement the protocol in an equivalent manner by having the Cloud generate the part of the commitments which requires these functions and send them to the mobile device. The mobile device then finishes generating the commitments that match its input and forwards them to the evaluator. Our proofs of security remain valid even with this small protocol modification. Our preliminary implementation using instructions specific to the ARM architecture has shown that we could further reduce the measured bandwidth values by over 60%. With already significant bandwidth

reductions from previous outsourcing schemes, our protocol will see further improvements as mobile hardware begins to incorporate more machine-specific libraries.

7. CONCLUSION

With the increasingly pervasive and personal nature of mobile computing, garbled circuits provide a solution that preserves both privacy and application functionality. However, to make these computationally expensive protocols usable on mobile devices, secure outsourcing to the cloud is necessary. We develop a new scheme that eliminates the most costly operations, including oblivious transfers, from the mobile device. By requiring that the mobile device instead produce the randomness required for circuit generation, we significantly reduce the number of algebraic group operations and communication rounds for the mobile device. At the same time, we bolster the security guarantees against certain types of collusion, yielding a more secure protocol than any other in this space. Our performance evaluation shows average gains of 75% for execution time and 60% for bandwidth over the previous outsourcing protocol. These improvements allow large circuits representing practical applications to be computed efficiently from a mobile device. As a result, we show that garbled circuit protocols can be made nearly as efficient for mobile devices as they are for server-class machines.

Acknowledgments This material is based on research sponsored by DARPA under agreement number FA8750-11-2-0211. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. 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 DARPA or the U.S. Government.

References

[1] M. J. Atallah and K. B. Frikken. Securely outsourcing linear algebra computations. In *Proceedings of the ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, 2010.

[2] Y. Aumann. Security Against Covert Adversaries: Efficient Protocols for Realistic Adversaries. *Journal of Cryptology*, 18(3):554–343, 2010.

[3] D. Beaver. Server-assisted cryptography. In *Proceedings of the workshop on New security paradigms (NSPW)*, 1998.

[4] M. Bellare, V. T. Hoang, S. Keelveedhi, and P. Rogaway. Efficient garbling from a fixed-key blockcipher. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2013.

[5] J. Brickell and V. Shmatikov. Privacy-preserving graph algorithms in the semi-honest model. In *Proceedings of the international conference on Theory and Application of Cryptology and Information Security*, 2005.

[6] H. Carter, C. Amrutkar, I. Dacosta, and P. Traynor. For your phone only: custom protocols for efficient secure function evaluation on mobile devices. *Journal of Security and Communication Networks (SCN)*, 7(7):1165–1176, 2014.

[7] H. Carter, C. Lever, and P. Traynor. Whitewash: Outsourcing garbled circuit generation for mobile devices. Cryptology ePrint Archive, Report 2014/224, 2014. `http://eprint.iacr.org/`.

[8] H. Carter, B. Mood, P. Traynor, and K. Butler. Secure Outsourced Garbled Circuit Evaluation for Mobile Devices. In *Proceedings of the USENIX Security Symposium*, 2013.

[9] comScore. comScore Reports February 2013 U.S. Smartphone Subscriber Market Share. `http://www.comscore.com/Insights/Press_Releases/2013/4/comScore_Reports_February_2013_U.S._Smartphone_Subscriber_Market_Share`, 2013.

[10] I. Damgård, M. Geisler, and J. B. Nielsen. From passive to covert security at low cost. In *Proceedings of the 7th international conference on Theory of Cryptography*, 2010.

[11] I. Damgard, V. Pastro, N. Smart, and S. Zakarias. Multiparty computation from somewhat homomorphic encryption. In *Proceedings of the Annual International Cryptology Conference on Advances in Cryptology*, 2012.

[12] C. Gentry, S. Halevi, and N. P. Smart. Homomorphic evaluation of the AES circuit. In *Advances in Cryptology - CRYPTO*, 2012.

[13] S. D. Gordon, J. Katz, V. Kolesnikov, A.-l. B. Labs, F. Krell, and M. Raykova. Secure Two-Party Computation in Sublinear (Amortized) Time. In *Proceedings of the ACM conference on Computer and communications security (CCS)*, 2012.

[14] M. Green, S. Hohenberger, and B. Waters. Outsourcing the Decryption of ABE Ciphertexts. In *Proceedings of the USENIX Security Symposium*, 2011.

[15] C. Hazay and Y. Lindell. Efficient Protocols for Set Intersection and Pattern Matching with Security Against Malicious and Covert Adversaries. *Journal of Cryptology*, 23(3):422–456, 2008.

[16] Y. Huang, D. Evans, J. Katz, and L. Malka. Faster Secure Two-Party Computation Using Garbled Circuits. In *Proceedings of the USENIX Security Symposium*, 2011.

[17] Y. Huang, J. Katz, and D. Evans. Quid-pro-quo-tocols: Strengthening semi-honest protocols with dual execution. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2012.

[18] Y. Huang, J. Katz, and D. Evans. Efficient secure two-party computation using symmetric cut-and-choose. In *Advances in Cryptology–CRYPTO*, 2013.

[19] N. Hustead, S. Myers, abhi shelat, and P. Grubbs. GPU and CPU parallelization of honest-but-curious secure two-party computation. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, 2013.

[20] A. Iliev and S. W. Smith. Small, Stupid, and Scalable: Secure Computing with Faerieplay. In *The ACM Workshop on Scalable Trusted Computing*, 2010.

[21] S. Jha, L. Kruger, and V. Shmatikov. Towards practical privacy for genomic computation. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2008.

[22] S. Kamara, P. Mohassel, and M. Raykova. Outsourcing multi-party computation. Cryptology ePrint Archive, Report 2011/272, 2011. `http://eprint.iacr.org/`.

[23] S. Kamara, P. Mohassel, and B. Riva. Salus: A system for server-aided secure function evaluation. In *Proceedings of the ACM conference on Computer and communications security (CCS)*, 2012.

[24] F. Kerschbaum. Collusion-resistant outsourcing of private set intersection. In *Proceedings of the ACM Symposium on Applied Computing*, 2012.

[25] M. Kiraz and B. Schoenmakers. A Protocol Issue for The Malicious Case of Yao's Garbled Circuit Construction. In *Proceedings of the Symposium on Information Theory in the Benelux*, 2006.

[26] M. S. Kiraz. *Secure and Fair Two-Party Computation*. PhD thesis, Technische Universiteit Eindhoven, 2008.

[27] B. Kreuter, a. shelat, B. Mood, and K. Butler. PCF: A portable circuit format for scalable two-party secure computation. In *Proceedings of the USENIX Security Symposium*, 2013.

[28] B. Kreuter, a. shelat, and C. Shen. Billion-Gate Secure Computation with Malicious Adversaries. In *Proceedings of the USENIX Security Symposium*, 2012.

[29] L. Kruger, S. Jha, E.-J. Goh, and D. Boneh. Secure Function Evaluation with Ordered Binary Decision Diagrams. In *Proceedings of the ACM conference on Computer and communications security (CCS)*, 2006.

[30] Y. Lindell. Fast cut-and-choose based protocols for malicious and covert adversaries. In *Advances in Cryptology–CRYPTO*, 2013.

[31] Y. Lindell and B. Pinkas. Privacy preserving data mining. In *Proceedings of the Annual International Cryptology Conference on Advances in Cryptology*, 2000.

[32] Y. Lindell and B. Pinkas. An efficient protocol for secure two-party computation in the presence of malicious adversaries. In *Proceedings of the annual international conference on Advances in Cryptology*, 2007.

[33] Y. Lindell and B. Pinkas. Secure two-party computation via cut-and-choose oblivious transfer. In *Proceedings of the conference on Theory of cryptography*, 2011.

[34] L. Malka. Vmcrypt: modular software architecture for scalable secure computation. In *Proceedings of the 18th ACM conference on Computer and communications security*, 2011.

[35] D. Malkhi, N. Nisan, B. Pinkas, and Y. Sella. Fairplay–a secure two-party computation system. In *Proceedings of the USENIX Security Symposium*, 2004.

[36] A. Miyaji and M. S. Rahman. Privacy-preserving data mining in presence of covert adversaries. In *Proceedings of the international conference on Advanced data mining and applications: Part I*, 2010.

[37] P. Mohassel and M. Franklin. Efficiency tradeoffs for malicious two-party computation. In *Proceedings of the Public Key Cryptography conference*, 2006.

[38] B. Mood, L. Letaw, and K. Butler. Memory-efficient garbled circuit generation for mobile devices. In *Proceedings of the IFCA International Conference on Financial Cryptography and Data Security (FC)*, 2012.

[39] N. Nipane, I. Dacosta, and P. Traynor. "Mix-In-Place" anonymous networking using secure function evaluation. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, 2011.

[40] a. shelat and C.-H. Shen. Two-output secure computation with malicious adversaries. In *Proceedings of the Annual international conference on Theory and applications of cryptographic techniques*, 2011.

[41] a. shelat and C.-H. Shen. Fast two-party secure computation with minimal assumptions. In *Proceedings of the ACM conference on Computer and communications security (CCS)*, 2013.

[42] D. Talbot. Security in the ether. `http://www.technologyreview.com/featuredstory/416804/security-in-the-ether/`, 2009.

[43] A. C. Yao. Protocols for secure computations. In *Proceedings of the Annual Symposium on Foundations of Computer Science*, 1982.

Using Automatic Speech Recognition for Attacking Acoustic CAPTCHAs: The Trade-off between Usability and Security

Hendrik Meutzner, Viet-Hung Nguyen, Thorsten Holz, Dorothea Kolossa
Horst Görtz Institute for IT-Security (HGI), Ruhr-University Bochum
{hendrik.meutzner, viet.nguyen-c7k, thorsten.holz, dorothea.kolossa}@rub.de

ABSTRACT

A common method to prevent automated abuses of Internet services is utilizing challenge-response tests that distinguish human users from machines. These tests are known as CAPTCHAs (*Completely Automated Public Turing Tests to Tell Computers and Humans Apart*) and should represent a task that is easy to solve for humans, but difficult for fraudulent programs. To enable access for visually impaired people, an acoustic CAPTCHA is typically provided in addition to the better-known visual CAPTCHAs. Recent security studies show that most acoustic CAPTCHAs, albeit difficult to solve for humans, can be broken via machine learning.

In this work, we suggest using speech recognition rather than generic classification methods for better analyzing the security of acoustic CAPTCHAs. We show that our attack based on an automatic speech recognition system can successfully defeat reCAPTCHA with a significantly higher success rate than reported in previous studies.

A major difficulty in designing CAPTCHAs arises from the trade-off between human usability and robustness against automated attacks. We present and analyze an alternative CAPTCHA design that exploits specific capabilities of the human auditory system, i.e., auditory streaming and tolerance to reverberation. Since state-of-the-art speech recognition technology still does not provide these capabilities, the resulting CAPTCHA is hard to solve automatically. A detailed analysis of the proposed CAPTCHA shows a far better trade-off between usability and security than the current quasi-standard approach of reCAPTCHA.

1. INTRODUCTION

In order to limit or even prevent automated abuse of online services (e.g., automated account creation or crawling), it is necessary to distinguish human users from programs. A common approach is to rely on a so-called *Completely Automated Public Turing Test to Tell Computers and Humans Apart* (CAPTCHA) that should be easy to solve by humans, but difficult to break by machines [5, 29, 30]. Such challenge-response tests are predominantly based on some kind of obscured graphic such as a highly distorted text that has to be recognized by the user. Some tests also require the user to solve simple mathematical equations [7] or to identify specific objects within an image [20]. The security properties of such visual CAPTCHAs have been discussed several times within the last decade [14, 31, 32] and various security improvements have been proposed [6, 10, 24].

To enable access for visually impaired users and to be applicable for non-graphical devices, most Internet services also provide an acoustic verification scheme in addition to visual CAPTCHAs. Up to now, acoustic CAPTCHAs have received less scientific attention compared to the visual ones and existing studies have demonstrated that most acoustic CAPTCHAs can be easily broken [3, 4, 22, 26]. Thus, a major problem arises in practice from the fact that many popular Internet services make use of acoustic CAPTCHAs that are possibly vulnerable, rendering any supplementary image-based CAPTCHA irrelevant.

We consider an acoustic CAPTCHA to be broken if an algorithm is able to find the correct sequence of words in the audio signal, which we will refer to as *transcription* later on. Therefore, the crucial measure for the security strength of a CAPTCHA is given by the success rate for transcribing the signals in an automated manner. Here, the definition of a plausible value for the maximum allowed success rate is highly controversial, as one has to consider a trade-off between human usability and robustness against automated attacks. Some authors require the success rate of an automatic solver to be less than 0.01 % in order to render the CAPTCHA secure [5], whereas other studies refer to a maximum allowed success rate of 1 % [3] or 5 % [26]. However, the definition highly depends on the underlying threat model that arises from several properties and resources of the attacker. Here, the degree of theoretical knowledge about signal processing and machine learning as well as the number of available IP addresses will influence the success of an automated attack significantly. For our purposes, we assume that all potential vulnerabilities are induced by the CAPTCHA itself (i.e., insecure implementations are not within the scope of this work).

Preliminaries and Prior Work.

Acoustic CAPTCHAs are commonly based on a sequence of words where the underlying lexicon is often limited only to digits between zero and nine, which can be beneficial for non-native speakers or just to keep the user instructions sim-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.

ACSAC '14 New Orleans, Louisiana, USA
Copyright 2014 ACM 978-1-4503-3005-3/14/12 ...$15.00.
http://dx.doi.org/10.1145/2664243.2664262

ple (e.g., "please type in every digit you hear"). However, the security aspect of such small-vocabulary-based CAPTCHAs becomes highly questionable as state-of-the-art machine learning methods are fairly advanced, especially when the number of individual words to be recognized is low.

Previous studies have performed a security analysis of available acoustic CAPTCHAs by using automated solvers. Bursztein et al. [3] and Tam et al. [26] utilize a two-stage approach by first segmenting the CAPTCHA into individual word segments and then finding the respective word transcriptions by using a generic classification algorithm. With such an approach, Bursztein et al. evaluated the strength of six acoustic CAPTCHAs and found that they can break all but one scheme: the success rate was between 41 % and 89 % for five schemes, only reCAPTCHA [12] (as of 2010) withstood their attack since the success rate was only 1.5 %. Sano et al. [22] demonstrate that a newer version of reCAPTCHA (as of April 2013) can be broken with a success rate of 52 % by using *automatic speech recognition* (ASR).

Up to now, only a limited number of studies explored secure design strategies for acoustic CAPTCHAs. Recent studies analyze the effectiveness of artificial signal distortion [1, 16] or suggest to use speech signals that exhibit a poor overall quality like old radio broadcast recordings [25]. Schlaikjer [23] proposed a CAPTCHA where the task is to recognize open-domain speech[1].

As discussed above, attacking CAPTCHAs in an automated fashion generally represents a machine learning problem that can be tackled by various methods. Hence, it is not only practical but also indispensable to exploit a broad range of existing weaknesses in current machine learning techniques when designing secure acoustic CAPTCHAs. Obviously, the introduction of artificial distortions into a speech signal can lower the performance of an automated solver. The difficulty lies in distorting the signals such that the speech intelligibility is preserved.

Our Approach.

We consider automated attacks on acoustic CAPTCHAs as a speech recognition task and demonstrate how state-of-the-art ASR techniques can be utilized to accomplish it. A major benefit of using ASR arises from the fact that the temporal characteristics of speech are inherently considered and incorporated by the employed algorithms (cf. Sec. 2) as compared to a generic classification algorithm.

Bursztein et al. reported [4] that the ASR-based approach performed worse than the classification-based approach for attacking the eBay CAPTCHA. We assume that this finding is based on the authors using pre-trained acoustic models—freely available on the Internet—without adapting the models to the CAPTCHA signals.

In contrast to previous work [3, 26, 22], our approach is not based on a pre-segmentation of the CAPTCHA into individual parts. This is advantageous as we do not require any task-specific adaption of the segmentation stage, rendering the CAPTCHA solver more general.

The results indicate that our method clearly outperforms previous approaches based on segmentation and word recognition: we achieve a success rate of 19.19 % for the older version of reCAPTCHA analyzed by Bursztein et al. [3] (13× improvement). We can also break the current reCAPTCHA with a success rate of 62.8 % which is 11 % higher than the success rate reported for reCAPTCHA by Sano et al. [22]. In addition, we show that the human success rate for the current reCAPTCHA scheme is 38.4 % below that of our automated CAPTCHA solver, rendering the trade-off between usability and security fairly disadvantageous.

Furthermore, we systematically explore how CAPTCHAs can be improved upon to attain the original goal of being much more intelligible to humans than to machines. More specifically, we have generated CAPTCHAs by exploiting the auditory perception of humans in the design process. Our proposed CAPTCHA achieves a far better trade-off between usability and security as compared to the current reCAPTCHA scheme.

Our Contributions.

In summary, our contributions in this work are as follows:

- We demonstrate that speech recognition methods can yield considerable performance improvements for solving acoustic CAPTCHAs in comparison to a classification-based approach as it is used by Bursztein et al. [3]. To allow for a fair comparison, our analysis is based on a previous reCAPTCHA scheme from 2010.
- We analyze the current reCAPTCHA scheme and present the respective human recognition scores that were obtained from a listening test. In this context we discuss the usability of the CAPTCHA by comparing the performance of humans vs. machines.
- We propose alternative design strategies for creating more secure acoustic CAPTCHAs. These design strategies are motivated by the auditory perception of humans, where we exploit human abilities that current machine learning algorithms are lacking to a certain degree. We also conduct a security analysis of the proposed design strategies by using an automated solver and present the results of a listening experiment to examine human usability.

2. TECHNICAL BACKGROUND

State-of-the-art speech recognizers are usually based on *hidden Markov models* (HMMs) or *artificial neural networks* (ANNs) [11, 15, 18], which is advantageous compared to a generic classification-based approach. This is due to the fact that speech represents a slowly time-varying process, so it is reasonable to incorporate algorithms that can handle time series by nature.

Our attacks are conducted by using HMM-based speech recognition. For convenience, we make use of the hidden Markov model toolkit (HTK) [33], which represents a standard toolkit in the research community. For our approach, all model topologies, i.e., the number of states and mixtures and the allowed state transitions have been manually adjusted to be well suitable for the task and the ensuing model training has been done from scratch.

2.1 HMM-based Speech Recognition

In order to perform speech recognition, the time-domain representation (i.e., the waveform), of the input signal is first segmented into a series of overlapping frames. In practice,

[1]Open-domain speech covers complex scenarios (e.g., lectures and talks) and it is not limited to a specific task.

typical frame lengths are in the range of 20–25 ms as speech can be regarded to be short-time stationary within this interval. The overlap between consecutive frames is preferably 50–75 % to capture the temporal evolution of the waveform continuously. Then, a feature vector is computed for each frame of the input signal. Two prominent types of features for ASR are *Mel frequency cepstral coefficients* (MFCC) [19, 21] or *perceptual linear prediction* (PLP) coefficients [13]. These features are motivated by several properties of the auditory system and enable a compact representation of the speech frames, where PLP features are more suitable for suppressing speaker-dependent details. In order to incorporate signal dynamics, it is valuable to also include the 1st and 2nd-order derivatives of the feature coefficients into the final feature set. After feature extraction, the next step is to learn suitable segment models for representing the speech using the features and the label information of the training material. The segment models should represent the basic units of speech, e.g., phonemes[2], in large-vocabulary applications, but for small-vocabulary tasks, it is feasible and ultimately leads to higher accuracy, when entire word models are trained. Each segment model is represented by an HMM that consists of a customizable number of states and transitions, and a common topology for speech recognition is a left-to-right model [21].

Each state q of the model possesses an individual output probability:

$$\mathrm{b}(\mathbf{o}_m) = \mathrm{P}(\mathbf{o}_m | q_m = i) \quad i = 1, 2, \ldots, Q \,, \tag{1}$$

where Q denotes the number of states and $\mathbf{o}_m$ is the feature vector—the so-called *observation*—at frame m. For modeling speech, the output probabilities are usually represented by a *Gaussian mixture model* (GMM) comprised of K components:

$$\mathrm{b}_q(\mathbf{o}) = \sum_{\kappa=1}^{K} \gamma_{\kappa,q} \mathcal{N}(\mathbf{o} | \boldsymbol{\mu}_{\kappa,q}, \underline{\boldsymbol{\Sigma}}_{\kappa,q}). \tag{2}$$

Here, $\mathcal{N}(\mathbf{o} | \boldsymbol{\mu}_{\kappa,q}, \underline{\boldsymbol{\Sigma}}_{\kappa,q})$ is the multivariate normal distribution parameterized by its mean vector $\boldsymbol{\mu}_{\kappa,q}$ and its covariance matrix $\underline{\boldsymbol{\Sigma}}_{\kappa,q}$. Each Gaussian component is scaled with the mixture weight $\gamma_{\kappa,q}$, where

$$\sum_{\kappa=1}^{K} \gamma_{\kappa,q} = 1 \quad \forall q. \tag{3}$$

The complete HMM is defined by the parameter set:

$$\lambda = (\underline{\mathbf{A}}, \mathcal{B}, \boldsymbol{\Pi}), \tag{4}$$

where $\underline{\mathbf{A}}$ denotes a matrix of state transition probabilities and $\mathcal{B}$ is a set containing the parameters of the output probability distributions of all states. The vector $\boldsymbol{\Pi}$ contains the initial probabilities that can be given, for example, by the word probabilities or by a fixed value. The model parameters λ can then be estimated by optimizing a *maximum likelihood* (ML) criterion:

$$\lambda_{\mathrm{ML}} = \arg\max_{\lambda} \mathrm{P}(\mathbf{O} | \lambda), \tag{5}$$

where $\mathbf{O} = \mathbf{o}_1 \ldots \mathbf{o}_T$ represents a sequence of T observations. A solution for Eq. 5 can be found by using the Baum-Welch algorithm [21]. Before we can solve Eq. 5, the mean vectors $\boldsymbol{\mu}_{\kappa,q}$ and covariance matrices $\underline{\boldsymbol{\Sigma}}_{\kappa,q}$ of the respective state output distributions need to be initialized properly. Since ML estimation can only yield a local optimum for a finite number of data points, the initialization of the model parameters can influence the ASR performance considerably later on. A simple approach for initialization is to use the global mean and variance of the entire training data, which is often referred to as *flat-start* initialization. When the temporal occurrence of words in the training material is known, better results are achieved by initializing each word model individually with the corresponding statistics, which is regarded as a *bootstrap* initialization.

After model training (i.e., after the final parameters of Eq. 5 have been obtained), the individual segment models are combined into a compound HMM that incorporates a user-defined grammar of the speech recognition task. For recognizing a speech utterance, the sequence of observed features $\mathbf{o}_1 \ldots \mathbf{o}_T$ can be decoded by searching for the best state sequence $q_1^* \ldots q_T^*$ through the compound HMM:

$$[q_1 \ldots q_T]^* = \arg\max_{q'_1 \ldots q'_T} \mathrm{P}(q'_1 \ldots q'_T | \mathbf{o}_1 \ldots \mathbf{o}_T, \lambda). \tag{6}$$

Equation 6 can be solved by means of a Viterbi decoder, which implements a dynamic programming approach for finding the most likely sequence of states.

2.2 Performance Assessment

In order to assess the ASR performance, it is necessary to compare the speech recognition output with a corresponding reference transcription. Here, one problem arises from the fact that the recognition output can differ from the reference transcription in its length if the underlying grammar is not restricted to a fixed number of words. However, the comparison of two text strings of different lengths can be achieved by using a dynamic programming procedure. This procedure aligns the recognizer output with the reference transcription and then counts the word deletion (W_D), insertion (W_I) and substitution (W_S) errors. The standard metric for assessing ASR performance is given by the word and sentence accuracy, which are computed by:

$$\text{Word Acc.} = 100 \cdot \frac{W - W_D - W_I - W_S}{W}, \tag{7}$$

$$\text{Sent. Acc.} = 100 \cdot \frac{S_C}{S}, \tag{8}$$

where W, S denote the total number of words and sentences in the reference transcription, respectively, and S_C is the number of correctly recognized sentences. In the following, we regard the transcription of a CAPTCHA as a sentence.

3. ATTACKING CAPTCHAS USING ASR

In this section, we conduct a security analysis on a previous version of reCAPTCHA [12] and compare our results with those of Bursztein et al. [3]. We show that state-of-the-art speech recognition is more suitable for evaluating the security strength of a CAPTCHA in that it leads to higher success rates than a generic classification-based approach.

Data Description.

We use data of a previous reCAPTCHA scheme comprising 198 labeled signals where the labels include the respective temporal information about the exact digit positions.

[2]A phoneme is the smallest linguistic unit in a certain language.

The audio files were downloaded from the reCAPTCHA website [12] in 2010 and hand-labeled at our institute. Each CAPTCHA is constructed from a sequence of 8 digits between zero and nine. The CAPTCHAs consist of real speech and the digits in each CAPTCHA are spoken by different male and female speakers. All CAPTCHAs are distorted with additive background noise that represents a time reversed speech signal as it is reported by Tam et al. [26].

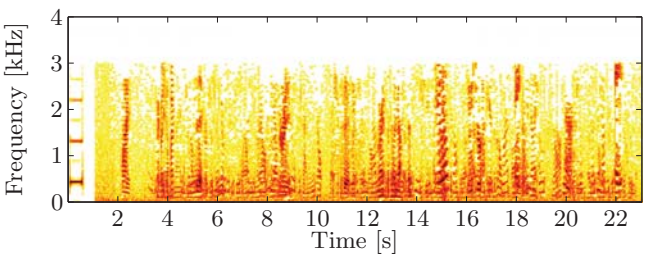


Figure 1: Example of reCAPTCHA 2010 showing the digit sequence "5-6-3-6-0-8-9-8".

The signals are sampled at 8 kHz and the spectral content is band-limited to 3 kHz. Each signal starts with a short tonal sound (beep) which is then followed by eight isolated digits that are clearly separated in time. The signal parts between the digits contain only the interfering background noise and will thus be referred to as *noise segments* later on. Figure 1 provides an example of the CAPTCHA by showing the spectrogram, which is derived from the squared magnitude of the short-time Fourier transform. Since all CAPTCHAs exhibit the same principal structure, it is straightforward to derive a suitable grammar that can be incorporated into the decoder to improve the recognition performance. Due to the limited number of signals, we make use of a K-fold cross-validation. For this purpose, the corpus is partitioned into $K = 6$ complementary subsets, where we train an individual recognizer for each subset by using 60 % of the data and the remaining data is used for assessing the recognition performance. The overall recognition performance, i.e., the word and the sentence accuracies, are obtained by averaging the decoding results of all cross-validation sets.

ASR Training.

Each word is modeled with an HMM that has 3 states per phoneme and 8 Gaussian components per state. Two additional models are utilized for representing the starting tone and the noise-only segments, respectively. We provide our analysis results for two different feature types that are either derived from 13 MFCCs or 13 PLPs. In each case, the features are augmented with their respective first and second order temporal derivatives, yielding a 39-dimensional feature vector. To train the recognition system, it is first necessary to initialize the model parameters λ as defined by Eq. 4. We have found that a flat-start initialization of the model parameters leads to poor recognition rates. As we have the temporal label information at hand, we derive the initial model parameters by using a bootstrap approach. Here, we vary the amount of the bootstrap data and compare the results in Sec. 3.1. After model initialization, several iterations of Baum-Welch training are performed to estimate the model parameters.

Table 1: Average cross-validation results in percent for reCAPTCHA 2010 shown for a varying amount of bootstrap data (Init [%]). The best results are highlighted.

Feat.	Init [%]	Sent Acc.	Word. Acc
MFCC	25	11.11	74.24
	50	10.10	73.42
	75	**14.14**	**74.87**
	100	12.12	73.86
PLP	25	16.67	78.03
	50	**19.19**	**78.98**
	75	13.64	78.34
	100	18.18	77.84

Signal Decoding.

Due to the consistent structure of the CAPTCHA, we utilize a task-specific grammar for decoding:

$$\mathrm{S} \rightarrow \mathrm{T\,N\,D\,N\,D\,N\,D\,N\,D\,N\,D\,N\,D\,N\,D\,N\,D\,N}\,, \tag{9}$$

with S, T, N denoting the start symbol, the beep tone and the noise segment, respectively. The symbol D represents the individual digits, i.e.,:

$$\mathrm{D} \rightarrow \mathrm{zero|one|two|} \cdots \mathrm{|eight|nine}\,. \tag{10}$$

3.1 Evaluation

Table 1 shows the average cross-validation results of reCAPTCHA 2010 for a varying amount of bootstrap data. The PLP features outperform the MFCC features by approximately 5 % for the sentence accuracy, which might be due to the occurrence of different speakers in the CAPTCHA, as the PLP features are more suitable for speaker-independent recognition. The highest sentence accuracy for the PLP features is achieved for initializing the models with 50 % of the available data in the respective cross-validation set. Thus, we can see that an increased amount of bootstrap data does not necessarily result in a higher recognition performance, which might be due to overfitting effects that lead to a reduced generalizability of the models.

When comparing our results with those of Bursztein et al. [3], we can see that our ASR-based approach leads to a considerably increased success rate. The maximum achieved sentence accuracy for reCAPTCHA reported in [3] is 1.52 %, a factor of 13 below the best score of Tab. 1. Hence, it is advisable to use speech recognition methods for evaluating an acoustic CAPTCHA as this provides a more realistic indication of the level of robustness against automated attacks.

4. ANALYSIS OF RECAPTCHA 2014

We now provide a security and usability study of the current reCAPTCHA scheme. For this purpose, we have collected approximately 2,500 CAPTCHAS from the reCAPTCHA website [12] between November and December 2013 where we utilize only a fractional amount of the data for training our system. The respective transcriptions were obtained in a listening experiment that is outlined in detail in Sec. 4.2. We selected reCAPTCHA since it is widely used and previous versions of the scheme have proven to be relatively robust against automated attacks in comparison to other CAPTCHAs [3, 26].

4.1 Overview of the CAPTCHA Scheme

The current version of reCAPTCHA is similar to the version of 2010 in that it is only constructed from a sequence of digits between zero and nine. However, the total number of spoken words is not fixed but varied between 6 and 12. Furthermore, the digits are spoken in a block-wise manner, which means that some digits are immediately followed by other digits without any noticeable speech pause in between and some of the digits are even overlapping in time. However, all blocks of those contiguously spoken digits are well separated in time by a short speaking pause and the majority of CAPTCHAs consists of three blocks. The digits are spoken by both a male and a female voice at a very slow speaking rate. The speech appears synthetic and the overall voice quality is comparatively low. All signals exhibit the same stationary background noise and the energy of the background noise is inversely proportional to frequency. The sampling frequency is 16 kHz and the signals are bandlimited to approximately 5.8 kHz. An example of the CAPTCHA is given in Fig. 2 by showing the spectrogram.

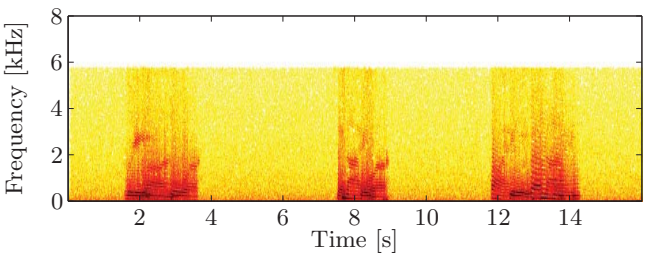


Figure 2: Example of reCAPTCHA 2014 showing the digit sequence "314-694-5279".

4.2 Obtaining Transcriptions

To obtain the CAPTCHA transcriptions that are essential for training the ASR system, we conducted a listening experiment at our institute. The listening experiment took place in a controlled environment, i.e., an audiometric room, in which the acoustic conditions were kept constant throughout. All audio signals were presented via professional studio headphones and the audio volume could be adjusted to the level of comfort by the participants.

A group of 12 human listeners participated in the experiment and each participant was asked to label a set of 50 CAPTCHA signals. The participants were briefed that the signals consist of a varying number of digits that are separated in time by distinct speech pauses. The task was to provide a numeric transcription using only the letters {"0", "1", "2", . . . , "9"} for each digit as well as a marker symbol "-" for each speech pause, e.g., a complete transcription may read "894-770-136". The marker symbol is used to enable an identification of individual digit blocks later on. We created a graphical user interface in MATLAB that enables the audio playback as well as the labeling for a set of successively presented CAPTCHA signals. The setup allows the user to playback each audio signal multiple times until a transcription has been saved by the user. It was not allowed to play or edit a previously transcribed CAPTCHA. We collected 4 transcriptions for each CAPTCHA to enable an identification of reliable transcriptions later on. Some of the listeners participated in the experiment multiple times where we ensured that no CAPTCHA was labeled more than once by the same person. In this way, we collected 1000 transcriptions, corresponding to 250 individual CAPTCHAs.

Table 2: Inter-labeler agreement in percent for reCAPTCHA 2014. The results are shown for the individual digit block transcriptions and for the full transcription (i.e., an agreement on all blocks). The scores are based on the listening test results of 250 different CAPTCHAs.

	# Agreements			
	1 (No)	2	3	4 (All)
Block transcription	11.20	29.73	31.87	27.20
Full transcription	49.20	36.00	10.00	4.80

Ethical Considerations.

Before performing the listening test, all participants were informed that they were to take part in a scientific study and that their data was used to analyze the properties of a CAPTCHA. All data collected during our study was used in an anonymized way so that there is no link between collected data and individual participants. Our institute does not fall under the jurisdiction of an IRB or similar ethics committee.

4.3 Usability

After the listening experiment, we received a feedback from some of the participants, stating that most signals are very difficult to understand. This can be confirmed by the fact that there is only a relatively small number of CAPTCHAs where the transcriptions of four different participants match with each other. Table 2 shows the percentage of CAPTCHAs and digit blocks on which there was no agreement (1), and on which 2, 3 or 4 participants agreed. Since each CAPTCHA consists of a fixed number of digit blocks, an agreement on all digit blocks is equivalent to an agreement on the full transcription of the CAPTCHA.

We can see from Tab. 2 that the overall transcription agreement between the participants is comparatively low, as there is only a full agreement on the transcriptions for 4.8 % of the signals between all participants and 49.2 % of the signals received diverging transcriptions, i.e., no pair of participants agreed on the same transcription. This indicates that the spoken digit sequences are perceived differently among the individual participants and that most CAPTCHAs are presumably only partially solvable. Nevertheless, the majority of digit blocks, i.e., 59.07 % (31.87 % + 27.20 %), received an identical transcription by at least 3 participants, which we exploit for training the ASR system in the next section.

4.4 Security Analysis

We assess the security of reCAPTCHA by first training our automated solver offline and then attacking the CAPTCHA signals on the reCAPTCHA website [12]. The employed ASR system uses the type of models described in Sec. 3, since the principal recognition task remains unchanged.

Training the ASR System.

Due to a high discrepancy between the CAPTCHA transcriptions, we are not able to train the ASR system on the full transcriptions in a reliable manner. In order to cope with this issue, we identify those signal parts, i.e., digit blocks, with a high inter-labeler agreement. We extract all digit

blocks that were labeled with at least 3 identical transcriptions. The extraction of individual digit blocks from the CAPTCHAs is based on a *voice activity detector* (VAD), which identifies the temporal locations of digit blocks and noise segments. The VAD represents the same ASR system that is used for the attack with the difference that it uses only two different models, i.e., one universal model for speech and one model for noise. The VAD was trained by using 5 randomly chosen CAPTCHAs that were manually labeled. Using the VAD yields an overall number of 443 individual digit blocks—corresponding to 1500 single digits—that are used for training the attacking ASR system.

Performing the Attack.

The attack on the reCAPTCHA scheme was conducted on February 12th and 13th 2014 by solving the acoustic CAPTCHAs on the reCAPTCHA website [12] via an automated script. This script successively downloads the audio signals and submits the respective transcriptions that have been produced by the ASR decoder. Due to a limited number of possible CAPTCHA downloads within a certain period of time from a single IP address, we used TOR [8] to increase the number of available IP addresses and enable a larger number of CAPTCHA requests for our experiment.

When ignoring the Internet connection speed, the solving time per CAPTCHA depends mainly on the Viterbi decoder of HTK. On average, we solve 2 CAPTCHAs per second on a standard desktop computer, which is 32× faster than a user would need to listen to the signals.

For speech recognition, we exploit our prior knowledge about the CAPTCHA scheme and adjust the employed grammar as follows:

$$\begin{aligned} &\mathrm{S} \rightarrow \mathrm{N\,B\,N\,B\,N\,B\,N}\,, \\ &\mathrm{B} \rightarrow \mathrm{D\,D\,[D]\,[D]}\,, \end{aligned} \tag{11}$$

where S, B, and N denote the start symbol, a digit block and a noise segment, respectively. The symbol D represents the individual digits as defined by Eq. 10 and [D] specifies an optional digit.

We solved 10,864 CAPTCHAs in total, receiving a correct response from the website for 6,827 attempts, which corresponds to an average success rate of 62.8 %.

Comparing this result with the success rate of 19 % for the older version of reCAPTCHA (cf. 3) we can see that the current scheme appears to be easier to for the ASR system. We assume that this is due to the fact that the current version of reCAPTCHA uses synthetic speech rather than real speech recordings as well as a relatively simple background noise signal[3].

Usability Revisited.

After the attack, we possess a reference transcription for 6,827 CAPTCHA signals, which enables us to assess the human recognition scores for the reCAPTCHA scheme. We have conducted a subsequent listening experiment, involving 10 participants, where each participant was asked to provide a transcription for 50 randomly chosen CAPTCHAs. Again, the participants were briefed that they were to take part in a scientific study. With the help of this second listening test, we have collected the human recognition scores for 500 different CAPTCHA signals. Table 3 depicts the human recognition scores for the current reCAPTCHA scheme.

[3]The difference in the complexity of the background noise signals is clearly noticeable when comparing Fig. 1 and 2.

Table 3: Human recognition scores in percent for reCAPTCHA 2014. The scores are based on the listening test results of 500 different CAPTCHAs (sentences), corresponding to 5,140 digits (words).

	Sent Acc.	Word Acc.
Mean	24.40	84.69
Std. dev.	17.35	6.55

We can see from Tab. 3 that the mean sentence accuracy is quite low at only 24.40 %, rendering the intelligibility of the CAPTCHA fairly questionable. The word accuracy is given by 84.69 %, which represents an acceptable score regarding the intelligibility of individual digits.

5. PERCEPTUALLY MOTIVATED DESIGN

In order to render the training of an automated CAPTCHA solver more difficult, it is beneficial to introduce as much variability into the speech signals as possible. In principle, this can be achieved by expanding the lexicon, i.e., by using a higher number of words for the CAPTCHA. Further, it is advantageous to introduce several speakers into the signals that exhibit different speaking characteristics like fundamental frequency, speaking rate and, particularly, dialect. Thus increasing the *complexity* of the speech signal will typically compel an attacker to acquire more labeled CAPTCHA signals for training a robust automated solver.

Another important design aspect is to prevent certain attack strategies. Segmentation-based attacks (cf. [3, 26]) for instance can be hindered or even prevented when designing a CAPTCHA by filling the speech pauses with supplementary signals that have roughly the same energy as the speech, but do not carry any semantic information. This procedure can also confuse the attacking system, especially when the supplementary signals exhibit similar spectral characteristics as the speech signal itself. Suitable signal types are, for instance, speech shaped noise or multi-talker babble noise.

5.1 Exploiting Auditory Perception

There is a variety of complex acoustic scenarios that humans with normal hearing can easily cope with, whereas they represent a challenging task for ASR. One major problem for ASR is the simultaneous occurrence of multiple sound sources that overlap both in time and in frequency. However, the human auditory system possesses the ability of isolating acoustic streams from a mixture of sound events, which is known in the literature as the phenomenon of *auditory streaming* [2, 9]. Hence, for designing robust acoustic CAPTCHAs, we can attempt to exploit the human ability of sound separation while creating a more challenging ASR task. We can construct the audio signal such that the individual words are somewhat superimposed on each other. This renders word isolation in time virtually impossible as there is no separation between adjacent words. Of course, the superposition of words has to be done in a way that allows a human listener to identify the designated order of words. Thus, there has to be a certain time delay between the beginning of two adjacent words.

Another difficulty for ASR is the recognition of speech in the presence of reverberation, as this causes a tempo-

ral smearing or dispersion of the audio signal. Despite this temporal smearing, speech intelligibility in reverberant environments remains relatively high for human listeners [28].

Based on these insights, we generate CAPTCHAs by exploiting both of the above mentioned concepts, i.e., auditory streaming and reverberation, as discussed in the following.

5.2 Generation of Signals

In order to produce CAPTCHAs that are comparable to the reCAPTCHA schemes, we also restrict the underlying vocabulary to digits between zero and nine. For signal generation, we use the TIDIGITS speech corpus [17] which originally consists of 326 different speakers (111 men, 114 women, 50 boys and 51 girls) each pronouncing 77 digit sequences. Here, we created a subset of the corpus by incorporating only the single-digit recordings of 25 male and 25 female speakers, yielding a total number of 1000 individual digits that represent the signal repository for our CAPTCHA generation approach.

Creation of Digit Blocks.

Random digit signals are chosen from the database, alternating male and female speakers. The signals are concatenated in time such that two consecutive digits are superimposed to some extent and we refer to the outcome as a digit block in the following. The number of digit blocks is varied between 4 and 5, which results in a maximum number of 8 and 10 digits per CAPTCHA, respectively. At first, all digit blocks are separated by a silence segment with its length chosen randomly and uniformly from the interval [0.75 s, 2.5 s]. The superposition of signals is based on the short-time *root mean square* (RMS) power of the isolated digit sequences and is controlled by a small set of parameters. As a first step, the RMS power of the k-th digit signal $x_k(t)$ is computed in a frame-wise manner:

$$p_k(m) = 10\ \log_{10}\left(\sqrt{\frac{1}{L}\sum_{i=0}^{L-1} x_k(mR+i)^2}\right). \tag{12}$$

In order to perform signal concatenation, we derive an individual pivot element for each RMS power curve by searching for the location of the maximum:

$$\hat{m}_k = \arg\max_{m}\left(p_k(m)\right). \tag{13}$$

Next, we determine two power minima that enclose $\hat{m}_k$ and lie within a certain power range. To achieve this, we first compute a power floor for each digit:

$$\check{p}_k = \hat{p}_k - \beta\overline{p_k(m)}, \tag{14}$$

where $\hat{p}_k$ represents the value of the power maximum that is:

$$\hat{p}_k = p_k(\hat{m}_k), \tag{15}$$

with β as a threshold constant and $\overline{p_k(m)}$ denoting the sample mean of $p_k(m)$. The location of the power minimum on the left hand side of $\hat{m}_k$ is then given by:

$$\check{m}_k^{\mathrm{L}} = \arg\min_{0\ldots\hat{m}_k}\left(|p_k(m) - \check{p}_k|\right) \tag{16}$$

and the location of the power minimum on the right hand side of $\hat{m}_k$ reads:

$$\check{m}_k^{\mathrm{R}} = \arg\min_{\hat{m}_k\ldots M}\left(|p_k(m) - \check{p}_k|\right), \tag{17}$$

where M is the number of available frames. Using these maximum and minimum values, we compute two anchor points for each digit. The left anchor point of the k-th digit is midway between $\hat{m}_k$ and $\check{m}_k^{\mathrm{L}}$:

$$\acute{m}_k = \frac{\hat{m}_k - \check{m}_k^{\mathrm{L}}}{2} + \check{m}_k^{\mathrm{L}} \tag{18}$$

and the right anchor point of the k-th digit is centered between $\hat{m}_k$ and $\check{m}_k^{\mathrm{R}}$:

$$\grave{m}_k = \frac{\check{m}_k^{\mathrm{R}} - \hat{m}_k}{2} + \hat{m}_k. \tag{19}$$

A digit block is then created by aligning two digits at their respective left and right anchor points. Therefore, we superimpose two consecutive digit signals in time such that:

$$\acute{m}_{2l} \overset{!}{=} \grave{m}_{2l+1} \quad \forall\, l = 0, 1, 2, \ldots \tag{20}$$

is fulfilled and we denote the signal that contains all digit blocks by $y(t)$. Figure 3 shows an example of the block creation method by means of the RMS power curves for two consecutive digits.

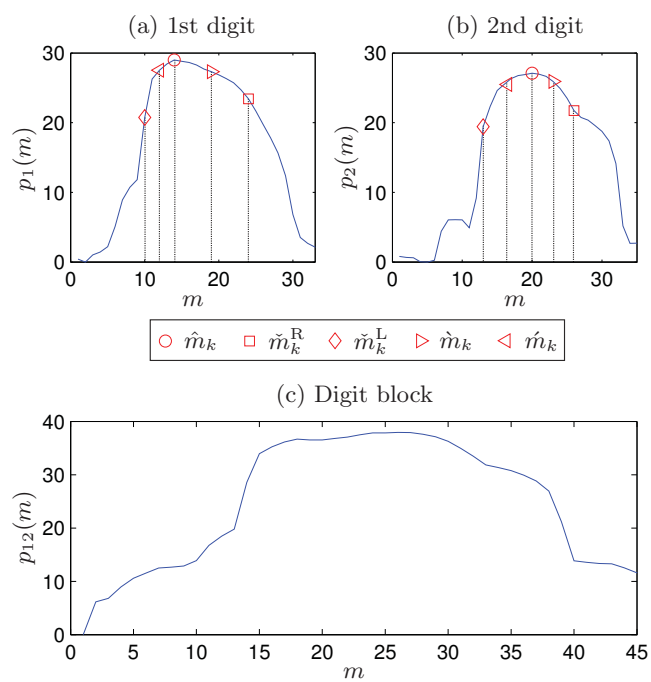


Figure 3: Example of the block creation method. (a) shows the short-time RMS power of the first digit and (b) shows the short-time RMS power of the second digit. The short-time RMS power of the digit block that results from signal concatenation is shown in (c).

Incorporating Background Noise.

After creating the sequence of digits blocks $y(t)$, all silent parts within the signal are superimposed by multi-talker babble noise from the NOISEX-92 database [27]. The NOISEX babble noise is 235 s long and we extract random noise segments of suitable length for creating the noise parts of

the CAPTCHA. The noise segments are scaled such that the RMS power curve of $y(t)$ exhibits no conspicuous peaks and remains fairly stationary. We denote the noisy waveform of digit blocks by $y'(t)$.

It should be stressed at this point that the utilized corpora, i.e., TIDIGITS and NOISEX, were chosen for convenience to provide a proof of concept. We are aware of the fact that the data is not suitable for a practical application of the CAPTCHA as these corpora are publicly available, which would facilitate the training of automated CAPTCHA solvers.

Reverberating Signals.

The final CAPTCHA signal $z(t)$ is obtained by a convolution of the noisy digit sequence $y'(t)$ with an artificially generated room impulse response $w(t)e^{-t/\tau}$:

$$z(t) = y'(t) * \left(w(t)e^{-t/\tau}\right), \tag{21}$$

where $w(t)$ represents white Gaussian noise with zero mean and unit variance and τ is a predefined decay time.

For creating CAPTCHA signals, two different reverberation times are compared. A common way for characterizing reverberation is the time constant T_{60}, which is the time it takes for the room impulse response power to drop by 60 dB. We create CAPTCHA signals for $T_{60} = 100$ ms and for $T_{60} = 300$ ms and compare the human performance as well as the robustness against ASR-based attacks in Sec. 5.3 and Sec. 5.4, respectively. The spectrogram of the CAPTCHA is shown in Fig. 4 for a reverberation time of 100 ms.

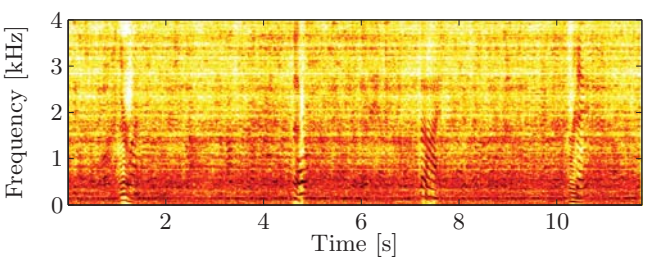


Figure 4: Example of the proposed CAPTCHA design for $T_{60} = 100$ ms showing the digit sequence "01-64-75-36".

5.3 Usability

We assess the usability of the proposed CAPTCHA, i.e., the human success rate, by means of the same listening experiment as it is outlined in Sec. 4.2. This environment does, of course, constitute an ideal situation, which has been chosen for reproducibility and consistency of results.

Two different variants of the proposed CAPTCHA are compared, one with a reverberation time of 100 ms and the other with a reverberation time of 300 ms. In the experiment, we find the results for 200 randomly selected signals at each reverberation time.

The overall number of individual participants was 20 and each participant was asked to transcribe a set of 50 audio signals per experiment. In order to derive the transcription agreements as in Sec. 4.2, each signal is labeled by four different listeners. Some participants attended multiple times where we again ensured that no CAPTCHA was labeled more than once by the same listener.

Table 4: Inter-labeler agreement in percent for the proposed CAPTCHA. The results are shown for the individual digit block transcriptions and for the full transcription (i.e., an agreement on all blocks). The scores are based on the listening test results of 200 different CAPTCHAs for each reverberation time.

	# Agreements			
	1 (No)	2	3	4 (All)
Block transcription	10.43	11.26	24.69	53.63
Full transcription	19.50	28.00	34.00	18.50

(a) $T_{60} = 100$ ms

	# Agreements			
	1 (No)	2	3	4 (All)
Block transcription	9.55	31.42	58.67	0.37
Full transcription	57.50	29.00	13.50	0.00

(b) $T_{60} = 300$ ms

Table 5: Human recognition scores in percent for the proposed CAPTCHA. The scores are based on the listening test results of 800 CAPTCHAs (sentences), corresponding to 7,280 digits (words).

	Sent Acc.	Word Acc.
Mean	56.38	91.74
Std. dev.	21.47	7.18

(a) $T_{60} = 100$ ms

	Sent Acc.	Word Acc.
Mean	37.81	86.88
Std. dev.	17.65	7.90

(b) $T_{60} = 300$ ms

The results for the transcription agreement between the participants are shown in Tab. 4a and Tab. 4b, for the reverberation time $T_{60} = 100$ ms and for $T_{60} = 300$ ms, respectively. The actual human recognition scores, computed by using the reference transcription, are presented in Tab. 5.

Considering first the reverberation time $T_{60} = 100$ ms, we can see from Tab. 4a that the majority of digit blocks (53.63 %) received a transcription agreement from all participants, whereas the number of agreements on the full transcription is comparatively low (18.50 %). This effect is also reflected in the human recognition scores (cf. Tab. 5a), where the word accuracy is satisfactorily high (91.74 %) but the sentence accuracy is only 56.38 %. For the reverberation time $T_{60} = 300$ ms, the number of transcription agreements is significantly lower than with $T_{60} = 100$ ms. Here, the majority of CAPTCHA signals (i.e., 57.50 %) has no agreement on the full transcription. Furthermore, only 0.37 % of the digit blocks show an agreement between all participants and no CAPTCHA has a full transcription agreement between all participants. Regarding the human recognition scores for $T_{60} = 300$ ms, we can see from Tab. 5b that only 37.81 % of the CAPTCHAs are transcribed correctly.

5.4 Security Analysis

The security analysis of the proposed CAPTCHA is based on the same ASR system that is used in the previous sections, to allow for a fair comparison.

System Setup.

Since we have generated all CAPTCHAs ourselves, we have the exact knowledge about the temporal positions of digits within the audio signals. Thus, we utilize 200 signals for bootstrapping the HMM parameters, which represents a high-quality initialization procedure compared to Sec. 3 and Sec. 4, since the transcriptions are perfectly time-aligned with the spoken digits.

After initializing the HMMs, the model parameters are reestimated using a varying number of CAPTCHA signals to analyze the influence of additional training material on the recognition performance. Here, we vary the number of training examples that are used for model reestimation between 200 and 1600 CAPTCHAs. In addition, we compare the results for three different reverberation times, i.e., 0 ms, 100 ms, and, 300 ms, where the former represents the case of a non-reverberant signal. In each case, the recognition results are based on 10,000 randomly chosen CAPTCHAs. We apply a task-specific grammar for signal decoding:

$$\begin{aligned} &\mathrm{S} \rightarrow \mathrm{N\,B\,N\,B\,N\,B\,N\,B\,N\,[B\,N]}\,, \\ &\mathrm{B} \rightarrow \mathrm{D\,D}\,, \end{aligned} \tag{22}$$

where the symbols S, B, D and N are as defined by Eq. 11.

Results.

Table 6 shows the ASR recognition results for a varying number of training examples and different reverberation times. It should be emphasized that this security analysis of the proposed CAPTCHA is much more rigorous than the one applied to reCAPTCHA (cf. Sec. 4.4) as we make use of the ideal reference transcriptions for training the ASR system. Furthermore, the number of digits that were utilized for training is 1.2–9.6 × (1800–14400 digits) higher than for reCAPTCHA (1500 digits).

We can see from Tab. 6 that the sentence accuracy for the non-reverberant case, i.e., $T_{60} = 0$ ms, and the lowest amount of training examples, i.e., 200 CAPTCHAs, is comparatively high at 15.86 %. Additionally, the word and sentence accuracies are further improved by increasing the number of training examples. Here, it appears that the relationship between the number of training examples and the resulting accuracies is virtually linear. Thus, an ASR-based CAPTCHA solver can easily cope with the overlapping digit sequences as well as with the babble noise segments to some degree, rendering these signals non-robust against automated attacks.

For $T_{60} = 100$ ms, the word and sentence accuracy drops by approximately 10–20 %, depending on the number of training examples. This demonstrates that already moderate reverberation effects are already problematic for speech recognition. However, the ASR system is able to find the correct transcription for approximately 6 % of the attacked CAPTCHA signals, averaged over the number of utilized training examples.

As one would expect, the lowest ASR scores are given for $T_{60} = 300$ ms. Here, the ASR system only identifies approximately 3 % of the CAPTCHAs correctly.

We can conclude that the best trade-off between usability and security for the proposed scheme is given for $T_{60} = 100$ ms. A side-by-side comparison between the proposed CAPTCHA and the current reCAPTCHA scheme is provided by Tab. 7. Here, the success rate of the attack is 5.33 % when using 1800 digits for training the ASR system. This is a factor 12 below the success rate found for the reCAPTCHA scheme (62.8 %) by using 1500 digits for system training. Furthermore, the human success rate for the proposed scheme was found to be 56.38 % that is twice as high as for the current reCAPTCHA scheme.

Table 6: ASR results in percent for the proposed scheme. The scores are based on 10,000 different CAPTCHAs (sentences), corresponding to 90,140 digits (words).

# Train	T_{60} [ms]	Sent Acc.	Word Acc.
200	0	15.86	77.03
200	100	5.33	64.49
200	300	1.25	56.11
400	0	17.42	78.38
400	100	5.06	65.32
400	300	2.34	60.20
800	0	20.38	79.71
800	100	6.87	67.21
800	300	3.14	62.88
1600	0	26.43	82.43
1600	100	6.26	67.37
1600	300	4.11	64.66

Table 7: Summary of recognition scores in percent for ASR and human listeners, shown for the proposed CAPTCHA ($T_{60} = 100$ ms) and the current reCAPTCHA scheme.

	ASR	Human
Proposed CAPTCHA	5.33	56.38
reCAPTCHA 03/2014	62.8	24.40

6. CONCLUSION AND FUTURE WORK

In this paper, we show that it is advisable to utilize state-of-the-art speech recognition techniques for evaluating the security of acoustic CAPTCHAs, as this yields significantly higher success rates for CAPTCHA solving than previously suggested classification methods and is therefore more indicative of the security risk of a CAPTCHA. Using the well-known reCAPTCHA as an example, we can see that the general characteristics of the scheme have not changed notably within the last four years, which leads to comparatively high success rates for automated attacks. In addition, the degree of signal distortion has been increased for this scheme at the cost of reducing human accuracy, rendering the trade-off between usability and security problematic.

We propose an experimental CAPTCHA scheme that exploits both the auditory capabilities of humans and weaknesses in current ASR techniques. In this scheme, we render the ASR-based attack more difficult by introducing overlapping speakers and artificial room reverberation effects. The outcome of our experiments is that the proposed CAPTCHA exhibits a far better trade-off between human usability and security as the current reCAPTCHA scheme.

Our results confirm the findings of previous research in that we show that a very conservative CAPTCHA, even if it is only barely intelligible for humans, can potentially be learned by an automated system at a relatively low cost. We can hence assume that it is infeasible to create a small-vocabulary-based CAPTCHA with a very high human success rate ($\geq 90\,\%$) while remaining robust against automatic solvers, i.e., yielding only insignificantly low success rates ($\leq 1\,\%$ or even $\leq 0.01\,\%$). Thus, it becomes necessary to investigate into new directions for CAPTCHA design, such as the utilization of a larger vocabulary or the incorporation of context-based questions that require human intelligence.

For the time being, though, our acoustic CAPTCHA did succeed in significantly improving human pass rates—from 24 % to 56 %—while reducing the attack success rate of an ASR solver from 63 % to 5 %, when comparing with the current, state-of-the-art reCAPTCHA scheme. This lends credibility to the general approach of explicitly considering human auditory processing capabilities in CAPTCHA designs, and should serve as a building block for new systems that offer a satisfactory trade-off between usability and security.

Acknowledgments

The authors would like to thank all participants of the listening experiments. The research was supported by the DFG Research Training Group GRK 1817/1. The CAPTCHA examples of our proposed scheme are available upon request.

7. REFERENCES

[1] S. Bohr, A. Shome, and J. Z. Simon. Improving Auditory CAPTCHA Security. Technical report, ISR, A. James Clark School of Engineering, 2008.

[2] A. S. Bregman. *Auditory scene analysis: The perceptual organization of sound*. MIT press, 1994.

[3] E. Bursztein, R. Bauxis, H. Paskov, D. Perito, C. Fabry, and J. C. Mitchell. The Failure of Noise-Based Non-Continuous Audio Captchas. In *Proc. IEEE Symposium on Security and Privacy*, 2011.

[4] E. Bursztein and S. Bethard. Decaptcha Breaking 75% of eBay Audio CAPTCHAs. In *Proc. WOOT*, 2009.

[5] K. Chellapilla, K. Larson, P. Y. Simard, and M. Czerwinski. Building Segmentation Based Human-Friendly Human Interaction Proofs (HIPs). In *Human Interactive Proofs*. Springer, 2005.

[6] R. Datta, J. Li, and J. Z. Wang. IMAGINATION: A Robust Image-based CAPTCHA Generation System. In *Proc. ACM MM*, 2005.

[7] dfactory. Math Captcha, 2014. `http://wordpress.org/plugins/wp-math-captcha/`.

[8] R. Dingledine, N. Mathewson, and P. Syverson. Tor: The Second-generation Onion Router. In *Proc. USENIX Security Symposium*, 2004.

[9] P. Divenyi. *Speech separation by humans and machines*. Springer, 2005.

[10] I. Fischer and T. Herfet. Visual CAPTCHAs for document authentication. In *Proc. MMSP*, 2006.

[11] M. Gales and S. Young. The Application of Hidden Markov Models in Speech Recognition. *Foundations and Trends in Signal Processing*, 2007.

[12] Google. reCAPTCHA. `http://www.recaptcha.net`.

[13] H. Hermansky, B. Hanson, and H. Wakita. Perceptually based linear predictive analysis of speech. In *Proc. ICASSP*, 1985.

[14] A. Hindle, M. W. Godfrey, and R. C. Holt. Reverse Engineering CAPTCHAs. In *Proc. WCRE*, 2008.

[15] G. Hinton, L. Deng, D. Yu, G. E. Dahl, A.-r. Mohamed, N. Jaitly, A. Senior, V. Vanhoucke, P. Nguyen, T. N. Sainath, et al. Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups. *IEEE Signal Processing Magazine*, 2012.

[16] G. Kochanski, D. P. Lopresti, and C. Shih. A reverse turing test using speech. In *Proc. INTERSPEECH*, 2002.

[17] R. G. Leonard and G. Doddington. TIDIGITS Linguistic Data Consortium, 1993. Linguistic Data Consortium.

[18] R. P. Lippmann. Review of neural networks for speech recognition. *Neural computation*, 1989.

[19] N. Morgan, H. Bourlard, and H. Hermansky. Automatic Speech Recognition: An Auditory Perspective. In *Speech Processing in the Auditory System*. Springer, 2004.

[20] picatcha.com, 2014. `http://picatcha.com`.

[21] L. Rabiner and B.-H. Juang. *Fundamentals of Speech Recognition*. Prentice Hall, 1993.

[22] S. Sano, T. Otsuka, and H. Okuno. Solving Google's Continuous Audio CAPTCHA with HMM-Based Automatic Speech Recognition. In *Advances in Information and Computer Security*, 2013.

[23] A. Schlaikjer. A dual-use speech CAPTCHA: Aiding visually impaired web users while providing transcriptions of Audio Streams. *LTI-CMU Technical Report*, 2007.

[24] R. Soni and D. Tiwari. Improved captcha method. *IJCA*, 2010.

[25] J. Tam, J. Simsa, D. Huggins-Daines, L. von Ahn, and M. Blum. Improving Audio CAPTCHAs. In *Proc. SOUPS*, 2008.

[26] J. Tam, J. Simsa, S. Hyde, and L. von Ahn. Breaking Audio CAPTCHAs. In *Proc. NIPS*, 2008.

[27] A. Varga and H. J. M. Steeneken. Assessment for automatic speech recognition: II. NOISEX-92: A database and an experiment to study the effect of additive noise on speech recognition systems. *Speech Communication*, 1993.

[28] T. Virtanen, R. Singh, and B. Raj. *Techniques for Noise Robustness in Automatic Speech Recognition*. Wiley, 2012.

[29] L. von Ahn, M. Blum, N. J. Hopper, and J. Langford. CAPTCHA: Using Hard AI Problems for Security. In *EUROCRYPT*. Springer, 2003.

[30] L. von Ahn, M. Blum, and J. Langford. Telling Humans and Computers Apart Automatically. *CACM*, 2004.

[31] J. Yan and A. S. El Ahmad. Breaking visual captchas with naive pattern recognition algorithms. In *Proc. ACSAC*, 2007.

[32] J. Yan and A. S. El Ahmad. A Low-cost Attack on a Microsoft CAPTCHA. In *Proc. ACM CCS*, 2008.

[33] S. Young. The HTK Hidden Markov Model Toolkit: Design and Philosophy. *Entropic Cambridge Research Laboratory, Ltd*, 1994.

Understanding Visual Perceptions of Usability and Security of Android's Graphical Password Pattern *

Adam J. Aviv
U.S. Naval Academy
Annapolis, MD
aviv@usna.edu

Dane Fichter
Swarthmore College
Swarthmore, PA
dficht1@swarthmore.edu

ABSTRACT

This paper reports the results of a user study of the Android graphical password system using an alternative survey methodology, pairwise preferences, that requests participants to select between pairs of patterns indicating either a security or usability preference. By carefully selecting password pairs to isolate a visual feature, a visual perception of usability and security of different features can be measured. We conducted a large IRB-approved survey using pairwise preferences which attracted 384 participants on Amazon Mechanical Turk. Analyzing the results, we find that visual features that can be attributed to complexity indicated a stronger perception of security, while spatial features, such as shifts up/down or left/right are not strong indicators for security or usability. We extended and applied the survey data by building logistic models to *predict* perception preferences by training on features used in the survey and other features proposed in related work. The logistic model accurately predicted preferences above 70%, twice the rate of random guessing, and the strongest feature in classification is *password distance*, the total length of all lines in the pattern, a feature *not* used in the online survey. This result provides insight into the *internal visual calculus* of users when comparing choices and selecting visual passwords, and the ultimate goal of this work is to leverage the visual calculus to design systems where inherent perceptions for usability coincides with a known metric of security.

1. INTRODUCTION

Graphical passwords, credentials based on users recalling or drawing shapes or images, have become more common place with the increase in touch oriented devices, such as smartphones and tablets. Perhaps the most prevalent graphical password system in use is Android's graphical password pattern which comes standard on Android devices. Like any password system, graphical passwords suffer from many of the same issues as text based passwords: users select weak and insecure passwords. We know about text-based password from large studies [10, 14, 19, 20, 13] of publicly available, real-world passwords databases [2] and through collaboration with industry [9] and academic institutions [19].

Studies of the Android password pattern have demonstrated that users are similarly inclined to make poor password choices [27, 3]. Unlike studies in the text-based password space which are based on large-scale analysis of leaked or real-world password data, the results for user selection of graphical passwords are generally based on in-lab surveys where participants are asked to provide a password that meets some criteria, such as being memorable or easy or secure. The findings are informative and persuasive regarding user-choice in this domain – users are inclined to choose weak passwords as compared to the total entropy of the space – but these results also belie larger challenges of studying graphical passwords in the same maner as text-based passwords. Due to the unavailability of large graphical password corpora, standard metrics, such as guessability probability [18], and large-scale study of actually used passwords are not possible.

Motivated by the challenges of large-scale studies of graphical passwords, we explored the use of a new survey methodology in this space that is based on *pairwise preferences* that attempts to measure the visual perceptions that influence secure and usable choices. The pairwise preference survey consists of users selecting between two passwords, indicating a preference for one password in the pair that meets some criterion, such as perceived security or usability compared to the *other* password. By carefully selecting the password pairs, visual features of the passwords can be isolated and the impact of that feature on users' perception of security and usability can be measured.

It is important to note that pairwise preference provides *different* and *complementary* insights into user choice as compared to related in-lab user studies. Pairwise preference data instead attempts to assess the visual processes that indicate *perceived* security or usability under some personal/user-based understanding and calculation of those terms. Perceptions influence choice, especially for passwords where users are consciously attempting to select password that adhere to a security and usability criteria that may not be well defined. The goal of this research is to better understand the visual process that drive security and usability perceptions that in turn can inform new designs of security systems where user perceptions actually match some known metrics of security.

Using the pairwise preference methodology we conducted a large IRB-approved survey that attracted 384 participants on Amazon Mechanical Turk. The resulting dataset averages 19.6 preference selections for over 1,100 password pairs that focus on 6 visual features, including password length, crosses, and left/right and up/down shifting. Additionally, users provided individual security and usability ratings of over 2,000 passwords, averaging 4.6 ratings per

*This research was partially supported by ONR grant N001614WX30023, Swarthmore Faculty Research Grant, and the Swarthmore STEM Student Summer Support. Material is based upon work supported by the Maryland Procurement Office under contract H98230-14-C-0127. Additional acknowledgement to Aashish Srinvas for preliminary work on this research.

©2014 Association for Computing Machinery. ACM acknowledges that this contribution was authored or co-authored by an employee, contractor or affiliate of the United States government. As such, the Government retains a nonexclusive, royalty-free right to publish or reproduce this article, or to allow others to do so, for Government purposes only.

ACSAC '14 New Orleans, LA USA
Copyright 2014 ACM ACM 978-1-4503-3005-3/14/12 ...$15.00
http://dx.doi.org/10.1145/2664243.2664253.

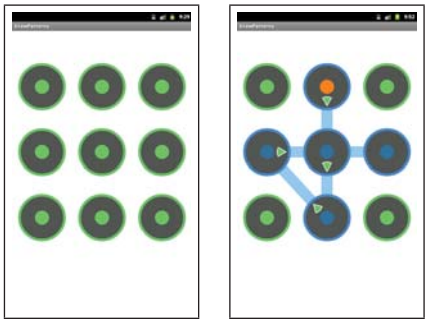


Figure 1: A sample pattern entry screen and a sample pattern as appearing in the survey. The orange marker indicates the start point with shaded lines and arrows indicating the direction of the pattern.

password. In analysis, we find that *length*, the number of contact points used in the password, is the strongest separator of usable or security preferences —passwords with more contact points are perceived as more secure as compared to those patterns with fewer contact points. Other visually complex features, such as crosses, also affect preferences, and when comparing complexity features, users were more inclined to perceive crossing patterns as the most secure. Spatial features, such as patterns shifted to the right/left or up/down, have little to no effect on user perceptions. When users individually rated passwords, they noted that perceived symmetry of the password negatively correlated with security, as in users find symmetric patterns to have less security, while perceived symmetry has little to no effect on perceived usability.

The survey data was expanded and applied to *predict* user preference of password pairs to model perceptions. Using the standard machine learning technique of logistic regression, we constructed a predictive model trained on features from the online survey as well as additional visual features described in related work [4, 21, 27, 3]. The model is able to predict the preferred secure patterns at an accuracy of 70% and preferred usable patterns at an accuracy of 65%. Surprisingly, of the training features *pattern distance*, the total length of all the lines in the password (not the total number of contact points) was the strongest predictor of a preference, a feature *not* used in the online survey.

2. BACKGROUND

The Android password pattern is a recall based graphical password scheme based loosely on Pass-Go [24] and Draw-A-Secret [17][1]. The user is required to select a pattern by traversing a grid of 3x3 points. A pattern must contain at least four points, cannot use a point twice, and all points along a path must be included (no skipping points). Figure 1 provides a reference visual for the input of the pattern (*left*) as well as a sample pattern (*right*) where the orange dot indicates the start point in the pattern with arrows describing the direction of the traversals. Note that in the '+' shaped pattern, the middle point is not associated with the horizontal traversal since it was previously selected during the vertical traversal. In our survey, we used this visualization of the password pattern because it mimics the pattern entry screen on Android 2.x versions, which is still widely used.

Analysis of the Android password pattern has shown that there exist 389,112 possible passwords [6]. Despite the relative complexity of the scheme, the total number of passwords is actually quite small compared to a general ordering of points, which would allow for over a million passwords. The entropy of the password space is also quite small, on the order of a 3-digit PIN [27]. Other studies have also suggested that the actual number of patterns that are likely usable, that is, can be entered accurately without repetition, is probably much smaller than 389,112. In a study of accelerometer readings, the authors reported many users complaining about certain patterns being "hard to enter" [7].

[1]For a larger review of the graphical password landscape, we refer to the reader to two surveys [8, 23].

3. RELATED WORK

Password Study Methodologies .

Research in text-based password selection is very well developed. Early work in the area by Morris and Thompson in 1979 showed that humans are bad at choosing passwords [20]. More recent studies break down into two main categories: (1) the study of large leaks of passwords, and (2) survey based studies where users either self-report password statistics or researchers have users provide a password in the context of a system.

The study of large password leaks [10, 14, 19] is a fairly standard method, but it is not clear how well these data sets represent broader password choice because they tend to only contain the passwords that were cracked through conventional brute force methods [1]. This tends to bias results towards a study of weak passwords and not necessarily passwords in general use. In the graphical password arena, the likelihood of such a leaked password set is extremely low because graphical passwords are not used for remote authentication, yet.

Another methodology is to perform surveys where participants self-report properties about their passwords [22, 30]. These surveys are quite helpful in understanding user preferences, but the demographics of these surveys are often limited to university settings. Online surveys provide a much better option because it can reach a broader set of participants with limited expense, and have become a common methodology [18, 28]. Online surveys try and gauge user choice by having participants select passwords to protect an important account, like a bank account or class web pages [29], and then researchers analyze the results [16].

The graphical password studies, due to a lack of large data sets, have relied primarily on in-lab experimentation. For example, they either bring participants into a lab to use the graphical password scheme [24, 17] or perform pen-and-paper experiments [27]. Unfortunately, the collected data for these experiments can be inconsistent [12], particularly when asking participants to provide passwords [17].

3.1 Studies of Passwords on Mobile Devices

Zhao *et al.* investigated the security of the picture password scheme [29] used by Microsoft tablets, which requires users to select a password by circling points and drawing lines over a chosen image. The user then must recall the specific drawings in order to unlock the device. Picture password schemes often suffer from hotspots [26]. Zhao *et al.* collected sample picture passwords by securing a course website with the picture password scheme and analyzing the chosen passwords, including an analysis of images chosen and how circles and lines are drawn upon those images (hotspots). They concluded that for many images, users are likely to circle and draw on certain spots which can be identified using image processing techniques.

Andriotis *et al.* conducted a survey to advance smudge attacks [6] on Android password patterns by training a predictive model with user surveyed data [3]. The goal was to improve the performance of the smudge attack with information about likely user patterns. Androitis *et al.* surveyed 22 participants requesting them to provide a

"secure" password and an "easy" password to help train their predictor. From these results, they found that users are inclined to select certain shapes, like the "L"- or ¬-shape, as well as a strong bias to start in the upper left contact point. We were able to obtain the user choices from this study, and we reevaluated 8 of those choices, selected at random, from the set of 22. The goal is to see if user choice of security and usability is consistent for in-lab settings. We refer to this as the *Bristol* study because it was conducted at the University of Bristol, UK.

In recent related work, Uellenbeck *et al.* developed a method for quantifying the entropy of Android passwords [27] using partial guessing entropy [9]. They found that the security of the password pattern is on the same order as a 3-digit pin, low entropy. They also conducted a number of pen-and-paper surveys of patterns and identified a number of key n-grams, or sequences of swipes, that users are most inclined to chose. For example, the various forms of "L" were very common, and they also identified a strong bias for users to initiate their patterns in the upper-left corner. The work of Uellenbeck *et al.* is very informative and persuasive about the processes of user choice, and these results compliment our own. Unfortunately, we did not have access to the Uellenbeck results until after our online surveys completed, and were not able to integrate their findings with our survey[2]. We believe, however, that there are interesting places to expand prior results by investigating the preferences of security and usability of the identified n-grams. We do note that at least one of the n-grams was in our survey a ¬-like shape that individually rated as usable but not secure.

4. SURVEY METHODOLOGY

The survey was conducted in two rounds on Amazon Mechanical Turk (MTurk), running within a HIT iFrame (Human Intelligence Task). The survey was hosted on a server at our institution, and all data collected was initially stored at Amazon and now resides at a secure server onsite. Survey participants were limited to those residing in the United States over the age of 18, as per the requirements of the IRB. We did not collect additional demographic information, and instead relied on MTurk to properly check the qualifications of the participants.

The online survey is organized into five parts:

- *Pre-Survey Questionnaire*, which inquires about ownership of smartphone and the use of graphical passwords;
- *Secure Pairwise Preference*, which requires participants to select between a pair of pasword patterns based on a security preference;
- *Password Rating Section*, which requires participants to answer questions about individual passwords;
- *Usable Pairwise Preference*, which requires participants to select between a pair of patterns based on a usability preference;
- *Post Survey Questionnaire*, which asks participants to reflect on the survey.

The survey is designed such that participants cannot easily bypass sections by simply clicking through, and if they do so, such actions are easily detected. For a visual reference to the survey parts, refer to Figure 2. On the left is a sample of a preference selection and on the right is post selection. Not pictured is the individual password rating survey which displayed a single pattern with a series of questions.

[2]We do, however, use the start and end points of a pattern as a feature in the predictive models. Like other spatial features, such as up/down and left/right shifting, the start and end point had little to no impact on predicting preferences. See Section 6 for details.

The online survey is interactive and user-responsive, written in a combination of PHP and Javascript. The pairwise survey requires participants to highlight their selections and confirm the selection before proceedings. We also included attention tests that require participants to recognize that a pair of patterns were identical and thus select the same choice, and we performed post survey analysis to measure the structure of the responses for any indication dishonest intentions during survey taking, for example always selecting left or right, or back and forth. A failure of either test could result in a rejection of the data from the study and/or a rejection of the user within the MTurk ecosystem.

4.1 Survey Questions and Formats

Pre-Survey Questions.

The pre-survey questionnaire was designed to gather some baseline data on the participants and their familiarity with smartphones and Android in particular. We asked the following questions:

- *Do you currently own an Android device?*
- *If you own an Android device, do you use the graphical password feature?*
- *How frequently do you have to re-enter your graphical password because you entered it incorrectly?*
- *Do you find that your graphical password sufficiently secures your Android device?*
- *Describe your handedness?*

We found that a large number of the survey participants owned or used an Android device: 354 of the 384, or 92%. Of the Android owners, 257 of the 354 use graphical passwords, or 72%, and 249 of the 257 graphical password users found that their password was "sufficient" or 97% qualified respondents. Due to the informal definition in the questionnaire, the participant is open to define "sufficiently secure" as he/she sees fit. There is likely a range of interpretation, for example the participant may interpret the question as his/her pattern is *sufficiently secure* for the purpose of preventing casual, unintended use of the device, such as snooping by a friend or family member. The term could also imply a stronger sense of security, such as preventing access by a determined actor like law enforcement. In either case, the high percentage of qualified respondents describing their password as sufficient does suggest a level of comfort with their password choice based on some personal security criteria.

With regards to how frequently users must reenter their passwords due to incorrect entry, a potential measure of usability: 11 reported that they must do so "Very Frequently", 44 reported "Frequently," and 153 reported that he/she "Rarely" must reenter their pattern due to incorrect entry. When combined with the previous responses about security, most participants are Android users who feel comfortable with their password choice with respect to a personal security criteria and ease of password entry. Finally, and unsurprisingly, the majority of participants are right handed (336) as compared to ambidextrous (15) and left handed (33).

Pairwise Preference Surveys.

In both pairwise preference surveys, for security and usability, the same survey format was used. Due to the similarity between the surveys, the password rating survey was placed between the two pairwise surveys to break up the work flow. In both pairwise surveys, each participant indicated a security and usability preference on 48 pairs of patterns. The same set of patterns was used in both the secure and usable sections, but the order of the survey as well as the order within the pair (*i.e.*, left vs. right) was randomized

Figure 2: Survey screenshots: (*left*) Secure comparison prior to selection and (*right*) secure comparison after selection before confirmation.

within each survey. The selection of the 48 password pairs was at random from a carefully selected pool of 1,109 possible password pairs. The pair selection methodology is discussed in the following section. An additional 2 pairs were added to the survey for attention tests bringing the total to 50 pairs used in the preference surveys. A visual of the pairwise preference survey is presented in Figure 2, showing both the comparison state, prior to selection, and the confirmation state, post selection.

Depending on the survey part, in the comparison state, the participant was instructed: "Your job is to determine which of the passwords is the most secure" or "most usable." Recall, that there are two pairwise preference surveys, one for indicating secure preferences and one for indicating usable preferences. Since both surveys are visually similar, an additional prompt was presented prior to starting the second pairwise survey on usability: "The next section looks like the first password section you completed, but it is not. Instead of choosing the most secure of the two passwords, your task is to choose the most usable." This information and the fact the two pairwise surveys are separated by the individually rating survey (discussed next), we feel this properly informed and directed the participant.

We left the definition of security and usability intentionally vague in the survey prompts. Participants applied their own definitions to these terms, and it is possible that many interpreted "secure" as "not usable" or "usable" as "not secure" or in some combination. The goal of the survey is not to precisely identify a user preference for security/usability based on a predetermined definition, but rather to assess the collective visual stimulants that would influence a preference for security/usability. Furthermore, individual preferences have little effect on the result, and the combined preference as a fraction of participants selecting a pattern in the pair was considered.

In addition to indicating a preference for a password in the pair, the participant may also indicate "same" if he/she felt that both passwords exhibit the same security and usability. Once a password is selected, it is highlighted, and confirmation state is entered, where the participant must confirm their selection by pressing a "submit" button or select "cancel" to re-select. Similarly, if "same" is selected, the confirmation state is entered with the same choice, submit or cancel.

Finally, each participant was presented with four attention tests, two in the security survey and two in the usability survey. The attention test consisted of showing the participant the same password on the left and right, and, thus, the participant should select "same." Failing three of four attention tests and/or the entropy test, resulted in rejecting the results of the user. In the end, we accepted data from 384 participants and rejected 54. This resulted in an average of 19.6 total preference classifications per password pair, with 11 usable choices and 11 secure choices per pair, on average.

Password Rating Survey.

The password survey differed from the pairwise comparison in that participants were only directed to assess a single password pattern at a time. Each participant rated 25 patterns chosen at random from the 2,214 patterns in the pool of password pairs. The survey presented the participant with a single password surrounded by the following prompts:

- *Does the above password appear to be symmetrical?*
- *Describe any symbols or geometric shapes which appear in the password.*
- *Rate the security of this password.*
- *Describe the usability of this password.*

Questions of symmetry is a binary, "Yes" or "No", response; the question of shapes is an free text, optional answer; and, the usability and security questions provided four responses between "Highly Unusable", "Unusable", "Usable", and "Highly Usable", with "usable" replaced with "secure" appropriately for the security question. Overall, all 2,114 password patterns received ratings averaging 4.6 ratings per password.

Post-Survey Questionnaire.

At the end of the survey, we asked the participant to answer a brief post survey questionnaire.

- *If you use Android's graphical password, how has this survey changed your opinion of your own password?*
  - My password is less secure than I believed.
  - My password is more secure than I believed.
  - My opinion of my password is unchanged.
  - I do not use Android's graphical password.
- *Which statement about the relationship between security and usability of passwords is most true?*
  - The passwords which look the most secure look more difficult to remember.
  - The passwords which look the most secure look easier to remember.
  - There is no clear relationship between usability and security.

The goal of the post-survey questionnaire is to assess if the passwords viewed affected the participant's security preference of his/her own password and if there was an inverse view of security and usability.

Unsurprisingly, in response to the question about the relation-

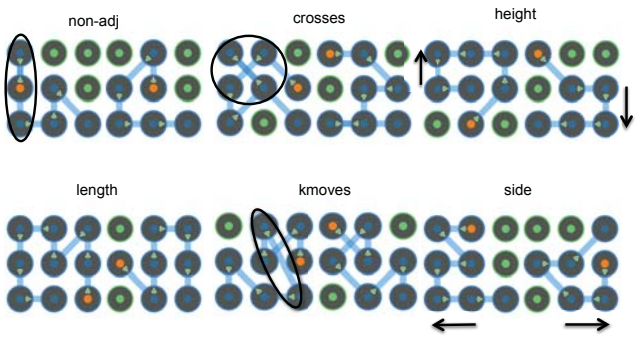


Figure 3: Features tested in the survey

ship between security and usability, 92% of respondents reported that the relationship is inversely related. This is also present in the pairwise data results presented in the following sections. The response to the question about participants who use a graphical password indicates that after viewing many of the passwords selected for the survey, 45% of participants felt that his/her current password was less secure and only 4% felt that it was more secure. *This suggests that a mechanisms that simply shows users a random graphical password before making a selection could improve user choice.* Developing an experiment to test this hypothesis is a focus of future work.

4.2 Selecting Password Pairs

Password pairs were selected to isolate a single feature of a password, such that one password exerts that feature and all other features in the set are similar within some bound. For example, if a password pair was selected for the *length* feature, then one password would be long, connecting more contact points, than the other, but all other features would be similar within a threshold, such as the number of crosses.

Six features were considered in the survey, and examples of those features and pairs are presented in Figure 3. The features are described below with the shorthand inset:

- Length (*length*): The total number of contacts points used in the pattern. This does not consider the total length of the swipes, but in Section 6 the total length of the swipes is characterized.
- Crosses (*crosses*): The total number of times the patterns cross over each other. A subcategory of crosses is *exes*, which is a 90 degree cross.
- Non-Adjacent (*non-adj*): The total number of non-adjacent swipes which occur when the pattern double-backs on itself by tracing over a previously contacted point. For example, starting in the middle contact point, swiping right, then doubling back to the contact point to the left of the initial point.
- Knight-Moves (*kmoves*): The total number of knight-moves which occur when a contact point is connected to another point that is two spaces in one direction and then one space over in another direction, like how a knight moves in chess. These are also referred to as 30 degree swipes [6] in related work.
- Height (*height*): The amount the pattern is shifted towards the upper contact points or the lower contact points.
- Side (*side*): The amount the pattern is shifted towards the left or right contact points.

The goal is to select pattern pairs randomly from all possible patterns, but, unfortunately, many patterns are visually complex at a level where participants could not directly distinguish the patterns and the tested features easily. To counteract this effect, passwords were first classified based on the features above using a simple normalized metric. This rated the prevalence of each feature on the range of [0,1], where 0 indicated the feature was *not* present and 1 indicated that the feature was prevalent at its most extreme. For example, a pattern can have at most 14 crosses in it, so by counting the total number of crosses in a pattern and dividing by 14 provided a simple normalized metric is computed.

Feature	Total Pairs	Strict Pairs
Length	114	53
Crosses	135	35
Non-Adj	151	43
Kmoves	132	44
Height	134	51
Side	139	60
Crosses vs. Non-Adj	100	-
Crosses vs. Kmoves	100	-
Non-Adj vs. Kmoves	100	-
Bristol	8	-

Table 1: Number of Feature Pairs Used in the Survey

To limit the overall visual complexity of the pattern, only patterns whose features were under the 0.7 normalized threshold were considered, except for the features *length*, *height*, or *side* since extremes in those categories are necessary for completeness, for example, to have a pattern completely left or right shifted or of length 8 or 9. Additionally, we limited *kmoves* such that at least one of the intermediary contact points, between which the knight move traverses, must be selected prior. This was argued to increase the usability of the pattern [6, 7]. With these limitations, we selected patterns by randomly sampling from all possible patterns until coverage across features was achieved. In addition to the single feature pairs, we selected a set of pairs that pitted two features against each other for the combination of *kmoves*, *non-adj*, and *crosses* to better assess which features have stronger visual preferences.

Table 1 describes the total number of pairs selected in each category. Due to randomization, we do not have equal coverage across pairs. Additionally, each feature has a strict characterization as well as a general characterization. This is because we wished to identify pairs that generally characterized a difference in the features, for example, a password pair where one pattern has more *crosses* but the two passwords also differed in length by a single contact point. Strict pairs contained passwords that were nearly identical in all features except for the tested feature, and we found that the results are consistent across general and strict pairs.

We acknowledge that the limitation on password selection for the pairs can impact the results; however, the goal of the experiment is to assess the features that inform perceptions of security and usability. Participants who are challenged to quickly recognize patterns due to visual complexity are unlikely to produce informed data, and we felt that decreasing the visual complexity was necessary to this end. Patterns are still selected through a random process, and it should be noted that a number of password pairs contained highly complex patterns which are not likely to be password used in the wild[3].

However, prior studies do provide an opportunity to reevaluate findings in a different setting. In particular, the study by Andriotis *et al.* asked users to first select a password that is secure and one that is usable. We obtained the selected passwords from the re-

[3]One participant did report in non-survey feedback that his/her pattern was part of the dataset.

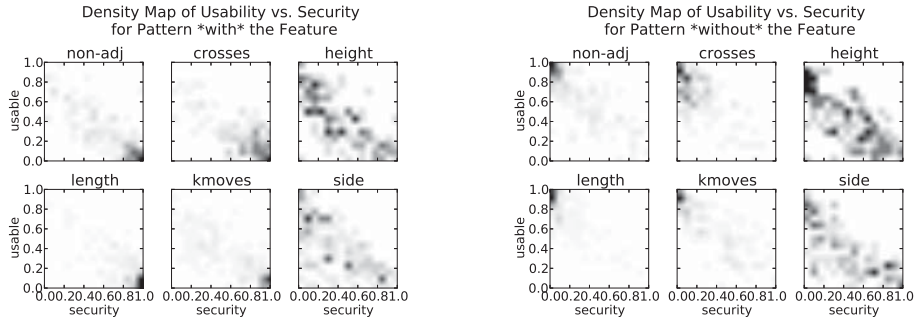


Figure 4: Density maps between security (x-axis) and usability (y-axis) for each features: *left*, considering the pattern in the comparison pair that had the feature; *right*, considering the pattern in the comparison that did have the feature.

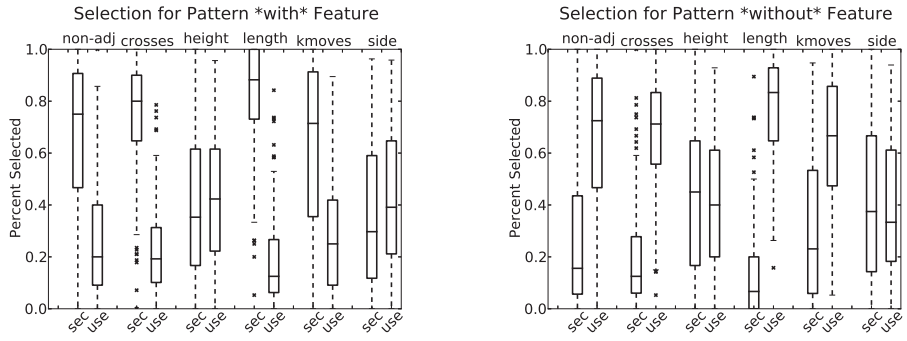


Figure 5: Confidence intervals for percent chosen secure or usable for each pattern that (*left*) has the feature and (*right*) dos not have the feature.

searchers and used a subset of those selections in the survey. Since the study was conducted at the University of Bristol in the UK, we describe these pairs as *Bristol* pairs. We were unable to include password features and n-grams identified by Uellenbeck *et al.* because prior work was not available at the time of the survey.

4.3 Compensation and Reducing Harm

The survey was posted as a MTurk HIT (Human Intelligence Test) twice over the summer of 2013. In the first instance, we compensated participants at a rate of $2.50 for completing the survey without failing the attention and entropy tests. We reject users through MTurk who failed the attention test, which resulted in a number of complaints, including to our IRB. Upon further review, we determined that, yes, these participants should be rejected, but the stigma of MTurk rejections was undue harm to the participant. After re-reviewing the survey with the IRB, we decided to accept all participant through MTurk and release compensation but to remove their data from the data set.

As a result of the complaints, in the second round survey, we changed the compensation to $1.50 for completing the survey — all participants, regardless of the quality of the data received the $1.50 compensation and were accepted within the MTurk ecosystem. We then further reviewed the data, and if a participant passed the attention tests, we provided a bonus of $1.00, equalling the $2.50 compensation from the previous survey. We found that this methodology reduced harm to participants within the MTurk ecosystem.

4.4 Survey Limitations

The described surveys are designed to provide insight into the perceptions of security and usability as queued by different visual features. We believe that this data is informative about user choice habits during password selection; however, there are limitations to this method, and in particular, the fact that perceptions should not be interpreted as a metric for security or usability. While perceptions may correlated with security metrics, unfortunately we are currently unable to directly compare perceptional metrics with those of a true security metric which is in part a consequence of the system being analyzed. There are currently no standard metrics for password strength for the Android password pattern. The closest proposal is described by Uellenbeck *et al.* [27], but its effectiveness remains untested in the field and is based on data unavailable to the authors. In the text based passwords space there is more opportunity as standard metrics are known, and the pairwise preference methodologies of measuring perceptions could be fruitfully applied there, an area of future work. For graphical passwords, however, we see perceptional metrics playing a role in designing selection systems; for example, as discussed in Section 6, generating a ranking of passwords based on perceptions where a user first chooses a password they deem usable, and then are suggested passwords with the same usability perceptions but have some stronger security property. A user will have a password that they perceive as usable but meets a security criteria (perhaps from [27]), and thus will be more likely to use the stronger password.

5. SURVEY RESULTS

In the survey, over 1,100 pairs were rated with a preference assigned to one pasword in the pair for security and usability or same usability/security. Each pair received 19.6 ratings on average, and the combined the overall preferences for a single password is based on the fraction of participants preferring the password. Each password is assigned four preference statistics:

- *sec*, the fraction of participants preferring the pattern more

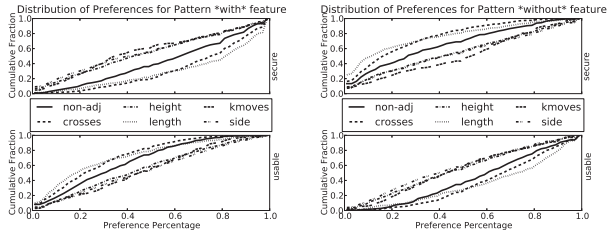


Figure 6: Cumulative fraction distribution of percent selected secure (top) and usable (bottom) for the pattern with the feature (left) and without the feature (right).

secure in the pair;

- *use*, the fraction of participants preferring the pattern more usable;
- *samesec*, the fraction of participants preferring neither pattern as more secure;
- *sameuse*, the fraction of participants preferring neither pattern as more usable.

Note, that for a given password pair, (p_1, p_2):

$$sec(p_1) + sec(p_2) + samesec(p_1, p_2) = 1$$
$$use(p_1) + use(p_2) + sameuse(p_1, p_2) = 1$$

Each password pair is selected to test a visual feature, and one password has the feature, while one does not. This distinction is indicated with the keywords *with*, indicating results for the password *with* the feature, and *without*, indicating results for the password in the pair *without* the feature.

5.1 Pairwise Preference

The most straight forward analysis of pairwise preferences is to measure the relationship between usability and security preferences for each password pair based on the tested feature. This comes in two flavors: an assessment of usability and security preferences for the passwords *with* the feature and a complimentary assessment for the passwords *without* the feature. Note that *with* and *without* are not inverses because participants can indicate *same* preference if neither password is preferred under the criteria.

Figure 4 presents a density image map of the distribution of security versus usability for each visual feature tested, both for passwords *with* the feature (*left*) and for passwords *without* the feature (*right*). Density maps with higher concentrations in the lower right corner indicate a stronger preference for security, while density maps with a higher concentration in the upper left corner indicate a stronger preference for usability.

Of the tested features, *non-adj*, *crosses*, *kmoves* and *length*, the passwords *with* the feature demonstrate a clear preference for security while passwords *without* the feature demonstrate clear preference for usability. For these features, there is a strong inverse relationship between usability and security. The spatial features, *height* and *side*, however, do not exhibit an inverse relationship and have scattered densities along the diagonal.

These observations continue in Figure 5, which is a box-and-whisker graph of the distribution of preferences for passwords *with* and *without* the feature. Here, the inverse relationship of *non-adj*, *crosses*, *kmoves*, and *length* is even more prevalent. For example, the difference in the median between usability and security for passwords *with* the *length* feature is extreme at 0.75, and, at the same time, the difference in the median preference for *side* and *height* are nearly identical for usability and security both *with* and *without* the feature, varying at most 0.09.

Feature	With-Pref	Feature	Without-Pref
length	24.36	height	3.22
kmoves	22.12	side	-2.16
crosses	21.75	length	-17.28
non-adj	20.56	non-adj	-22.07
height	-2.27	crosses	-22.60
side	-5.55	kmoves	-23.80

Table 2: Security versus usability Preferences for patterns with and without the feature as measured by the area between the cumulative fraction distributions: Left the increased security preference, and right, the negative security preference.

5.2 Preference Ranking Metric

We wished to develop a ranking metric to better understand the usability and security preferences of the feature. To do so, we first generated the cumulative fraction distribution for each feature, which considers the fraction of passwords that receive the preference, or higher. The cumulative fraction graphs are presented in Figure 6. Distributions that are shifted towards the lower right corner, or *cupped*, suggest a stronger usability/security preference because a higher fraction of passwords receive stronger ratings in the category. Similarly, distributions that are shifted towards the upper left corner, or *capped*, suggest a weaker usability/security preference since a larger fraction of passwords received weaker preferences in the category.

In the left graphs of Figure 6, we generate the cumulative fraction graphs (cfg) for the passwords *with* the feature, the security cfg on top and the usability cfg on the bottom. The ranking metric for the features is based on measuring the area between the usability cfg and the security cfg: the more strongly capped the usability cfg is and the more strongly cupped the security cfg is, the greater the security preferences are for that feature. Table 2 presents the results of that measure. For the *with* passwords, the most strongly preferred secure feature is *length*, the number of contact points used in the pattern, and all major non-spatial features also exhibit a strong security preference.

Interestingly, for the *without* passwords, this metric can be seen as how much the *lack of* the feature reduce the security preference for the password. We describe this as the *negative security preference* since it negatively impacts the perceived security of the password in the pair. The ordering of negative security preferences are not consistent with the prior result. The *kmoves* feature exerts the strongest negative security preference, and *length* has the weakest negative security preference. This implies that of the features tested increasing the length of the password increases the security preferences but a lack of length does not strongly decrease security preference as compared to other features, like a lack of *kmoves*. In the Section 6, a broader set of features is explored in relation to user preference, and confirms that *length* is a strong indicator of perceived security and usability, but an additional feature, the password distance, or the length of the individual strokes, is an even stronger indicator than *length* alone.

5.3 Comparing Feature Preferences

The previous results considered the tested features in isolation, that is, a single pattern in the pair has the feature while the other does not. Now, we consider features in comparison to each other where within a pair one pattern has a feature and the other has a different feature with all other aspects being equal.

To start, the effect of the *length* feature is compared to other tested features. Recall that the data set contains both strict pairs and general pairs. A strict pair is a pure comparison with only one feature changed. A general pair may have all features being the

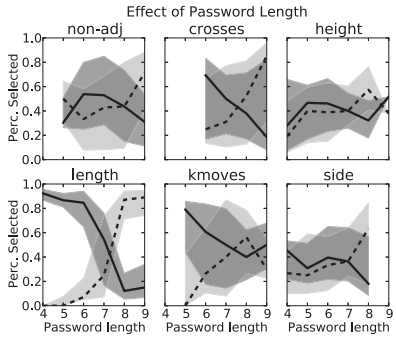


Figure 7: Length Correlation: Effects of usability (dashed line) and security (solid line) for increasing length. The shaded region is the first and third quartile.

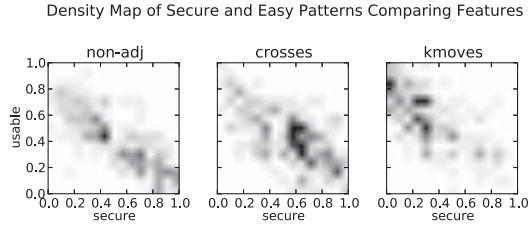


Figure 8: Density distribution of security and usability preferences for pairs of patterns that compared two features.

same, but have one feature vary slightly under the threshold, such as *length*. We investigated all general pairs that varied in length to determine if *length* was a singular driver in preference. These results are presented in Figure 7. The y-axis in each subplot is the length of the pattern; the solid line is the median security preference; the dashed line is the median usability preference; and the shaded region is the first and third quartile.

For some of the features, a clear inverse relationship between length and usability/security preference is present. Most demonstrable is the *length* feature itself; however, the strength of this relationship is not present for all features. Notably, both *non-adj* and *kmoves* do not have as strong an inverse relationship — as length increases, security increases and usability decreases — and the spatial features *height* and *side* have almost no relationship to length, which is aligned with prior findings. This suggests that while the length of the password is a major contributor, participants are sensitive to the other tested feature.

We further explored the relationship between features by pitting the non-spatial features, excluding *length*, in a pairwise preference comparison. The density map of those results are presented in Figure 8. From these results, we can see that *non-adj* is weakly shifted towards usability when compared to *crosses* and *kmoves*, while *crosses* are considered more secure in comparison. The *kmoves* feature, interestingly, was the usable preferred of the features tested, clearly shifted towards the upper left.

5.4 Comparison to Bristol Studies

Included in the pairwise features was passwords that were selected by participants from a related study by Andriotis *et al.* at the University of Bristol, UK [3]. In that study, participants were asked to provide one password that is "secure" and one password that is "easy." We wished to test if user generated passwords purposely selected in a similar usable/secure criteria would be preferred in the same way within our study. A set of 8 password pairs from the *Bristol* study were included in the pairwise preference study and those results are presented in Figure 9. There does exist a secure preference for the user selected secure choice, as well as usable

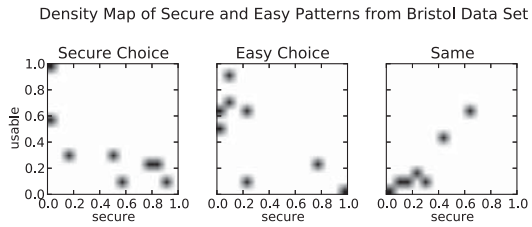


Figure 9: Bristol Data: Usability and Security preferences of data collected in the Bristol study.

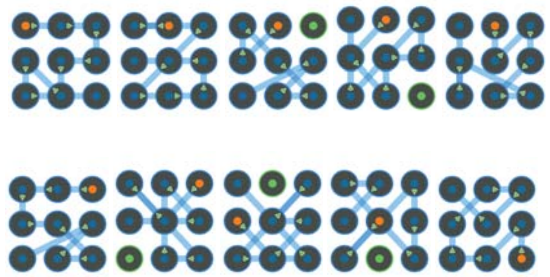


Figure 10: Top 10 highest rated passwords for most "Highly Usable" and "Highly Secure." The highest rated is in the top left, and moving left-to-right by row, the tenth, highest rated is the lower right

preference for the user selected easy choice. There are significant outliers, however, at the opposite extremes. This experiment does not draw direct conclusions on the validity of the Bristol study, but does suggest that user provided data for usability studies can contain inconsistencies. The pairwise preference survey could provide a way to validate prior results and expand on those studies.

5.5 Individual Password Ratings

Finally, we report on the results of the password rating survey. Recall that participants were directed to provide a rating for individual passwords in addition to pairs of passwords, indicating the password's symmetry, any apparent shapes, a usability assessment, and a security assessment. Of most interest to this study is the rating of security and usability, which we can place on a range [0,3], where 0 indicates "Highly Insecure" or "Highly Unusable" and 3 indicates "Highly Secure" and "Highly Usable." A rating of 1.5 would be neutral with respect to usability and security. The rating was compiled by taking the average response from all ratings by participants for a given password. We only included passwords that received at least 5 ratings in these results.

Participants identified a large portion of passwords used in the study as at least "secure" and "usable" as indicated by an average score of at least 2 in both categories. To further understand this result, the top 10 highest rated secure *and* usable patterns in our data set of approximately 2,000 passwords were identified and presented in Figure 10, with the number one most secure and usable password in the upper left and the number ten in the lower right, moving left-to-right by row. The set of features in these top ten patterns runs the gamut, but all have a length of at least 7, again demonstrating the impact of length on perception of security. While these patterns apear complex, they all received positive usability scores, suggesting that *users would be open to using a wider variety of pattern forms than they might use if selecting independently.* A mechanism that simply shows users randomly selected passwords may increase the overall security of user choice, as these results and the exit survey demonstrates.

5.6 Perceptions of Symmetry

Finally, the relationship between perceived password symmetry and usability/security is measured. Recall that participants were directed to rate "Yes" or "No" if the pattern "appeared to be symmetric." We looked at the correlation of the average rating, on the scale for [0,1] where 1 indicate symmetry and 0 is the lack thereof, with the security and usability rating. We applied the standard Pearson product-moment correlation coefficient, and found that symmetry and security correlation have an $r = -.249$ while symmetry and usability have an $r = 0.058$. This result indicates that users perception of security is negatively impacted by perceived symmetry, but symmetry does not have a strong effect on perceived usability. In some ways, this is contrary to studies that suggest symmetry is a strong indicator of memory in psychology [5, 15] and usability studies for the Draw-a-Secret graphical password scheme [25].

These results could be explained by how participants associated symmetry with guessability of a passwords and usability with ease-of-entry. For example, these results might imply that users see symmetric password as something that an attacker would likely guess before an asymmetric password, thus decreasing the perceived security. When interpreting usability as ease-of-entry, it is also inline that symmetry has little impact on usability because, in the end, the user must still enter the password and even symmetric passwords can be hard to enter. This is consistent with related work on symmetry and mnemonics [5, 15]; a password could be easy to remember but still hard to enter.

6. PREDICTING PREFERENCES

In this section, we move beyond presenting the direct empirical results of the survey and instead broaden the focus by expanding the set of passwords features that may affect preference. The following experiments involves constructing a logistic regression model based on a set of extended features to measure their predictive power in anticipating a user's security or usability preference.

6.1 Extended Features

For these experiments, the set of features were extended beyond those surveyed to include metrics presented in prior studies [3, 4, 27]. This larger set of features were used to train a logistic regression model on training data randomly sampled, 900 pairs from the 1,100 collected, and the remaining data was withheld as testing. Below, the additional features and their characteristics are explained, as well as where the feature was used in prior work.

- Ex-Crosses (*exes*): the number of exes within corners of the patterns, not just general crosses.
- Starting Point: (*start*) : the start contact point (as used in [3, 27])
- Ending Point (*end*): the end contact point (as used in [3, 27])
- Distance (*dist*): total geometric distance of the strokes in the pattern, not just the length, as calculated by the total points used [4].
- Point Entropy (*pfreq*): the entropy of the points used in the password as calculated from the total distribution of point found in all patterns [4].
- Stroke Entropy (*sfreq*): the entropy of pairwise points in the pattern (or strokes) as calculated from the total distribution of strokes found in all patterns [4].
- Vertical Symmetry (*vsym*): the vertical symmetry of the pattern with respect to mirroring the pattern top to bottom including both points and strokes [4, 21].
- Horizontal Symmetry (*hsym*): the horizontal symmetry of the pattern with respect to mirroring the pattern left to right including both points and strokes [4, 21].

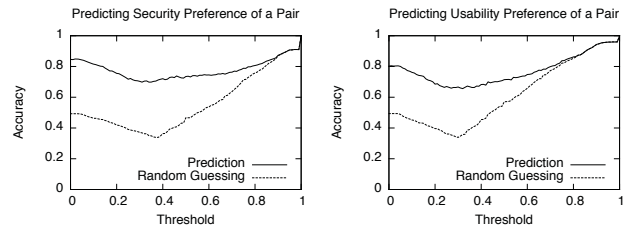


Figure 11: Predicting rates for selecting the preferred password in a pair above a threshold as compared to random guessing.

Each password in the pair received values based on the features above as well as the normalized feature values used in pair selection for the survey (*length*, *kmoves*, *crosses*, *non-adj*, *height*, *side*), totalling 13 different features. Features were assigned to pairs based on the difference between the individual features of the two passwords, and labeling occurred per pair (see below).

6.2 Preference Labeling

Each pair received a label from three values -1, 0, 1, where a -1 indicated that the left item in the pair was preferred, 0 indicating that neither was preferred, or 1 indicating the right item was preferred. Labels were assigned based on a threshold difference in preference: if $sec(p_1) - sec(p_2) > \tau$ a label of -1 is assigned to indicate a preference for p_1, or if $sec(p_2) - sec(p_1) > \tau$ a label of 1 is assigned to indicate a preference for p_2, and if neither is true, then a label of 0 is applied to indicate no strong preference.

There is a range of choices for the threshold with tradeoffs, and to assess these tradeoffs, we ran a five-fold cross-validation over the training set with the thresholds ranging from 0 to 1. The results of the experiment is presented in Figure 11 with a comparison to random guessing (i.e., always selecting the most common label). At a threshold of approximatively 0.33, random guessing is at a minimun because the number of samples with each label is roughly equal. We use this threshold going forward because this represents the most challenging predictive scenario. Additionally, the 0.33 threshold is a natural choice because it requires that one password in the pair have a preference at least twice as great as the other to receive a preference label other than 0.

6.3 Prediction Results

At the 0.33 threshold rating, the logistic model can accurately separate the data for a security preference at a rate of 70% and usable preference at a rate of 66%, which is twice the rate of random guessing in both instances. With lower thresholds, such as threshold of 0.0 which would label the data with -1 or 1 if there is any difference in preference between the patterns, the model can accurately separate the data at a rate of 80% or higher for both usability and security which is significantly greater than random guessing, which is just below 50%.

The final consideration is to analyze which of the features are the strongest predictors of preference. To measure this, we used the 0.33 threshold value and constructed separate models using only a single feature from the set. The results of that experiment is presented in Figure 12. One might expect, as before, that the features used in the survey would be the strongest indicator as these were the features used when selecting pairs. As indicated, the *length* feature continues to be a strong predictor of user preference for both usability and security, but the *dist* feature is by far the strongest feature followed by *crosses* and then *length*. In some ways, this is intuitive. The total distance of a pattern, which is a measure of

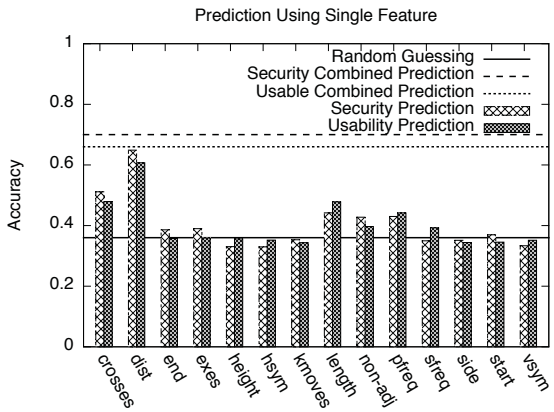


Figure 12: Predictions using single feature from the set.

the length of all the lines, is a good indicator of visual complexity because to add any of the other tested features, such as *non-adj*, *kmove*, *crosses* and *length*, requires increasing the distance of the pattern, by definition, and thus increased pattern distance results in increased visual complexity. This result is also important because it is indicative of the visual calculus that users apply when determining a preference for usability and security, and it can be leveraged in designing new visual security systems as well as aides, like password meters, to help user selection of patterns.

Other findings of this experiment are interesting in that they indicate that the more theoretic metrics used in prior work [4] have little impact on pairwise preferences. This includes metrics involving symmetry; neither horizontal nor vertical symmetry can predict a preference above random guessing. Information theoretic metrics were more capable, including *pfreq* and *sfreq*, but primarily the entropy of contact point distributions seems to be a more reasonable predictor of preference. Finally, spatial metrics, again, fail to strongly correlate with preference, including the start and end point, which were highlighted in prior studies [3, 27] as having strong biases in user-selected passwords.

7. CONCLUSION

We presented the results of a large user study of pairwise preferences for usability and security of the Android password pattern which provides insights into user perceptions that inform choice. Pairs of patterns were selected based on six visual features, and we concluded that spatial features, such as shifting to the side or up/down, had little impact on a preferred pattern in the pair. More visually striking features have a stronger impact, with the length of the pattern being the strongest indicator of preference. These results were extended and applied by constructing a predictive model with a broader set of features from related work, and we found that the distance feature, the total length of all the lines in a pattern, is the strongest predictor of preference. Our findings provide insight into users' visual calculus when assessing a password, and this information could be leveraged to develop new password systems or user selection tools, like password meters. Moreover, with a good predictive model of user preference, it can be applied to a broader set of passwords, including those not used in the survey. Ranking data based on learned pairwise preferences is an active research area in machine learning [11], and the resulting rankings metric over all potential patterns in the space would be greatly beneficial to the community. It could enable new password selection procedures where users are helped in identifying a preferred usable password that also meets a security requirement.

References

[1] John the ripper. `http://www.openwall.com/john/8`.

[2] The Password Project. `http://thepasswordproject.com`.

[3] Panagiotis Andriotis, Theo Tryfonas, George Oikonomou, and Can Yildiz. A pilot study on the security of pattern screen-lock methods and soft side channel attacks. In *Proceedings ACM Conference on Security and privacy in Wireless and Mobile Networks*, WiSec, 2013.

[4] Majid Arianezhad, Douglas Stebila, and Behzad Mozaffari. Usability and security of gaze-based graphical grid passwords. In *Financial Cryptography and Data Security*, pages 17–33, 2013.

[5] Fred Attneave. Symmetry, information, and memory for patterns. *The American journal of psychology*, 68(2):209–222, 1955.

[6] Adam J. Aviv, Katherine Gibson, Evan Mossop, Matt Blaze, and Jonathan M. Smith. Smudge attacks on smartphone touch screens. In *Proceedings of the 4th USENIX Workshop On Offensive Technologies*, WOOT'10, 2010.

[7] Adam J. Aviv, Ben Sapp, Matt Blaze, and Jonathan M. Smith. Practicality of accelerometer side-channels on smartphones. In *Annual Computer Security Applications Conference (ACSAC)*, 2012.

[8] Robert Biddle, Sonia Chiasson, and Paul C Van Oorschot. Graphical passwords: Learning from the first twelve years. *ACM Computing Surveys*, 44(4):19, 2012.

[9] Joseph Bonneau. The science of guessing: analyzing an anonymized corpus of 70 million passwords. In *IEEE Symposium on Security and Privacy (SP)*, 2012.

[10] Joseph Bonneau, Sören Preibusch, and Ross Anderson. A birthday present every eleven wallets? the security of customer-chosen banking pins. In *Financial Cryptography and Data Security*, pages 25–40. Springer, 2012.

[11] Xi Chen, Paul N Bennett, Kevyn Collins-Thompson, and Eric Horvitz. Pairwise ranking aggregation in a crowdsourced setting. In *Proceedings of International Conference on Web Search and Data Mining*, pages 193–202, 2013.

[12] Sonia Chiasson, Robert Biddle, and Paul C van Oorschot. A second look at the usability of click-based graphical passwords. In *Proceedings of the Symposium on Usable Privacy and Security*, SOUPs, pages 1–12, 2007.

[13] Matteo Dell'Amico, Pietro Michiardi, and Yves Roudier. Password strength: an empirical analysis. In *Proceedings of IEEE INFOCOM*, pages 1–9, 2010.

[14] Serge Egelman, Joseph Bonneau, Sonia Chiasson, David Dittrich, and Stuart Schechter. It's not stealing if you need it: A panel on the ethics of performing research using public data of illicit origin. In *Financial Cryptography and Data Security*, pages 124–132. Springer, 2012.

[15] Robert Stanton French. Identification of dot patterns from memory as a function of complexity. *Journal of Experimental Psychology*, 47(1):22, 1954.

[16] Kirsi Helkala and Nils Kalstad Svendsen. The security and memorability of passwords generated by using an association element and a personal factor. In *Information Security Technology for Applications*, pages 114–130. Springer, 2012.

[17] Ian Jermyn, Alain Mayer, Fabian Monrose, Michael K Reiter, Aviel D Rubin, et al. The design and analysis of graphical passwords. In *Proceedings of the 8th USENIX Security Symposium*, pages 1–14. Washington DC, 1999.

[18] Patrick Gage Kelley, Saranga Komanduri, Michelle L Mazurek, Richard Shay, Timothy Vidas, Lujo Bauer, Nicolas Christin, Lorrie Faith Cranor, and Julio Lopez. Guess again (and again and again): Measuring password strength by simulating password-cracking algorithms. In *IEEE Symposium on Security and Privacy (SP)*, pages 523–537, 2012.

[19] Michelle L Mazurek, Saranga Komanduri, Timothy Vidas, Lujo Bauer, Nicolas Christin, Lorrie Faith Cranor, Patrick Gage Kelley, Richard Shay, and Blase Ur. Measuring password guessability for an entire university. In *Proceedings of the Conference on Computer & Communications Security (CCS)*, 2013.

[20] Robert Morris and Ken Thompson. Password security: A case history. *Communications of the ACM*, 22(11):594–597, 1979.

[21] PC van Oorschot and Julie Thorpe. On predictive models and user-drawn graphical passwords. *Transactions on Information and System Security*, 10(4):5, 2008.

[22] Jeffrey M Stanton, Kathryn R Stam, Paul Mastrangelo, and Jeffrey Jolton. Analysis of end user security behaviors. *Computers & Security*, 24(2):124–133, 2005.

[23] Xiaoyuan Suo, Ying Zhu, and G Scott Owen. Graphical passwords: A survey. In *Annual Computer Security Applications Conference (ACSAC)*, 2005.

[24] Hai Tao and Carlisle Adams. Pass-go: A proposal to improve the usability of graphical passwords. *IJ Network Security*, 7(2):273–292, 2008.

[25] Julie Thorpe and Paul C van Oorschot. Graphical dictionaries and the memorable space of graphical passwords. In *USENIX Security Symposium*, 2004.

[26] Julie Thorpe and Paul C van Oorschot. Human-seeded attacks and exploiting hot-spots in graphical passwords. In *USENIX Security Symposium*, 2007.

[27] Sebastian Uellenbeck, Markus Dürmuth, Christopher Wolf, and Thorsten Holz. Quantifying the security of graphical passwords: the case of Android unlock patterns. In *Conference on Computer & Communications Security (CCS)*, 2013.

[28] Blase Ur, Patrick Gage Kelley, Saranga Komanduri, Joel Lee, Michael Maass, Michelle L Mazurek, Timothy Passaro, Richard Shay, Timothy Vidas, and Lujo Bauer. How does your password measure up? the effect of strength meters on password creation. In *Proc. USENIX Security*, 2012.

[29] Ziming Zhao, Gail-Joon Ahn, Jeong-Jin Seo, and Hongxin Hu. On the security of picture gesture authentication. In *USENIX Security Sympsosium*, 2013.

[30] Moshe Zviran and William J Haga. Password security: an empirical study. *Journal of Management Information Systems*, 15:161–186, 1999.

It's the Psychology Stupid: How Heuristics Explain Software Vulnerabilities and How Priming Can Illuminate Developer's Blind Spots

Daniela Oliveira Marissa Rosenthal[†] Nicole Morin[†]
Kuo-Chuan Yeh[*] Justin Cappos[‡] Yanyan Zhuang[‡]
University of Florida Bowdoin College[†] Pennsylvania State University[*] NYU[‡]
daniela@ece.ufl.edu, mrosenth,nmorin@bowdoin.edu, yeh@cse.psu.edu, jcappos,yyzh@nyu.edu

ABSTRACT

Despite the security community's emphasis on the importance of building secure software, the number of new vulnerabilities found in our systems is increasing. In addition, vulnerabilities that have been studied for years are still commonly reported in vulnerability databases. This paper investigates a new hypothesis that software vulnerabilities are blind spots in developer's heuristic-based decision-making processes. Heuristics are simple computational models to solve problems without considering all the information available. They are an adaptive response to our short working memory because they require less cognitive effort. Our hypothesis is that as software vulnerabilities represent corner cases that exercise unusual information flows, they tend to be left out from the repertoire of heuristics used by developers during their programming tasks.

To validate this hypothesis we conducted a study with 47 developers using psychological manipulation. In this study each developer worked for approximately one hour on six vulnerable programming scenarios. The sessions progressed from providing no information about the possibility of vulnerabilities, to priming developers about unexpected results, and explicitly mentioning the existence of vulnerabilities in the code. The results show that (i) security is not a priority in software development environments, (ii) security is not part of developer's mindset while coding, (iii) developers assume common cases for their code, (iv) security thinking requires cognitive effort, (v) security education helps, but developers can have difficulties correlating a particular learned vulnerability or security information with their current working task, and (vi) priming or explicitly cueing about vulnerabilities *on-the-spot* is a powerful mechanism to make developers aware about potential vulnerabilities.

1. INTRODUCTION

Over the past decades, the security community has spent tremendous effort in emphasizing security awareness and building secure software. However, the number of new vulnerabilities keeps increasing in today's software systems. In 2013, the Symantec Internet Security report has announced that 5291 *new* vulnerabilities occurred in 2012, 302 more than in 2011 [1]. Despite the fact that vulnerabilities have been the focus of the security community for decades, frequently observed vulnerabilities such as buffer overflows and SQL injections are still repeatedly reported. With today's increasingly diverse software, and a society significantly dependent on networked computer systems, the inability to effectively handle software vulnerabilities will result in more serious security breaches in the future.

Facing this huge number of security vulnerabilities, the research community has chosen to criticize the current security education and developers. As an example, SQL injection has caused issues that commonly lead to password database disclosures. When referring to the cause for SQL injection, Bruce K. Marshall stated that *"[T]he popularity of the language has led to the rapid deployment of PHP sites and PHP-based content management systems by people who lack an education in web application security. Even though the risk of SQL injection in PHP should be fairly well understood, some organizations still end up deploying code that doesn't implement proper security controls"* [2]. The frequent condemnation of security education and criticism on software developers, however, do not help to reason about the root causes of security vulnerabilities.

We argue that the nature of increasingly insecure software with well-studied vulnerabilities does not lie in the lack of security education from the developer's part. We seek to examine the hypothesis that software vulnerabilities are *blind spots* in developer's heuristic-based decision-making processes. Heuristics are computational models that do not use all information available to reach a particular decision or course of action. During their everyday programming tasks, developers use heuristics, either consciously or unconsciously, that do not include security or vulnerability information. As software vulnerabilities represent uncommon cases not completely understood by developers and exercise unusual information flows, they are usually left out from developer's heuristics. In spite of that, decision-making is adaptive. Therefore, another hypothesis we investigate in this paper is whether priming developers *on the spot* about the possibility of vulnerabilities will change their mindset towards security and make security-thinking part of their repertoire of heuristics.

Despite the abundant security tips for safe programming [3], developers may not be considering these security practices when writing code in their daily tasks. Psychological research documents that humans often engage in heuristic-based decision-making processes that are due to the limitations in a human's working memory capacity [4, 5]. Heuristics occur when a human is facing complex problems with a large amount of information, and thus she tends to make simplified, sub-optimal decisions regardless of the rich information available [6, 7]. Although heuristics is an adaptive tool, they can lead to software faults and deleterious consequences. Seeking security-related information while coding is usually not the case in software development. When programming, developers tend to focus on their immediate goals that usually involve functional and performance requirements. As a result, they do not expect the possibility of an adversary exploiting their code [8].

To validate this hypothesis, we conducted an IRB-approved study with 47 participants from a variety of backgrounds, including de-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.
ACSAC'14 December 08 - 12 2014, New Orleans, LA, USA
Copyright 2014 ACM 978-1-4503-3005-3/14/12...$15.00
http://dx.doi.org/10.1145/2664243.2664254.

velopers from the industry, CS undergraduate and graduate students, CS faculty, software developer managers, and software testers. The participants were asked to work on six programming scenarios with software vulnerabilities in a survey session. The session lasted approximately one hour. Our study leveraged psychological manipulation [9] where participants did not know in advance that the study was security-related, and were *progressively cued* about possible vulnerabilities. The goal was to validate the hypothesis that security information is usually not part of developers' heuristic thinking during their everyday programming; however, security information can become part of their repertoire of heuristics when developers are cued about the possibility of vulnerabilities. Analysis of participant's answers showed that stronger cues have stronger effect on developers.

In summary, this paper has the following contributions:

1. We present a hypothesis that views software vulnerabilities as blind spots in developers' heuristic-based decision-making processes, and propose a paradigm that advocates security information and education should reach developers when they need it, *on the spot*.
2. We conduct an IRB-approved study with 47 developers to validate our hypothesis and show that 60% of the developers thought that when primed about security they become aware of the security implications of their code, and 83% of the developers thought that explicitly mentioning the possibility of vulnerabilities on the spot changed their mindset towards security.
3. We present a thorough discussion of developers' interview answers that are analyzed with techniques widely used in the social, behavioral and economic sciences. The results show that (i) security is not a priority in software development environments, (ii) security is not part of a developer's mindset while coding, (iii) developers assume common cases for their code, (iv) security thinking requires cognitive effort, (v) security education helps, but developers can have difficulties correlating particular learned vulnerability or security information with their working task, and (vi) priming or explicitly cueing about vulnerabilities *on-the-spot* is a powerful mechanism to make developers aware of the potential vulnerabilities.

This paper is organized as follows. Section 2 describes humans' heuristic-based decision making processes and how they relate to software vulnerabilities. Section 3 presents the study method to validate our hypothesis. In Section 4 we detail the scenarios presented to the developers during our study, and the use of psychological manipulation. Section 5 presents the results obtained, and the Section 6 discusses these results and provides our recommendations for developers. Section 7 gives an overview of related work and Section 8 concludes.

2. HEURISTICS AND VULNERABILITIES

Psychology research has documented that during evolution, humans have become hardwired for shortcut and heuristic-based decision making processes [4, 5]. Heuristics are cognitive processes that humans use to make decisions and perform tasks [10]. They are simple computational models that allow one to quickly find feasible solutions and that do not necessarily use all information available. Heuristics rely on core human mental capacities, such as recognition, recall and imitation [11]. They represent an alternative to optimization models that use all information available and always compute the best solution.

As psychological processes, heuristics are very useful as they require less cognitive effort for a particular task. Humans have a short working memory, which makes cognitive processes difficult when too much information, possibilities, or choices are available [12]. In such cases, humans employ sub-optimal decision-making processes that can lead to mistakes [6, 7]. This argument is reinforced by Zipf's principle of least effort [13], which states that humans use as little effort as necessary to solve a problem. Heuristics are adaptive responses to human's short working memory. They have high predictive accuracy when information is scarce, but can lead to severe biases and errors in decision making or ensuring the correctness of tasks [10, 11].

Such heuristic-based decision-making processes also largely affect software security. According to Thorngate, humans tend to ignore information in heuristics because they do not notice certain issues of a particular problem, or there are small or infrequent decrements in reward that result from their ignorance or misuse of relevant information about the problem in hand [10]. In software development, this is reflected by the fact that functional and performance requirements usually have higher priority. Kieskamp and Hoffrage [14] also argue that under time pressure, a common situation in software development, humans are likely to adopt heuristics that are even simpler, and do not require much integration of information. In spite of that, a decision-maker is adaptive such that through proper feedback they can improve their repertoire of heuristics. However, there also exists a forgetting process where a particular piece of knowledge or strategy can be gradually wiped out from the decision-maker's repertoire of heuristics, if not properly reinforced.

Our hypothesis is that software vulnerabilities are blind spots in developer's heuristic-based decision-making processes. Software vulnerabilities are introduced mostly because developers use heuristics to make decisions in their everyday tasks. When developers constantly make sub-optimal decisions, consciously or unconsciously, they are mostly concerned about finding *a* solution or an efficient solution to a particular problem. However, as software vulnerabilities often lie in the corner cases and unusual information flows, they tend to be left out from developers' heuristics. However, when properly primed about security, *on the spot* and with cues correlating with their current programming tasks, developers can properly develop a security mindset for the task at hand.

3. DEVELOPER'S STUDY METHOD

Forty-seven participants were tested in this study in exchange for a gift certificate. Participants were invited via direct e-mail sent to software development companies, universities and colleges in the United States and abroad.

3.1 Source Materials

Stimuli consisted of programming scenarios that included software vulnerabilities and that tested developer's understanding on the issue. The vulnerabilities in each programming scenario were: buffer overflow [15], cross-site scripting (XSS) [16], SQL injection [17], Python Secure Socket Layer (SSL) [18], time of check to time of use (TOCTTOU) [19], and brute force password exhaustion vulnerabilities [20]. Each scenario is an *exhibit question* in Qualitative Research [21], which sharpens the respondents' concentration by asking them to respond to a specific statement, story or artifact. A scenario is a short code snippet with comments that formulates the underlying vulnerability in an focused way, exercising it, adding all necessary context, and removing unnecessary noise. Noise was removed in a careful way, as vulnerabilities are often located in hidden cases and also in unusual information flows. Example programming scenarios will be given in Section 4.

3.1.1 Information Conditions

The scenarios are presented to developers with different information conditions: (i) with no information about security or vulnerabilities (controlled condition), (ii) with implicit information about possible unexpected results in code (priming condition), and (iii) with explicit information about the existence of vulnerabilities in code (explicit condition). Examples of these information conditions in our study are given in Section 4.

3.1.2 Pilot Study

Before inviting developers and distributing the survey, a pilot study was performed with undergraduate students from one of the

author's institutions. In this study, students were asked to answer questions for each scenario and provide feedback in a live follow-up interview about their experience. Follow-up interviews were tape-recorded and included a thorough debrief on each vulnerability. Based on their feedback, the scenarios were adjusted to maximize their clarity. An online survey was then created that consisted of the six finalized scenarios, followed by open-ended questions that reflected the interview questions in the pretest. Stimulus presentation and the collection of responses were controlled using Qualtrics platform for online data collection [22].

3.2 Procedure of Study

Participants were instructed to answer questions in the survey as accurately and thoroughly as possible, without knowing that the study was security-related. To ensure that subjects were not suspicious of the aim of our study, the survey employed psychological manipulation techniques [9] that involved a cover story presented in a consent form signed by the participants. The cover story explained that this was a study of how developers think about programming in general, so that researchers of this study could build mental models about typical developers. They were informed that in the survey, there would be six programming scenarios and general questions about snippets of code, and they would be asked to perform small programming tasks to modify the code.

Subjects were not restricted to a particular time frame to complete the survey. However, they were informed that once they completed one scenario, they would not be able to return to previous pages. Following the last scenario, subjects responded to open-ended questions that addressed the manipulation of information, the subjects' personal experience with computer security, and their familiarity with each of the tested vulnerabilities. The examples of open-ended questions are given in Section 4. To leverage the educational purposes of the study, subjects were debriefed on each security vulnerability.

3.3 Design

Psychological research experiments typically have both dependent and independent variables. In this study, the dependent variables are the participants' survey scores and their answers to the open-ended debrief questions. There are also two independent variables that are varied and manipulated by the experimenter in this study. The first one was the vulnerability scenario of a code snippet. The second is the level of information (condition) emphasized by each question and could be no information, implicit or explicit information. In a psychology experiment, researchers are interested in how the changes in an independent variable cause changes in the dependent variable [9]. Two one-way Analyses of Variance (ANOVAs) [23] across the three levels of information and six vulnerability scenario types were conducted on data accuracy.

Scenarios in the controlled and priming (implicit) conditions contained questions that addressed user input, code execution, and code modification. In the priming condition, subjects answered an additional question that asked whether unexpected results could arise from the presented snippet of code. This question aimed to prime participants to think about security-related vulnerabilities. In the explicit condition, subjects were cued to look for the vulnerability and were directly told that the code had a security flaw.

The order of the scenarios was randomized for each participant. However, because the manipulation intended for subjects to change their mindset, information conditions, or whether the scenario was in the controlled, implicit, or explicit condition, were presented to all participants in a specific order. The first two scenarios provided no information about the corresponding security flaws (controlled condition), the second two scenarios provided implicit information (priming) about the corresponding security flaws, and the last two scenarios explicitly addressed the fact that the snippets of code had security flaws (explicit).

```
1.  <? php
2.  session_start();
3.
4.  if(!isset($userinfo[$_POST['username']])) {
5.       // Invalid username
6.       echo "Authentication failed."
7.  }
8.  else {
9.
10.      if ($userinfo[$_POST['username']] ==
11.                  md5($_POST['password'])) {
12.          // Successful authentication
13.          echo "Welcome to Bank ABC."
14.      }
15.      else {
16.          // Invalid login
17.          echo "Login failed."
18.      }
19. }
20. // rest of the script
21. ?>
```

Figure 1: Brute force password exhaustion vulnerability scenario.

4. THE VULNERABILITY SCENARIOS

Each developer session exercised six vulnerabilities, where five are well-known and studied by the security community for years and one is relatively new, initially reported in vulnerability databases in 2013. In this section, we describe our IRB-approved study with these vulnerabilities, our psychological manipulation through different information conditions, and how the results are measured.

The five well-know vulnerabilities are buffer overflow [15], cross-site scripting (XSS) [16], SQL injection [17], time-to-check-to-time-to-use (TOCTTOU) [19], and finally an authentication vulnerability which can lead to brute-force password or dictionary attacks [20].

Figure 1 illustrates the brute force authentication vulnerability scenario. Developers were asked to consider a hypothetical banking application that requires users to authenticate themselves. A user is prompted to provide a user name and a password, which are verified before any access is granted. Developers were told that knowledge of PHP was not required, and the snippet of PHP code represented a simplified version of this authentication process. The scenario also mentioned that method `isset` determines whether a variable is set and is not NULL.

The code is vulnerable because different messages are given for valid and invalid user login, as shown in line 6, 13 and 17 in Figure 1. This helps an adversary to discover valid user names. In the **controlled condition**, developers were asked the following warm-up questions: *What is the user input to this program? What happens when this code executes?* These warm-up questions were present in all the scenarios in the controlled condition. Then the developer was asked to modify this code to allow a user to log in as *guest* if she enters a user name of `guest`. In this situation, the user will have restricted access to the system resources. The developer was told to consider places that might be improved while the developer was modifying the code. The goal was to give developers who were aware of any security concerns an opportunity to address them. This way, we also verify the developer's willingness or motivations to improve the code.

The last vulnerability is about the Python SSL API, which makes web applications vulnerable to man-in-the-middle attacks [18]. An example is given in Figure 2. In the figure, function `wrap_socket` is used by a client to open an SSL connection with a server. Normally when an application (SSL client) checks the server's SSL certificates, it ensures that there is a chain-of-trust from a root certificate (preloaded onto the system) to the provided certificate. While the end-to-end encryption of SSL provides integrity and confidentiality, certificate checking validates the authenticity of the server site. However, function `wrap_socket` in Python does not validate certificates received from the server. As a result of this vulnerability, any malicious party can provide a certificate claiming to be an official website, and the fake certificate will be trusted.

```
import socket, ssl

sock = socket.socket()
sock.connect((HOSTNAME, PORTNUMBER))
# wrap socket to add SSL support
ssl_sock = ssl.wrap_socket(sock,
cert_reqs = ssl.CERT_REQUIRED,
ca_certs =
"/etc/ssl/certs/ca-certificates.crt")

...
print("Successful SSL connection")
# transfer data between client and server
...
ssl_sock.close()
```

Figure 2: The Python SSL vulnerability scenario.

Figure 2 shows the Python SSL scenario in the survey. Developers first received basic information about how SSL works, and then were asked to consider an SSL client written in Python. In the **priming condition**, developers were asked the following questions: *Could a developer experience unexpected results when running such code? What could be examples of these unexpected results and where do they appear in the code?*

In the **explicit condition** for all scenarios, subjects were asked the following questions: *This code has a vulnerability (security flaw) that allows attackers to violate certain security policies of the [web application/program]. Can you pinpoint the problem? Please describe the vulnerability and where it occurs. Why do you think that developers have problems pinpointing this particular problem?*

Following the explicit conditions, subjects responded to **open-ended questions** that addressed the manipulation of information: (i) *Were all the programming scenarios and associated questions clear? If not, what confused you?* (ii) *When you were asked to modify the code, were you suspecting to find vulnerabilities? If not, why do you think you missed it?* (iii) *Did asking you about unexpected results cue you to think about potential vulnerabilities in the code?* (iv) *Did explicitly asking you to find the vulnerability force you to change your approach while examining the code and answering the subsequent questions? If so, explain.*

The participants were also asked about their personal experience with computer security (*have you taken computer security classes? If so did these classes help you pinpoint vulnerabilities in the snippets of code?*), their familiarity with each of the tested vulnerabilities (*Are you familiar with [the exemplified] vulnerabilities? If so please explain how your knowledge about these common vulnerabilities influenced or did not influence how you approached the scenarios?*), and whether they will be more aware of these vulnerabilities when developing new projects (*Now that you are aware of the different vulnerabilities, do you think you will think about vulnerabilities in future programming tasks?*). To leverage the educational purposes of the study, subjects were thoroughly debriefed on each of the security vulnerabilities and subsequently debriefed on the true purpose of the study. Due to space limitations, only two scenarios are shown in Figures 1 and 2.

In order to ensure a consistent measure for the accuracy of subjects' responses, each response was graded by at least two experimenters. Accuracy was measured for each response on a scale from zero (*i.e.*, completely incorrect) to two (*i.e.*, completely correct). A score of one signified an incomplete but correct response to the question, whereas a score of two indicated that the subject had correctly pinpointed the specific vulnerability. A separate score was included for each question signifying whether or not participants modified the code in a way that addressed the particular security vulnerability.

5. RESULTS AND ANALYSIS

A total of 84 developers agreed to take the survey. From this set, 47 surveys were considered in this section as the respondents worked through all six scenarios and answered all the debrief open-ended questions. Table 1 shows the distribution of the participants according to their occupation, and Table 2 describes their educational backgrounds. Participants ranged from 20-52 years of age (M = 31) and approximately 81% of participants were male and 19% were female. The majority of the participants (86%) have a degree in computer science or related majors. All participants who reported not having a degree are senior undergraduate students working towards a degree. Approximately 66% of the participants have never taken a security class in college or any type of security training in the course of their careers.

Occupation	Percentage
Developers from the industry	60%
Students (senior undergraduates and graduate students)	23%
CS Faculty	6%
Other occupations related to software development (*e.g.*, managers and testers)	11%

Table 1: Participant's occupations.

Educational background	Percentage
Doctoral degree	22%
Master's degree	34%
4-year college degree	38%
Not completed 4-year college degree	6%

Table 2: Participant's educational background.

5.1 Statistical Results

This subsection presents the statistical results of our study. The results described in this section use analysis of variance (ANOVA) to analyze the differences between group means and their associated procedures (such as "variation" among and between groups) [23]. ANOVA is a generalization of t-tests to more than two groups. A two-sample t-test is a statistical test that examines whether two samples are different. It is commonly used when the variances of two normal distributions are unknown and when an experiment uses a small sample size. Both ANOVA and t-tests compute the ratio between the obtained difference between the means and the mean difference expected by chance. The goal is to determine whether the obtained difference between means are larger than expected by chance, which yields a larger F value.

In ANOVA, the denominator of the F-ratio is the error variance or mean squared error (MSe), and measures how much variance is expected if there are no systematic treatment effects and no individual differences contributing to the variability of scores. A value of $p < 0.05$ means that the variation seen due to the manipulation of the variables has a probability of at least 95% certainty. The Pearson correlation (r) measures the degree of linear relationship between two variables.

5.1.1 Variables Used in Analysis

Table 4 summarizes the frequencies and mean for the variables used in this analysis. *Total score* is the maximum score that a participant could have obtained on a session. The *Missing* line represents particular questions within a scenario that were not answered by the participants. The *Explicit* variable indicates whether the re-

	N	Min	Max	Mean	Std. Dev.
Gender	47	1	2	1.19	0.398
Age	47	20	52	30.96	7.049
Occupation	47	1	4	1.68	1.002
Level of Education	47	1	8	4.68	1.200
Total Score	40	16	36	27.63	4.301
Explicit	44	1	2	1.11	0.321
Degree	44	1	2	1.14	0.347
Security Classes	44	1	2	1.66	0.479

Table 3: Descriptive statistics for the variables used in analysis.

		Gender	Age	Occupation	Level of Education	Total Score	Explicit	Degree	Security Classes
N	Valid	47	47	47	47	40	44	44	44
	Missing	0	0	0	0	7	3	3	3
Mean		1.19	30.96	1.68	4.68	27.63	1.11	1.14	1.66

Table 4: Frequencies for the variables used in the analysis.

		Occupation	Level of Education	Total Score	Degree	Security Classes
Occupation	Pearson Correlation	1	0.058	-0.262	0.320	0.058
	Sig. (2-tailed)		0.698	0.102	**0.034**	0.709
	N	47	47	40	44	44
Level of Education	Pearson Correlation	0.058	1	0.168	-0.168	-0.70
	Sig. (2-tailed)	0.698		0.301	0.277	0.653
	N	47	47	40	44	44
Total Score	Pearson Correlation	-0.262	0.168	1	-0.305	-0.313
	Sig. (2-tailed)	0.102	0.301		0.056	**0.050**
	N	40	40	40	40	40
Degree	Pearson Correlation	0.320	-0.168	-0.305	1	-0.133
	Sig. (2-tailed)	**0.034**	0.277	0.056		0.388
	N	44	44	40	44	44
Security Classes	Pearson Correlation	0.058	-0.070	-0.313	-1.133	1
	Sig. (2-tailed)	0.709	0.653	**0.050**	0.388	
	N	44	44	40	44	44

Table 5: Correlations among the variables used in the analysis. The highlighted correlations are considered significant.

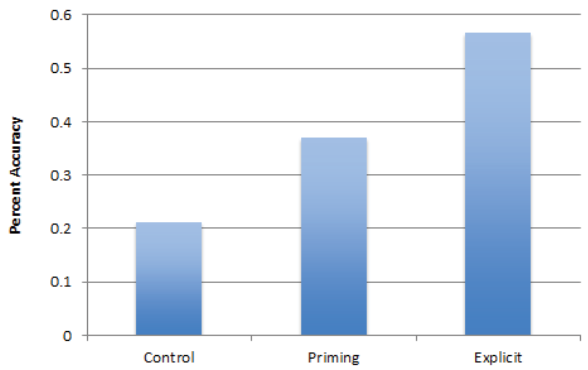


Figure 3: Mean percent accuracy for information conditions across all scenario types.

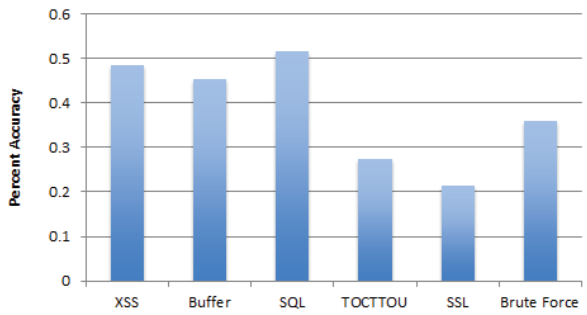


Figure 4: Mean percent accuracy for all scenario types across levels of information conditions.

spondent thought that explicitly asking about vulnerabilities while they worked on the scenarios changed their mindset towards security. The *Degree* variable indicates whether the respondent has a 4-year college degree in CS or related major. *Security Classes* indicates whether the respondent has ever taken security courses or training. Table 3 illustrates the descriptive statistics for these variables.

5.1.2 Result Interpretation

Because the accuracy for each response was graded by two separate experimenters, accuracy result was averaged across the two scores. For each subject, mean accuracy data was obtained for each of the six scenarios and subsequently averaged across all subjects. Two one-way ANOVAs revealed significant effects across the three levels of information ($F(2, 36) = 11.84$, $MSe = 0.30$, $p < 0.05$) and the six scenario types ($F(5, 80) = 7.80$, $MSe = 0.23$, $p < 0.05$).

With regard to the levels of information: Follow-up t-tests, which compares the means between the controlled, implicit, and explicit information conditions, revealed that subjects were significantly less accurate in controlled condition ($M = 0.31$) than in implicit ($M = 0.90$) and explicit ($M = 1.10$) conditions ($t(18) = 3.43$, $SE = 0.17$, $p < 0.05$; $t(19) = 5.86$, $SE = 0.14$, $p < 0.05$). As shown in Figure 3, these findings suggest that prompting developers to think about how other developers could encounter potential vulnerabilities with a particular snippet of code increases the likelihood that they will recognize the security flaw.

	Control	Priming	Explicit	Total
Non-security related	6	6	0	12
Security related	2	4	4	10
Total	8	10	4	22

Table 6: Questionnaire breakdown into security and non-security questions.

With regard to the types of vulnerability scenarios: Follow-up t-tests revealed that subjects were most accurate in identifying the SQL injection vulnerability ($M = 1.27$) and least accurate in identifying the SSL Python vulnerability ($M = 0.37$) ($t(17) = 4.54$, $SE = 0.19$, $p < 0.05$). As shown in Figure 4, these results suggest that the nature of the scenarios themselves may have a correlation with cognitive blind-spots. To further investigate how previous knowledge and experience may influence subjects' response accuracy, a correlation analysis revealed that the number of vulnerabilities reported as familiar to subjects was positively correlated with higher accuracy ($r(47) = 0.4$, $p < 0.05$).

Because familiarity is linked to experience, we conducted a correlation analysis on the dimensions of subjects' experience, including their occupation, whether or not they had a degree in computer science, and whether or not they had taken computer security courses and the number of known vulnerabilities. Results revealed

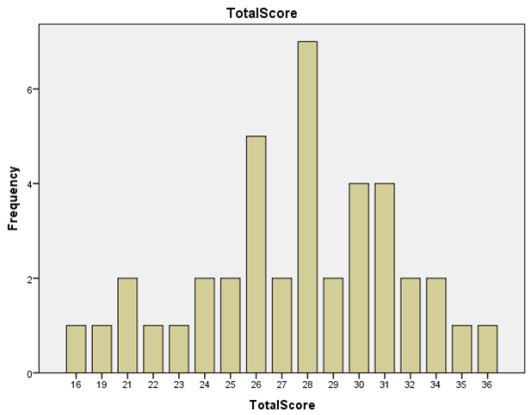


Figure 5: Participant's score distribution.

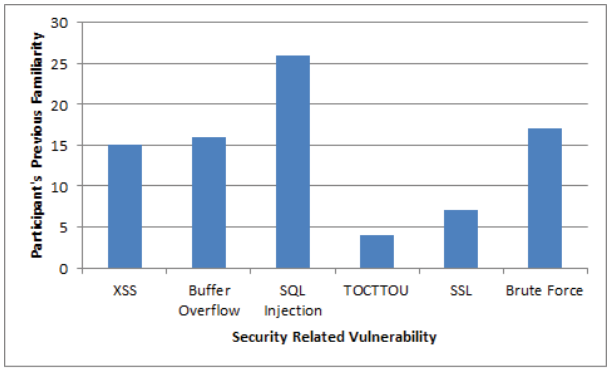


Figure 6: Participant's familiarity with vulnerabilities exercised in the questionnaire.

that the number of known vulnerabilities was moderately correlated with having taken computer security classes ($r(47) = 0.27$, $p < 0.05$). These results indicate that developers are more likely to find security-related blind-spots if they have been formally educated on security related issues. Table 5 presents the correlations among the variables and shows how the score obtained by a respondent is highly correlated with whether they took some security training or courses.

5.2 Results of Experiment Manipulation

The debrief open-ended questions asked participants how the manipulation used in the experiment worked and its effects on how well the participants answered the security-related questions. Table 6 details how each questionnaire was graded. Each questionnaire contained 22 questions where 12 (54%) were not security related and 10 (45%) were security related. The non-security questions were warm-up questions or questions used to aid in the psychological manipulation. Two questions in the explicit condition were open-ended and were not graded (*Why do you think developers have problems pinpointing this particular problem?*). Thus, the maximum score a respondent could obtain was 40. Figure 5 shows the distribution of participants scores.

5.2.1 *Familiarity with Vulnerabilities*

The debrief questions also asked the respondents whether they were familiar with the vulnerabilities exercised in the study and Figure 6 summarizes their answers. The goal of these questions was to manually correlate a respondent score on a scenario with the familiarity they reported for the exercised vulnerability. For example, how the participants that declared knowledge of the SQL injection vulnerability scored on the scenario that exercised this vulnerability? Were they able to correlate the scenario with the vulnerability? In other words, were they able to apply their previous knowledge about the vulnerability *at the time* they needed? When given the chance to improve the code containing the corresponding vulnerability did they remove the vulnerability? Table 7 summarize this analysis.

The results in Table 7 show that many developers who were familiar with the exercised vulnerabilities had difficulties correlating the information about them to the current programming scenario or failed to fix vulnerable code when given an opportunity. For example, 53% of the participants knew a particular vulnerability but did not correlate it to working scenario. This is because the needed security information or the security thinking was not included in their heuristics at the moment.

5.2.2 *Effectiveness of Psychological Manipulation*

Despite the familiarity with vulnerabilities is weakly correlated to secure programming, the results in Table 8 show that when primed about the possibility of finding vulnerabilities developers changed their mindset towards security. In particular, 83% of the participants stated that explicitly mentioning vulnerabilities changed their programming approach and security-mindset. Table 8 shows the effectiveness of the manipulation and the use of priming can change a developer's approach to coding. This is an encouraging result.

	Frequency	Overlooked Vulnerabilities
Knew vulnerability but did not correlate it to working scenario	25 (53%)	Brute force (17 instances - 68%) SQL injection (6 instances - 24%) Buffer overflow (5 instances - 20%) XSS (4 instances - 20%) TOCTTOU (3 instances - 12%) Python SSL (2 instances - 8%)
Knew vulnerability but did not remove it from code when given a chance	17 (36%)	Buffer overflow (7 instances - 41%) SQL injection (7 instances - 41%) Brute force, TOCTTOU and Python SSL (1 instance each - 0.05%)

Table 7: Vulnerability knowledge and the application of this knowledge when needed it.

	Yes	No	Maybe/Unsure
Suspecting of finding vulnerabilities	15 (32%)	30 (64%)	2 (0.04%)
Asking about unexpected results cued to think about vulnerabilities	28 (60%)	19 (40%)	0 (0%)
Explicitly mentioning vulnerabilities changed your approach to a security-mindset	39 (83%)	5 (10.6%)	3 (6.4%)

Table 8: Effectiveness of the psychological manipulation and priming to change developer's approach to security.

Theme	Mentioned by developers	Representative Quotes
1) Specifically mention vulnerabilities changed developer approach	39 (83%)	*"it it put me in the mindset that this code would be very easy to infiltrate"* *"Until the words vulnerability and/or security were used, I had not thought of security risks yet"*
2) Priming about security changes developer's mindset	28 (60%)	*"Yes, I expected to due to the power of suggestion"* *"Once we were looking for unexpected results, I immediately started thinking of what a dumb and/or malicious user would attempt to do with the program"*
3) Security is not a priority / Developer's mindset does not include security	23 (48%)	*"Developers usually focus in delivering functional requirements"* *"I generally don't look for vulnerabilities in code"* *"developers look for the most immediate solution to the current problem they are facing, such as copying one buffer to a second, and run with whatever solution Google brings them first"*
4) Developers assume common cases	16 (34%)	*"We usually try to solve the problem for a set of inputs, not for all possible inputs."* *"Developers seem to be constantly reminded of the fact that users are "dummies" and any mistake they can make, they will make, etc. However, we do not tend to think of the user as evil"*
5) Security thinking requires cognitive effort	14 (30%)	*"It's hard to understand the fact that the user input can directly affect the execution of the code, changing what it was supposed to do to something else."* *"Remembering to sanitize input is tedious"* *"In general, security and vulnerability problems are hard to find"*
6) Developers trust APIs	14 (30%)	*"It's not straightforward that misusing strcpy can lead to very serious problems. Since it's part of the standard library, developers will assume it's ok to use. It's not called unsafe_strcpy or anything, so it's not immediately clear that that problem is there"* *"normally people expects that an official library does not have this kind of vulnerability as default.* *"The security check made prior to accessing the file gives a false impression that the any non-permitted users will be denied access upfront. [For TOCTTOU scenario]"*
7) For certain tasks or CS fields, security is not an issue	6 (13%)	*"In terms of academic AI code, security is really not an issue"* *"But I don't do a lot of network programming so I feel like it matters a bit less [likely to think about security in the future]"* *"yes [i would think about security in the future], if the app is going to be exposed to general public. no , for internal apps, running inside a firewall."*
8) Economic incentives	6 (13%)	*"The reward for making a code safer is not easily seen by others and may come only on the long run, when the code needs less maintenance."* *"there's the economic side. Developers are measured and paid for the features delivered without functionality bugs"*
9) Security education	6 (13%)	*"When we are aware of a vulnerability it starts to be a part of our checklist"*

Table 9: Open-ended answers analysis - Coding.

5.3 Coding

Coding is a technique used in qualitative analysis studies in the social, behavioral and economic sciences for analysis of data. The main idea of coding is to associate what the respondent said in an interview or survey with a set of themes, concepts or categories [24]. Coding is done while reading the transcripts or an interview or the contents of an online survey. A code is a word or a short phrase that captures a datum's primary content or essence [25].

In this study the respondents' comments and answers to open-ended questions were coded according to seven themes. Table 9 shows the themes with the percentage of developers that mentioned the theme and some representative quotes from different developers.

6. DISCUSSION

The results of this study corroborated our hypothesis that security is not part of the heuristics used by developers in their daily programming tasks. Humans have a short working memory, and can only keep a limited number of mental elements readily available at any time [26]. For a variety of reasons, security information is generally not included among these elements. Developers mostly focus on functionality requirements and performance. Developers are trained, evaluated and paid for delivering feature-rich software with good performance levels. They do not see an economic incentive for *squeezing* security thinking into their working memories and producing safe code. Developers and managers see this issue as a zero-sum game, and time spent on *"quality"* will adversely affect function points[1], deadlines, timetables and budgets.

Also, developers usually assume common cases for the inputs a piece of code will receive and the possible states the program can reach. Vulnerabilities lie in uncommon cases overlooked by the developers and exploited by a clever adversary. Attackers make uncommon code paths happen, whereas system designers focus on the common code paths that they know about and are often not aware of the attack code path until the carefully crafted input is presented to them. Static analysis methods, such as Denning's lattice model [27], overcome this limitation by analyzing all possible code paths. This can be effective (despite being formally undecidable in the general case), but have limitations on the type of programs they can be applied and the type of vulnerabilities they can pinpoint.

Developers' failure to address uncommon information flows is also caused by the complexity of fault analysis. Therefore, security thinking requires significant cognitive effort, while people use as little effort as necessary to solve a problem [13].

This study also shows that developers tend to blindly trust code from a reputable source, *e.g.*, API code. Given the way humans think and use shortcuts, simply assuming the correctness of third party code from a reputable source simplifies developers heuristics and their cognitive efforts to do their work. This also has to do with attribution, as if anything goes wrong, developers are not to be blamed as they were just using a well-known API or component.

To make matters worse, our society provides perverse economic incentives for a market of insecure software. As discussed by Ross Anderson in his classic ACSAC 2001 paper [28], the party who is in

[1]A unit of measurement of the amount of functionality an information system (as a product) provides to a user.

a position to protect a system is not the party that suffers the results of security failure. The computer software and systems market is not regulated: they select software and systems that reach the user as quickly and as feature-rich as possible. Moreover, information warfare also plays a role because the same software system that can leave millions of people vulnerable to attacks can be leveraged by national states to conduct cyber warfare and espionage. Finally, Anderson argues that it is much easier to attack than to defend [28]. This shows that there is not a "silver bullet" to solve the technical, economical, legal problems, and psychological challenges of vulnerable software as it involves.

6.1 Recommendations for Developers

In spite of the above, from a Psychology viewpoint, this paper advocates that **security information should reach users when they need it, on the spot, and not the other way around**. It is commonly assumed that developers should educate themselves about security and then apply the acquired knowledge when needed. However, this assumption goes against how the human brain naturally behaves. Our security solutions would be most effective if they leveraged how humans think. This study showed how priming security information when developers need it, *on the spot*, changed their approach towards security and adapted them to include security thinking in their repertoire of heuristics.

This is not an argument against previous and general security education. On the contrary, security education is essential and the results of this study showed that participants who had previous security training performed better in the security-related questions than participants that have never looked for security information. Bringing security information to developers and not expecting them to look for it, as advocated here, will streamline the process of retrieving previously acquired security information. The results of this study also showed that many developers who were familiar with the exercised vulnerabilities had difficulties correlating the information about them to the current programming scenario or failed to fix vulnerable code when given an opportunity. This is because the needed security information or the security thinking was not included in their heuristics at the moment. However, when primed about the possibility of finding vulnerabilities developers changed their mindset towards security.

Given the importance of security in matters, such as cyber crime, cyber warfare and privacy, we recommend that systems and software that interface with the developer (IDEs, text editors, browsers, compilers, *etc.*) bring security information or prime developers *on the spot* when they need it: *while coding*. This priming information should be closely related to their current working scenario to increase the chances that this security cue will be included in the developer's heuristics. Our insight will influence the next generation of tools and applications for developers so that more secure software reach the market. While such paradigm does not solve the multifaceted challenges of cyber security, it can illuminate developers' mental blind spots in vulnerabilities and secure programming.

7. RELATED WORK

The work presented in this paper intersects with the areas of vulnerability analysis, information security perception, and cognitive and human factors. This section discusses the related work in these areas.

7.1 Vulnerability Studies

The first effort towards understanding software vulnerabilities appeared in the 1970's through the RISOS Project that investigates security flaws in operating systems [29]. Around the same time, Protection Analysis study [30] focused on developing vulnerability detection tools to assist developers. Other vulnerability studies followed, such as the taxonomies by Landwehr *et al.* [31] and Aslam [32]. In the 1990's, Bishop and Bailey [33] analyzed current vulnerability taxonomies and concluded that they are imperfect: depending on the layer of abstraction that a vulnerability was considered, it could be classified in multiple ways. More recently Crandall and Oliveira [34] proposed a view of software vulnerabilities as fractures in the interpretation of information as it flows across boundaries of abstraction.

There are also discussions about the theoretical and computational science of exploit techniques and proposals for explicit parsing and normalization of inputs. Bratus *et al.* [35] discussed a view that the theoretical language aspects of computer science lie at the heart of practical computer security problems, especially exploitable vulnerabilities. Samuel and Erlingsson [36] proposed input normalization via parsing as an effective way to prevent vulnerabilities that allow attackers to break out of data contexts. Garg and Camp [37] identified systematic errors by decision-makers leveraging heuristics as a way to improve security designs for risk averse people.

Researchers have also studied vulnerability trends. A study by Browne *et al.* [38] determined that the rates at which incidents were reported to CERT could be mathematically modeled. Gopalakrishna and Spafford [39] analyzed software vulnerabilities in five critical software artifacts using information from public vulnerability databases to predict trends. Alhazmi *et al.* [40] presented a vulnerability discovery model to predict long and short term vulnerabilities for several major operating systems. Anbalagan and Vouk [41] analyzed and classified thousands of vulnerabilities from OSVDB [42] and discovered a relationship between vulnerabilities and exploits. Wu *et al.* [43] performed an ontology-guided analysis of vulnerabilities and studied how semantic templates can be leveraged to identify further information and trends. Zhang *et al.* [44] analyzed vulnerabilities from the NVD database using machine learning to unsuccessfully discover the time to the next vulnerability for a given software application.

There are also studies on developer's practices. Meneely and Williams [45] studied developers collaboration and unfocused contributions into developer activity metrics and statistically correlated them. Schryen [46] analyzed the patching behavior of software vendors of open-source and closed-source software, and found that the policy of a particular software vendor is the most influential factor on patching behavior.

However, none of the above research efforts directly leveraged the human factor to understand software vulnerabilities as proposed in this paper.

7.2 Information Security Perception

Asghapours *et al.* [47] advocate the use of mental models of computer security risks for improvement of risk communication to naive end users. Risk communication consists of security experts messages to non-experts and a mental model in a simplified internal concept of how something works in reality. In their study they leveraged five conceptual models from the literature: Physical Safety, Medical Infections, Criminal Behavior, Warfare, and Economic Failures.

Huang *et al.*[48] studied ways to adjust people's perception of information security to increase their intention to adopt IT appliances and compliance with security practices. Their user study involving e-banking and passwords showed that knowledge is a key factor influencing the gap between people's perceived security and a system real security.

Garg and Camp [49] leverage the classic Fischhoff's canonical nine dimensional model of offline risk perception [50] to better understand online risk perceptions. They found that the results obtained for online risks differed from the ones obtained for offline risks and that the severity of a risk was the biggest factor in shaping risk perception.

Research has also been done in the area of computer warnings for end users [51, 52]. In computer security, warnings are designed to protect people from becoming victims of attacks, such as phishing, malware installation, e-mail spam, *etc.* Researchers have found that people tend to not pay attention to messages that do not map well onto a clear course of action [51]. This corroborates our hypothesis that unless the cue is related to the working scenario at hand, it will likely to be left out from the decision-maker's repertoire of

heuristics.

All these studies consider information security perception from the non-expert end user viewpoint and not developers.

7.3 Human Factors in Software Development

Using human factors in technology research is not a new concept. Curtis, Krasner, and Iscoe [53] studied the software development processes by interviewing programmers from 17 large software development projects. They tried to understand the effect of behavioral and cognitive processes in software productivity. They believed software quality in general could be improved by attacking the problems they discovered in this exploratory research. They summarized the study by describing the implication of their interviews and observations on different aspects of the software development process, including team building, software tools and development environment and model.

Others also recognized the role of cognition in program representation and comprehension [54, 55], design strategies and patterns [56, 57], and software design [58, 59]. These studies show the evolution of design paradigm and development tools from task-centered to human-centered. Current software development tools are very good at pinpointing errors and making sensible suggestions to avoid problems later. New derivatives are created to assist programmers. They have helped the software development process to be less error-prone in general. These studies paved the way for secure software development from the human aspect.

We believe this paper leverages those studies as stepping stones and investigate deeper in the human factor issues, in particular, to understand the impact of cognitive processes on software vulnerabilities. Like previous studies that lead to better software development tool, faster turnout, and more robust software integration, the insights obtained from this study can help the software security community gain insights, improve software security, better the design of guidelines, and build more effective vulnerability blind spot tools.

8. CONCLUSIONS

This paper investigated a hypothesis that software vulnerabilities are blind spots in developers' heuristics in their daily coding activities. Humans have been hardwired through evolution for adopting shortcuts and heuristics in decision making due to the limitations in their working memory. As vulnerabilities lie in uncommon code paths and we have a market that generates perverse incentives for insecure software, security information is often left behind from developer's repertoire of coding strategies.

A study with 47 developers using psychological manipulation was conducted to validate this hypothesis. In this study each developer worked for approximately one hour on six programming scenarios that contained vulnerabilities. The developers were told that the study's goal was to understand developer's mental models while coding. The sessions progressed from providing no information at all about possible vulnerabilities, to priming developers about unexpected results, and explicitly mentioning the existence of vulnerabilities in the code. The results show that developers in general changed their mindset towards security when primed about vulnerabilities on the spot. When not primed, even developers familiar with certain vulnerabilities failed to correlate them with their working scenario and fix them in the code when given a chance. Therefore, the assumption that developers should be educated about security and then apply what they learned while coding goes against the way the human brain behaves. This paper advocates that this assumption be reversed and that security information should reach developers when they need it, *on the spot* and correlated to their tasks at hand. The authors hope that these insights can influence the next generation of tools interfacing developers, such as IDE's, text editors, browsers and compilers so that more secure software reach the market.

Plans for future work include an investigation of the best methodologies for cueing developers and an analysis of the correlation of cueing effectiveness and previous security education.

Acknowledgments

We would like to thank the developers who participated in this study and the anonymous reviewers for valuable feedback. This research is funded by the National Science Foundation under grants CNS-1149730, CNS-1223588, and CNS-1205415.

9. REFERENCES

[1] "Symantec Internet Security Threat Report 2013." `http://www.symantec.com/content/en/us/enterprise/other_resources/b-istr_main_report_v18_2012_21291018.en-us.pdf`.

[2] B. K. Marshall, "PasswordResearch.com Authentication News: Passwords Found in the Wild for January 2013." `http://blog.passwordresearch.com/2013/02/passwords-found-in-wild-for-january-2013.html`.

[3] "Seventeen steps to safer C code." `http://www.embedded.com/design/programming-languages-and-tools/4215552/Seventeen-steps-to-safer-C-code`.

[4] D. Kahneman and A. Tversky, "On the reality of cognitive illusions," *Psychological Review*, pp. 582–591, 1996.

[5] G. Gigerenzer, R. Hertwig, and T. Pachir, *Heuristics: The Foundations of Adaptive Behavior*. Oxford University Press, 2011.

[6] B. Schwartz, "The tyranny of choice," *Scientific American*, pp. 71–75, 2004.

[7] S. Botti and S. S. Iyengar, "The dark side of choice: When choice impairs social welfare," *American Marketing Association*, pp. 24–38, 2006.

[8] C. Kern, A. Kesavan, and N. Daswani, *Foundations of security: what every programmer needs to know*. Apress, 2007.

[9] E. Harmon-Jones, D. M. Amodio, and L. R. Zinner, *Social psychological methods of emotion elicitation (Handbook of Emotion Elicitation and Assessment)*. Oxford University Press, 2007.

[10] W. Thorngate, "Efficient decision heuristics," *Behavioral Science*, vol. 25, no. 3, pp. 219–225, 1980.

[11] K. V. Katsikopoulos, "Efficient decision heuristics," *Decision Analysis*, vol. 8, no. 1, pp. 10–29, 2011.

[12] J. W. Payne, J. R. Bettman, and E. J. Johnson, *The Adaptive Decision Maker*. Cambridge University Press, 1993.

[13] G. K. Zipf, *Human Behavior and The Principle of Least Effort*. Addison-Wesley, 1949.

[14] J. Rieskamp and U. Hoffrage, *Simple Heuristics that Make Us Smart*. Oxford University Press, 1999.

[15] C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang, and H. Hinton, "StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks," in *USENIX Security*, pp. 63–78, Jan 1998.

[16] G. Wassermann and Z. Su, "Static Detection of Cross-site Scripting Vulnerabilities," in *30th International conference on Software engineering*, ICSE '08, (New York, NY, USA), ACM, 2008.

[17] Z. Su and G. Wassermann, "The Essence of Command Injection Attacks in Web Applications," in *Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '06, (New York, NY, USA), pp. 372–382, ACM, 2006.

[18] "Urllib and validation of server certificate." `http://stackoverflow.com/questions/6648952/`.

[19] W. S. McPhee, "Operating System Integrity in OS/VS2," *IBM Systems Journal*, vol. 13, no. 3, pp. 230–252, 1974.
[20] A. Narayanan and V. Shmatikov, "Fast dictionary attacks on passwords using time-space tradeoff," *ACM CCS*, 2005.
[21] R. E. Stake, *Qualitative Research: Studying How Things Work*. The Guilford Press, 2010.
[22] "Qualtrics (http://www.qualtrics.com/)."
[23] F. Gravetter and L. Wallnau, *Statistics for the Behavioral Sciences*. Wadsworth/Thomson Learning, 8th ed., 2009.
[24] R. S. Weiss, *Learning from Strangers - The Art and Method of Qualitative Interview Studies*. The Free Press, 1994.
[25] J. Saldana, *The Coding Manual for Qualitative Researchers*. SAGE Publications, 2012.
[26] A. Newell and H. Simon, *Human Problem Solving*. Prentice Hall, 1972.
[27] D. Denning, "A lattice model of secure information flow," *Communications of ACM*, 1976.
[28] R. Anderson, "Why information security is hard - an economic perspective," *ACSAC*, 2001.
[29] R. P. Abbot, J. S. Chin, J. E. Donnelley, W. L. Konigsford, and D. A. Webb, "Security Analysis and Enhancements of Computer Operating Systems," *NBSIR 76-1041, Institute for Computer Sciences and Technology, National Bureau of Standards*, 1976.
[30] R. B. II and D. Hollingsworth, "Protection Analysis Project Final Report," *ISI/RR-78-13, DTIC AD A056816, USC/Information Sciences Institute*, 1978.
[31] C. E. Landwehr, A. R. Bull, J. P. McDermott, and W. S. Choi, "A Taxonomy of Computer Program Security Flaws," *ACM Computing Surveys*, vol. 26, no. 3, 1994.
[32] T. Aslam, "A Taxonomy of Security Faults in the UNIX Operating System," 1995.
[33] M. Bishop and D. Bailey, "A Critical Analysis of Vulnerability Taxonomies," *Technical Report CSE-96-11, University of California at Davis*, 1996.
[34] J. Crandall and D. Oliveira, "Holographic Vulnerability Studies: Vulnerabilities as Fractures in Interpretation as Information Flows Across Abstraction Boundaries," *New Security Paradigms Workshop (NSPW)*, 2012.
[35] S. Bratus, M. E. Locasto, M. L. Patterson, L. Sassaman, and A. Shubina, "Exploit Programming: From Buffer Overflows to "Weird Machines" and Theory of Computation." USENIX *;login*, December 2011.
[36] M. Samuel and U. Erlingsson, "Let's Parse to Prevent pwnage (invited position paper)," in *Proceedings of the 5th USENIX conference on Large-Scale Exploits and Emergent Threats*, LEET'12, (Berkeley, CA, USA), pp. 3–3, USENIX Association, 2012.
[37] V. Garg and L. J. Camp, "Heuristics and biases: Implications for security," *IEEE Technology & Society*, March 2013.
[38] H. K. Browne, W. A. Arbaugh, J. McHugh, and W. L. Fithen, "A trend analysis of exploitations," *IEEE Symposium on Security and Privacy*, 2001.
[39] R. Gopalakrishna and E. H. Spafford, "A Trend Analysis of Vulnerabilities," *CERIAS Tech Report 2005-05*, 2005.
[40] O. H. Alhazmi and Y. K. Malaiya, "Prediction capabilities of vulnerability discovery models," *IEEE Reliability and Maintainability Symposium (RAMS)*, pp. 86–91, 2006.
[41] O. H. Alhazmi and Y. K. Malaiya, "Towards a unifying approach in understanding security problems," *IEEE International Conference on Software Reliability Engineering (ISSRE)*, pp. 136–145, 2009.
[42] "Open Source Vulnerability Database (http://www.osvdb.org/)."
[43] Y. Wu, R. A. Gandhi, and H. Siy, "Using Semantic Templates to Study Vulnerabilities Recorded in Large Software Repositories," *ICSE Workshop on Software Engineering for Secure Systems*, 2010.
[44] S. Zhang, D. Caragea, and X. Ou, "An Empirical Study on using the National Vulnerability Database to Predict Software Vulnerabilities," *International Conference on Database and Expert Systems Applications (DEXA)*, 2011.
[45] A. Meneely and L. Williams, "Secure Open Source Collaboration: An Empirical Study of Linus' Law," *ACM CCS*, pp. 453–462, 2009.
[46] G. Schryen, "A comprehensive and comparative analysis of the patching behavior of open source and closed source software vendors," *IMF*, 2009.
[47] F. Asgapour, D. Liu, and L. J. Camp, "Mental models of computer security risks," *Financial Cryptography and Data Security Lecture Notes in Computer Science*, vol. 4886, pp. 367–377, 2007.
[48] D.-L. Huang, Pei-Luen, P. R. abd Gavriel Salvendya, F. Gaoa, and J. Zhoua, "Factors affecting perception of information security and their impacts on it adoption and security practices," *International Journal of Human-Computer Studies*, vol. 69, no. 12, 2011.
[49] V. Garg and L. J. Camp, "End user perception of online risk under uncertainty," *Hawaii International Conference On System Sciences*, vol. 4886, 2012.
[50] B. Fischhoff, P. Slovic, S. Lichtenstein, and B. C. Stephen Read, "How safe is safe enough? a osychometric study of attitudes towards technological risks and benefits," *Policy Sciences*, vol. 9, no. 2, 1978.
[51] C. Bravo-Lillo, L. Cranor, J. Downs, and S. Komanduri, "Bridging the gap in computer security warnings: A mental model approach," *IEEE Security and Privacy*, vol. 9, no. 2, 2011.
[52] K. Witte, "Putting the fear back into fear appeals: The extended parallel process model," *Communication Monographs*, vol. 59, no. 4, pp. 329–349, 1992.
[53] B. Curtis, H. Krasner, and N. Iscoe, "A field study of the software design process for large systems," *Communications of the ACM*, vol. 31, no. 11, pp. 1268–1287, 1988.
[54] S. Letovsky, "Cognitive processes in program comprehension," *Journal of Systems and software*, vol. 7, no. 4, pp. 325–339, 1987.
[55] H. C. Purchase, L. Colpoys, M. McGill, D. Carrington, and C. Britton, "Uml class diagram syntax: an empirical study of comprehension," in *Proceedings of the 2001 Asia-Pacific symposium on Information visualisation-Volume 9*, pp. 113–120, Australian Computer Society, Inc., 2001.
[56] A. Chatzigeorgiou, N. Tsantalis, and I. Deligiannis, "An empirical study on students ability to comprehend design patterns," *Computers & Education*, vol. 51, no. 3, pp. 1007–1016, 2008.
[57] W. Visserl, J.-M. Hocz, and F. Chesnay, "Expert software design strategies," 1990.
[58] R. Jeffries, A. A. Turner, P. G. Polson, and M. E. Atwood, "The processes involved in designing software," *Cognitive skills and their acquisition*, pp. 255–283, 1981.
[59] B. Adelson and E. Soloway, "A model of software design," *International Journal of Intelligent Systems*, vol. 1, no. 3, pp. 195–213, 1986.

Exploring and Mitigating Privacy Threats of HTML5 Geolocation API *

Hyungsub Kim
Dept. of CSE, POSTECH
Pohang, Korea
hyungsubkim@postech.ac.kr

Sangho Lee
Dept. of CSE, POSTECH
Pohang, Korea
sangho2@postech.ac.kr

Jong Kim
Dept. of CSE, POSTECH
Pohang, Korea
jkim@postech.ac.kr

ABSTRACT

The HTML5 Geolocation API realizes location-based services via the Web by granting web sites the geographical location information of user devices. However, the Geolocation API can violate a user's location privacy due to its coarse-grained permission and location models. The API provides either exact location or nothing to web sites even when they only require approximate location. In this paper, we first conduct case studies on numerous web browsers and web sites to explore how they implement and utilize the Geolocation API. We detect 14 vulnerable web browsers and 603 overprivileged web sites that can violate a user's location privacy. To mitigate the privacy threats of the Geolocation API, we propose a novel scheme that (1) supports fine-grained permission and location models, and (2) recommends appropriate privacy settings to each user by inspecting the location sensitivity of each web page. Our scheme can accurately estimate each web page's necessary geolocation degree (estimation accuracy: ∼93.5%). We further provide suggestions to improve the Geolocation API.

1. INTRODUCTION

Location-based services (LBSs) are popular personalized services that are tightly associated with user privacy. Examples of LBSs include navigation services, local search services, traffic alert services, and localized weather services, which are especially useful for mobile device users. However, without assured privacy, users may not trust LBSs. Therefore, following the seminal work of Gruteser and Grunwald [13], numerous researchers propose various schemes [5, 7–9, 20, 23, 31, 32] to ensure location privacy in LBSs.

LBSs via the Web have become necessary because the number of mobile devices that access the Web has increased. The HTML5 specification satisfies such requirements by defining the Geolocation API [29] which grants permission to access the geographical location information (*geolocation*) of devices to web sites. The API consists of two methods: `GetCurrentPosition()`, which retrieve current geolocation including latitude and longitude; and `watchPosition()`, which tracks updated position according to user movements. The accuracy of the geolocation depends on the type of user device. For example, smart mobile devices allow web browsers to precisely estimate the device's geolocation by using various sources, such as the global positioning system (GPS), cell towers, and Wi-Fi access points (APs).

However, we identify that the current specification and implementations of the Geolocation API face *four privacy threats*, which are especially harmful to mobile device users along with exact geolocation. (1) They employ *no fine-grained location model*. Even if a web site only wants to know which country a user is in, both the web site and the user cannot decrease geolocation accuracy when requesting or delivering it. Furthermore, the web site should perform *geocoding* to obtain country information from the received geolocation. (2) They employ *no per-method permission model*. A user must allow a web site to track his or her movements even when the user only wants to allow the site to retrieve the current geolocation. (3) They employ a *per-domain permission model* without a *per-page permission model*. A user cannot allow a web page `site.com/map.html` to access the geolocation while disallowing another web page under the same domain `site.com/mail.html` to access the geolocation. (4) They employ *no re-confirm process* for changed web pages. A web site preserves its permissions to access the geolocation, even though it modifies or deletes the original web page that a user has permitted. We must solve these privacy problems to ensure the location privacy of users in LBSs via the Web.

In this paper, we first explore real-world privacy problems of the Geolocation API due to *vulnerable web browsers* and *overprivileged web sites*. We analyze a number of web browsers for popular mobile platforms (Android and iOS) to inspect (1) the number of web browsers that support Geolocation API, (2) the number of vulnerable web browsers that allow any web site to access the geolocation *without user permissions*, and (3) their interfaces to grant or revoke permissions. We have detected 14 vulnerable web browsers installed on *more than 16 million Android devices* via Google Play Store, and have reported the security problem to the browser developers.

We also analyze 1,196 web sites that use the Geolocation API to identify their characteristics and the degree of geolocation that they demand. We discover that *approximately half of the web sites using the Geolocation API are overprivileged*, i.e., they are irrelevant to precise geolocation, because their content does not much change even when we provide completely different geolocation.

Next, we propose a novel scheme for mitigating the privacy problems of the Geolocation API by supporting fine-grained permission/location models and measuring each web page's location sensitivity. We modify an open-source web browser for Android to (1) grant different degrees of geolocation to different domains/web pages, (2) separate permissions for location tracking, (3) support both per-domain and per-page permission models, and (4) inspect

*This work was supported by ICT R&D program of MSIP/IITP. [14-824-09-013, Resilient Cyber-Physical Systems Research]

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.
ACSAC '14 December 08 - 12 2014, New Orleans, LA, USA
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-3005-3/14/12$15.00
http://dx.doi.org/10.1145/2664243.2664247

web page changes and re-confirm permissions for changed web pages. Furthermore, our web browser can estimate a web page's location sensitivity and use this estimate to recommend a privacy setting to its user.

Evaluation results show that our scheme has low overhead while precisely estimating each web page's necessary geolocation degree (∼93.5% of estimation accuracy).

This paper makes the following contributions:

- **In-depth analysis.** To the best of our knowledge, this is the first study that analyzes the privacy problems of the HTML5 Geolocation API in depth.
- **New case study.** We conduct the first case studies on the Geolocation API regarding various web browsers and web sites, and discover a number of vulnerable web browsers and overprivileged web sites.
- **Effective countermeasure.** We propose effective countermeasures against the Geolocation API's privacy problems by modifying a web browser to support fine-grained permission/location models and inspect a web page's location sensitivity while considering portability and compatibility.
- **Reasonable suggestions.** We offer some suggestions to improve the Geolocation API, such as accuracy options, per-method permissions, and per-page permissions.

The remainder of this paper is organized as follows. §2 explains the HTML5 Geolocation API. §3 conducts case studies on web browsers and web sites with the HTML5 Geolocation API. §4 introduces the threat model and assumptions of this work. §5 explains our countermeasure in details. §6 discusses the limitations of this work and our suggestions to reduce the privacy threats of the Geolocation API. §7 introduces related work. Lastly, §8 concludes this work.

2. BACKGROUND

In this section, we briefly explain the current localization technologies, the Geolocation API of HTML5, and privacy concerns residing in the specification of the Geolocation API.

2.1 Localization Technologies

Smart mobile devices estimate their geographical location by using the global positioning system (GPS), cell-tower triangulation, and Wi-Fi access point (AP) triangulation [11, 14]. The GPS provides highly accurate latitude, longitude, altitude, heading, and speed information to the devices. Its accuracy can be up to several meters. The GPS, however, has two problems: (1) it takes time to initialize the communication between a GPS receiver and GPS satellites; and (2) it usually does not work indoors because GPS signals are difficult to penetrate roofs, walls, and other objects [21]. To overcome these problems, smart mobile devices also use cell-tower-based or Wi-Fi-based triangulation, or both to estimate their latitude and longitude information. Although the triangulation immediately returns location information and supports indoor positioning, its accuracy can be up to several hundreds (Wi-Fi) or thousands (cell tower) of meters. Its coverage and accuracy also depend on whether LBS providers (e.g., Google) correctly and widely collect the location information of cell towers and Wi-Fi APs.

2.2 HTML5 Geolocation API

The HTML5 specification defines the Geolocation API [29] that allows web sites to access the geolocation of user devices. This API consists of two methods. (1) `getCurrentPosition()` inspects geolocation of user devices including latitude, longitude, accuracy, altitude, heading, speed, and timestamp. (2) `watchPosition()` continually retrieves the current geolocation of user devices according to user movements. Both methods have a mandatory parameter

```
var map=document.getElementById("map");

function getLocation() {
❶ if (navigator.geolocation) {
❷   navigator.geolocation.getCurrentPosition(
      showPosition);
  }
  else {
    map.innerHTML="This browser does not support
        HTML5 Geolocation API.";
  }
}

function showPosition(position) {
❸ var latlon=position.coords.latitude+","+
    position.coords.longitude;

❹ var img_url="http://maps.googleapis.com/maps/
    api/staticmap?center="+latlon+"&zoom=13&size
    =500x400&maptype=hybrid&sensor=true";

❺ map.innerHTML="<img src='"+img_url+"'>";
}
```

Figure 1: JavaScript code to obtain a Google map image based on the current geolocation. It inspects whether a web browser supports the Geolocation API (❶), obtains the geolocation (❷), parses the geolocation (❸), receives a map-image URL from the Google map (❹), and displays the map image (❺).

`PositionCallback` that specifies a callback function to execute when they successfully obtain the geolocation.

Figure 1 shows a JavaScript code that uses the `getCurrentPosition()` method to check a user's current geolocation on the Google map (we refer [30].) The code first inspects whether a web browser supports the Geolocation API (❶). If the web browser supports the Geolocation API, the code calls the `getCurrentPosition()` method to obtain the current geolocation of the user (❷). When the method successfully retrieves the geolocation, it calls a callback function `showPosition()`. The callback function reads the current geolocation in an argument `position` (❸), receives a map-image URL from the Google map by using the geolocation (❹), and embeds the image URL in HTML content (❺). In addition, if we use the `watchPosition()` method instead of the `getCurrentPosition()` method, we can easily track the geolocation changes of the user.

The methods of the Geolocation API have two optional parameters. (1) `PositionErrorCallback` specifies a callback function to execute when they fail to obtain the geolocation. (2) `PositionOptions` represents a JavaScript object consisting of three attributes: `enableHighAccuracy` represents whether a web page prefers the best possible results (i.e., it demands GPS-based geolocation,) `timeout` represents the amount of time that the web page waits for receiving the geolocation, and `maximumAge` indicates the validity period of cached geolocation data. The default value of `enableHighAccuracy` is `false` to reduce power consumption due to GPS receivers, and the default values of `timeout` and `maximumAge` are "Infinity".

2.3 Privacy Concerns of Geolocation API

The Geolocation API specification [29] regulates some requirements of both web browsers and web sites to preserve user privacy. First, web browsers need to obtain permissions from users for each web site when it attempts to utilize the Geolocation API. Web sites can use the geolocation information only when the users grant per-

Table 1: How the 60 Android web browsers support the Geolocation API.

Geolocation permission	Number
Permanent & temporary	18
Permanent only	7
No permission check	14
Not available	21

Table 2: Vulnerable Android web browsers that do not ask for the Geolocation permissions.

Name	Version	#Downloads
Baidu Browser	4.1.0.3	10,000,000+
Maxthon Browser for Android	4.3.0.2000	5,000,000+
Angel Browser	12.30z	500,000+
Maxthon Web Browser for Tablet	4.0.4.1000	500,000+
Exsoul Web Browser	3.3.3	100,000+
Full Screen Browser	2.3	100,000+
Harley Browser	1.3.4	100,000+
Maxthon Browser for Pioneer	2.7.3.1000	100,000+
Safe Browser - The Web Filter	1.2.5	100,000+
Baidu Browser for Tablet	1.3.0.2	100,000+
Habit Browser	1.1.25	100,000+
Browser Omega	2.6.1	50,000+
Jelly Web Browser	1.1.4	10,000+
Zomi Mobile Browser	2.6.6	10,000+
Total		16,770,000+

missions. Second, the web browsers have to provide a user-friendly interface to revoke the granted permissions.

The specification also mentions other privacy problems in the non-normative section: (1) users may inadvertently grant permissions to web sites, and (2) already-permitted web sites can silently change their content regardless of a user's intention.

However, the specification mentions *no countermeasures* against these problems. It leaves to implementers the responsibility of solving the problems.

3. CASE STUDIES

In this section, we conduct case studies on the HTML5 Geolocation API to identify how web browsers and web sites implement and utilize it. The case studies are helpful to explore the real-world privacy problems of the Geolocation API.

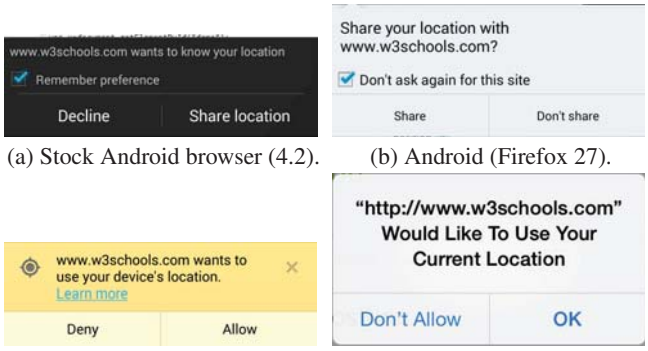
3.1 Case Study of Web Browsers

3.1.1 Android

We collect and analyze 60 free web browsers for Android on Google Play Store in August 2014, which had been installed more than 10,000 times. Except some major web browsers (e.g., Chrome and Firefox), most web browsers for Android rely on WebView [2] which allows Android Apps to embed a customized web browser. We use a Galaxy S III (Android 4.3) and a Galaxy Nexus (Android 4.2) when testing the web browsers.

We first investigate how the 60 Android web browsers support the Geolocation API (Table 1). We detect 39 web browsers that support the Geolocation API. They consist of 18 web browsers that support both permanent and temporary (one time) permissions, 7 web browsers that only support permanent permissions, and 14 web browsers that *do not ask for user permissions*.

More than 16 million Android users have installed the 14 vulnerable web browsers that do not ask for permission (Table 2). Their actual number is certainly *larger* than 16 million because the number



(a) Stock Android browser (4.2). (b) Android (Firefox 27). (c) Android (Chrome 32). (d) iOS 7.

Figure 2: Dialogs for requesting permissions to access geolocation.

of downloads shown in Google Play Store is a lower bound (Google does not reveal the exact number of downloads.) For example, "5,000,000+ downloads" means that the actual number of downloads is between five million and 10 million. Furthermore, Android users may install the vulnerable web browsers via alternative markets or by using Android application package (APK) files. If users visit web sites with the vulnerable browsers, the sites can silently collect the users' geolocation by exploiting the browsers. Therefore, we suggest that the developers of the vulnerable web browsers need to fix the problem (details: §3.2).

Figures 2a, 2b, and 2c show permission dialogs of the major web browsers for Android when users attempt to access a web page that uses the Geolocation API [30]. All of the web browsers give permissions not to web pages but to domains. Furthermore, the stock Android browser (Figure 2a) and Firefox for Android (Figure 2b) support temporary permissions for accessing geolocation whereas Chrome for Android does not (Figure 2c).

Lastly, we investigate the interfaces to determine which of them allow users to revoke the granted permissions stored in the Android web browsers. Among the 25 web browsers that support the Geolocation API and ask for user permission, only six *Android Browser, Chrome Browser, Firefox, iLunascape 2 - Web Browser, Sleipnir Mobile - Web Browser, and Skyfire Web Browser 5.0* allow users to revoke the granted permissions of each domain. With the remaining 19 web browsers, users must delete all granted permissions even if they only want to revoke the permission of a single domain.

3.1.2 iOS

We collect and analyze the top 30 free web browsers for iOS on App Store (e.g., Chrome and Mercury Browser) as of September 2013, and identify that most of the web browsers for iOS use the *same* mechanism to manage the Geolocation API and permissions. The reason is that Apple forces developers to use UIWebView [3] when rendering web pages to harden security. Exceptions are cloud-based web browsers (e.g., Opera Mini and Puffin) that have no JavaScript engine, but they do not support the Geolocation API. We use iPad (third generation with iOS 7) when testing the browsers.

We confirm that the web browsers for iOS only have the permanent permission model for the Geolocation API. Figure 2d shows an example of their permission dialog which requests a permanent geolocation permission for a domain.

Lastly, we identify that users should use the unified interface of iOS to reset *all* location and privacy permission settings when they want to revoke the geolocation permissions granted to web sites. After resetting, all apps and web sites must obtain permissions again to access both geolocation and other private information (e.g., calendars, reminders, and photos). Undoubtedly, this is highly

```
.method public
  onGeolocationPermissionsShowPrompt(
    Ljava/lang/String;
    Landroid/webkit/
      GeolocationPermissions$Callback;)V

    .locals 2
    .parameter
    .parameter

    .prologue
    .line 225
    const/4 v0, 0x1
    const/4 v1, 0x0

➠   invoke-interface {p2, p1, v0, v1},
      Landroid/webkit/
        GeolocationPermissions$Callback;
        ->invoke(Ljava/lang/String;ZZ)V

    .line 227
    return-void
.end method
```

Figure 3: Decompiled `onGeolocationPermissionsShowPrompt()` of the Maxthon Browser for Android.

inconvenient for users.

3.2 Details of Vulnerable Web Browsers

We analyze the vulnerable Android web browsers to know why they do not prompt geolocation permission dialogs and finally detect that this flaw is due to mis-implemented `onGeolocationPermissionsShowPrompt()` methods. The `onGeolocationPermissionsShowPrompt()` method of the `WebChromeClient` class is essential to support the Geolocation API in WebView-based browsers because they call the method when an unseen web site attempts to use the Geolocation API [1]. This method should invoke a callback method to set permissions with three parameters: (1) a domain name, (2) whether a user allows (`true`) or blocks (`false`) the domain, and (3) whether the granted permission is permanent (`true`) or temporary (`false`). However, if the method always invokes the callback method with `true` as the second argument, web browsers always allow any web site to access the geolocation without user permissions. Accordingly, we expect that the `onGeolocationPermissionsShowPrompt()` method of the vulnerable browsers always invokes the callback method with `true`.

Figure 3 shows the decompiled `onGeolocationPermissionsShowPrompt()` method of the Maxthon Browser for Android with `apktool` (the results of other browsers are similar.) As we expect, the method has no instructions to pop up a permission dialog and invokes a callback method while statically assigning `true` (`0x1`) to the second argument `v0` (➠, `p2` is the 0th argument representing the callback method.)

Consequently, we believe that developers need to carefully implement `onGeolocationPermissionsShowPrompt()` methods and Google has to provide a built-in permission dialog to eliminate such a vulnerability.

We reported the security problem to the browser developers. Some of them replied that they would patch it in a future release.

3.3 Case Study of Web Sites

We collect 1196 web pages that use the Geolocation API; to do this we (1) inspect web sites listed on Alexa, (2) use the Google search engine with keywords, such as "near me" and "around me", and (3) use an HTML code search engine [10] with keywords, such as "getCurrentPosition" and "watchPosition", between August 2013 and September 2013, then inspect them. Each of the three sources contributes 246, 140, and 810 web pages, respectively. When we visit web sites, we use user-agent strings of Android or iPhone web browsers and recursively retrieve child web pages. We manually verify that most of the collected web pages provide "near me" services to inform point of interest (POI) locations (e.g., stores, buildings, and the sights) or local information (e.g., local radio and TV channels)[1]. Table 3 summarizes the results. We also find other web pages, including those for local weather services, geographic information, location-based online social networks, local traffic information, and local news.

Table 3: Categories of web pages using the Geolocation API.

Category	Number	%
Near me	667	55.77
Local information	288	24.08
Weather	55	4.60
Geographic information	53	4.43
Social networking	42	3.51
Traffic information	32	2.68
News	13	1.09
Others	46	3.85

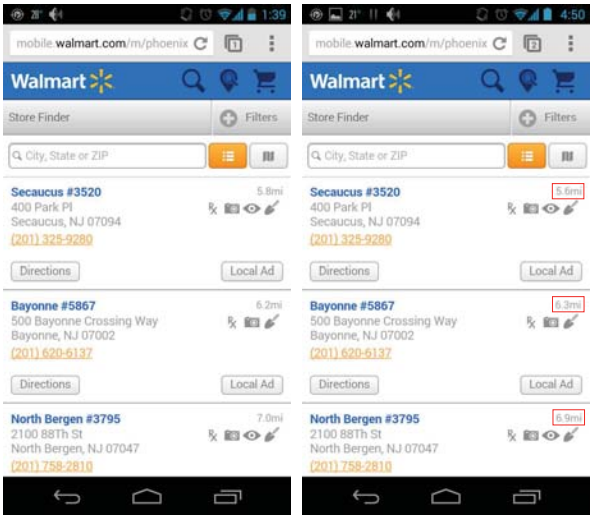
Table 4: Location sensitivity of web pages using the Geolocation API.

Sensitivity	Number	%
Pinpoint	593	49.58
City	426	35.62
State	22	1.84
Country	18	1.51
Unchanged	137	11.45

We aim to inspect the location sensitivity of the collected web pages to decide whether they are overprivileged. We manually perform the following procedure: (1) preparing a number of GPS coordinates around famous cities, (2) visiting the web pages while using various geolocations based on the GPS coordinates, and (3) verifying whether the web pages change according to the given geolocation. We treat street-level changes are equivalent to pinpoint-level changes because both levels are sufficiently fine grain. We thereby compare them for verifying pinpoint-, city-, and state-level geolocation changes, respectively. We also use locations around famous cities of other countries, such as Calgary (Canada), Paris (France), Seoul (Korea), and Sydney (Australia), to analyze whether some web pages work for a specific country. When visiting the web pages, we use Developer Tools (Chrome) or User Agent Switcher and Geolocator extensions (Firefox) to change the user agent strings and geolocation of web browsers.

Our inspection of the location sensitivity of the collected web pages reveals that *half of them do not need to use exact geolocation* (Table 4). Except for 49.6% of the web pages that demand pinpoint geolocation, other web pages provide city-, state-, or country-level information (35.6%, 1.8%, and 1.5%, respectively). For example, Groupon's web page for nearby deals provides city-level information; it surely does not need to obtain exact geolocation. Interestingly, the content of 137 web pages (11.5%) does not change even when we alter the geolocation drastically. We confirm that they unnecessarily demand geolocation by manually investigating them.

[1]Five graduate students participated in manual inspection. Decisions were made by majority voting.



(a) At a latitude of 40.71365 and a longitude of -74.00971. (b) At a latitude of 40.71500 and a longitude of -74.01000.

Figure 4: Mobile web pages of Walmart to find the nearest stores from two places in New York.

3.4 Considerations on "Near Me" Services

Even though the "near me" services include notable LBSs that use exact geolocation, we expect that many of these services can provide the same functionalities without relying on such exact geolocation. As an example, we consider a "near me" service web page of Walmart to find the nearest stores as an example (`http://mobile.walmart.com/m/phoenix#location/locate`). When we visit the web page with two slightly different GPS coordinates, latitudes and longitudes are (40.71365, -74.00971) and (40.71500, -74.01000). Walmart redirects us to two slightly different web pages (Figures 4a and 4b). Therefore, the "near me" service web page demands exact geolocation. The URLs of the final web pages are `http://mobile.walmart.com/m/phoenix#location/list/40.713/-74.009` and `http://mobile.walmart.com/m/phoenix#location/list/40.715/-74.010`, respectively, so Walmart can know the GPS coordinates of visitors from GET request parameters.

However, Walmart can use the city-level geolocation for providing the same service to mitigate privacy threats while reducing computational overhead. They can provide this service by (1) delivering a list of GPS-coordinates of its stores in New York to web browsers and (2) allowing the web browsers to execute a JavaScript code to calculate the distances between the stores and the user's current geolocation. Walmart does not need to compute and sort the distances, so it can reduce computational overhead especially when the number of concurrent users is large. Although each browser needs to perform some computations instead of Walmart, it can protect exact geolocation from Walmart. Consequently, we believe that this approach is good for both web sites and their users.

4. THREAT MODEL AND ASSUMPTIONS

In this section, we explain the threat model and assumptions of this work before introducing our scheme for mitigating the privacy threats of the HTML5 Geolocation API.

First, we assume that our attackers are *honest-but-curious* LBS providers, who use the Geolocation API in a *legitimate but overprivileged* way. They attempt to obtain precise geolocation of users even when their LBSs demand neither precise nor timely geolocation.

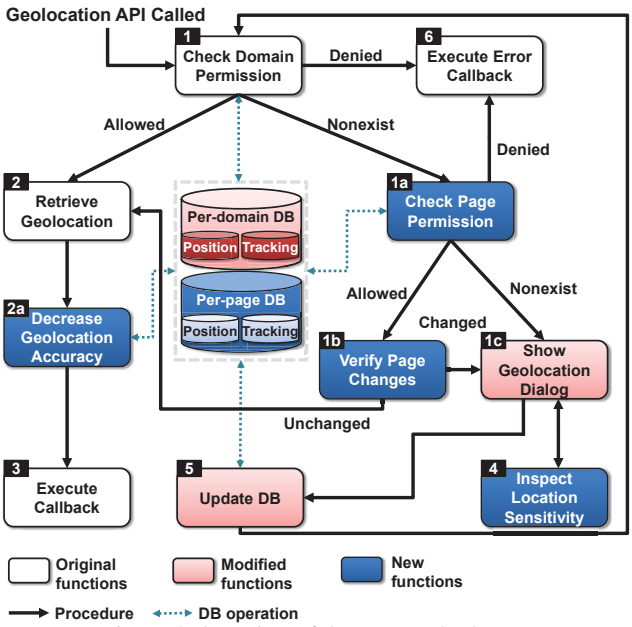


Figure 5: Overview of the proposed scheme.

Therefore, we focus on how to restrict the privilege of web sites in a fine-grained way.

Second, we assume that the attackers try to *implicitly* access the geolocation *without user interaction*. In the case studies, we discover some web sites that explicitly demand user interaction whenever the sites access the geolocation (e.g., users should click a button or type some search keywords to use LBSs.) Such user interaction reminds users that they are granting their geolocation to the web sites; they may deny it at that time to ensure privacy. Consequently, the attackers would not demand such explicit interaction when requesting geolocation, so we focus on geolocation requests without user interaction.

Lastly, we assume that both the web sites providing LBSs and the web browsers visiting the web sites have *no vulnerability* that attackers can exploit. External attackers cannot inject a malicious script into vulnerable web pages or exploit vulnerable web browsers to access geolocation. For this reason, we do not consider solutions to detect script injections and to mitigate vulnerabilities because they are out of the scope of this paper.

5. PROPOSED SCHEME

In this section, we explain the proposed scheme for mitigating the Geolocation API's privacy threats. We implement the proposed scheme on Android because it is popular and allows custom web browsers unlike iOS.

5.1 Overall Design

We explain the overall design of the proposed scheme. When a user visits a web page that contains a JavaScript code to execute the Geolocation API, the proposed scheme handles the execution as follows (Figure 5).

1. The proposed scheme checks whether the *domain* serving the code exists in a *per-domain* permission database (DB). If the domain exists in the DB and the user allows the domain, the scheme goes to Step 2. If the domain does not exist in the DB, it goes to Step 1a. If the user denies the domain, it goes to Step 6.
   1a. The proposed scheme checks whether the *web page*

serving the code exists in a *per-page* permission DB. If the web page exists in the DB and the user allows the web page, the scheme goes to Step 1b. If the web page does not exist in the DB, it goes to Step 1c. If the user denies the web page, it goes to Step 6.

1b. The proposed scheme verifies whether the web page *changes* after obtaining permissions. If the degree of changes exceeds a threshold value, it goes to Step 1c, otherwise, it goes to Step 2.

1c. The proposed scheme composes an enhanced *Geolocation permission dialog* which asks the user to (1) either allow or deny the domain/web page, (2) inspect the location sensitivity of web page, (3) choose the geolocation accuracy, and (4) grant either temporary or permanent permissions. It then goes to either Step 4 or Step 5 according to the user's choices.

2. The proposed scheme retrieves precise geolocation from the OS (Android) and goes to Step 2a.

2a. The proposed scheme *decreases* geolocation accuracy according to the *allowed geolocation accuracy* of the domain/web page stored in per-domain/per-page DBs. It then goes to Step 3.

3. The proposed scheme terminates while executing a *success* callback function with (less accurate) geolocation. The web page's JavaScript code handles the remaining procedures such as composing an HTML document and rendering.
4. The proposed scheme inspects the *location sensitivity* of the web page to estimate the necessary geolocation degree. When composing a Geolocation permission dialog, it uses the sensitivity information. It then returns to Step 1c.
5. The proposed scheme *updates* DBs according to a user's choices and goes to Step 1 to *re-initiate* the procedures.
6. The proposed scheme terminates while executing an *error* callback function when the user denies the domain/web page demanding geolocation.

We respectively apply the explained procedures to `GetCurrentPosition()` and `watchPosition()` as we separate the permissions for position and tracking.

5.2 Inspecting Geolocation Web Pages

We explain how our web browser verifies the changes in a web page that uses the Geolocation API and measures the location sensitivity of the web page.

5.2.1 Verifying web page changes

We verify the changes of web pages that have geolocation permissions because users may want to revoke the granted permissions if they do not prefer the changed web pages or the web pages no longer contain location-based content. For this goal, we need a method to effectively identify the differences between the old and current versions of web pages.

We use a *context triggered piecewise hash algorithm* [19], also known as a *fuzzy hash algorithm*, to verify changes. This algorithm divides a document into blocks based on triggers, computes hash values of each block, and uses the list of hash values to compare different documents. We use this algorithm to confirm changes by comparing the old and current versions of a rendered web page. For image files, we use their histogram values and file sizes for composing hash values.

We explain the procedure to verify web page changes. When our web browser visits a web page for the first time and a user allows the web page to retrieve geolocation, the browser computes a hash value of the web page and stores the hash value along with the current geolocation in its DB. On subsequent visits, the browser additionally opens the web page with the stored geolocation to compute a new hash value of the web page. If the difference between the stored and new hash values exceeds a threshold value, the browser may ask for user permissions again.

When computing the hash value of a web page, our web browser ignores *dynamic content* embedded in the page (e.g., a web banner) because it frequently changes regardless of whether the main text of the web page changes. The browser identifies such dynamic content by (1) visiting a web page several times in a short time period and (2) inspecting changed content over the visits. Finally, the browser computes hash values while excluding the dynamic content.

To lighten its burden, the web browser keeps a time-stamp of the last visit to the web page and verifies changes only when the elapsed time from the last visit is above a certain time.

5.2.2 Estimating location sensitivity

Our web browser estimates the location sensitivity of web pages by varying geolocation within a predefined set of GPS coordinates (§3.3) and verifying web page changes (§5.2.1). First, the web browser extracts *static content* from a web page (i.e., excluding banners). Second, the browser visits the web page several times while using five different addresses that differ in street, city, state, and country, respectively. Third, the browser verifies the changes of the rendered web pages to estimate location sensitivity. Lastly, the browser announces the estimated location sensitivity to a user for recommending privacy configurations. If the browser fails to estimate the location sensitivity because web pages have location-independent content, the browser notifies *random* to a user. Visiting a web page multiple times to enhance privacy is not a new idea. [18, 27] have already considered similar techniques.

5.2.3 Concurrent inspection

Our web browser should visit a web page several times for inspection, so a user may need to wait a long time to check the inspection result. We reduce the waiting time by *concurrently inspecting* a web page. First, the browser simultaneously creates five WebView activities for visiting a single web page with five different addresses that differ in street, city, state, and country, respectively. Next, the browser creates another WebView activity for visiting the web page with one of the five addresses and extracts static content from the two rendered web pages of the same address. Lastly, the browser compares the static content with the five rendered web pages of the different addresses, respectively, to estimate location sensitivity. The overall inspection time is 1.8 times longer than page loading time (details: §5.4.2).

5.3 Managing and Serving Geolocation

We explain how our web browser manages geolocation and serves it to web pages. Our goal is to develop a *portable* and *compatible* solution to the privacy problem of the Geolocation API. Our method, *overriding the Geolocation API*, meets the goal because it modifies neither Android platforms nor JavaScript engines.

5.3.1 Modifying the Geolocation API

To implement our scheme, we modify the behavior of the Geolocation API by overriding its JavaScript methods. We choose an open-source web browser for Android, Lighting Browser [25], relying on WebView [2]. WebView supports JavaScript and allows developers to inject arbitrary JavaScript codes into a loaded web page; we use these features to override the Geolocation API (we discuss possible limitations of this approach in §6.1.)

Figure 6 shows a code snippet to change the behavior of the

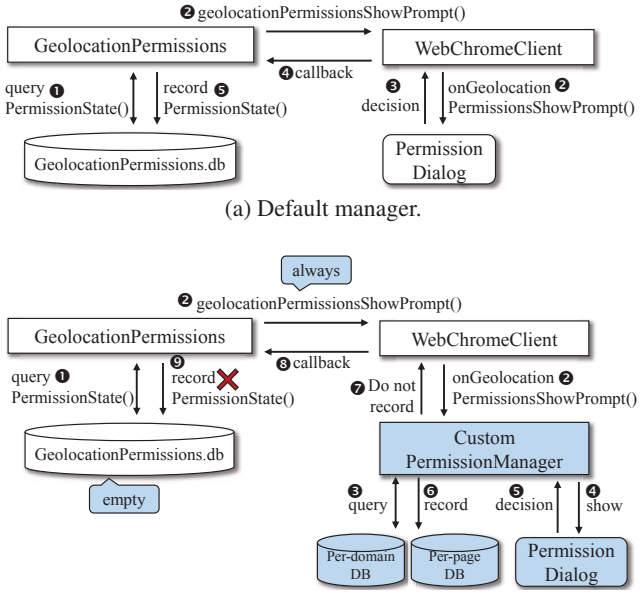
```
WebView webview = (WebView) findViewById(R.id.
    webview);

❶webview.getSettings().setJavaScriptEnabled(true);
webview.getSettings().setGeolocationEnabled(true)
    ;

❷webview.loadUrl("javascript:
   navigator.geolocation.getCurrentPosition
     = function(success) {
         success( {
❸          coords:{
             latitude:"+dLatitude+",
             longitude:"+dLongitude+",
           },
           timestamp:Date.now()
         });
       }"
);

❹webview.loadUrl(URL);
```

Figure 6: Java code to override the `getCurrentPosition()` method using WebView. It enables JavaScript and the Geolocation API (❶), overrides `getCurrentPosition()` (❷), changes location information (❸), and reloads the current web page (❹).



(a) Default manager.

(b) Custom manager.

Figure 7: Procedures of default and custom geolocation permission managers.

`getCurrentPosition()` method. First, the code enables both JavaScript and the Geolocation API in WebView (❶). Also, an application must acquire Android location permissions (ACCESS_FINE_LOCATION, ACCESS_COARSE_LOCATION) or both, and implement an `onGeolocationPermissionsShowPrompt()` callback method to use the Geolocation API [1]. The callback method pops up a permission dialog (§5.3.3) when a browser visits web pages that use the Geolocation API, but are not in the web browser's per-domain nor per-page permission DBs. Second, the code overrides the `getCurrentPostion()` method with a custom function (❷). We use the `loadUrl()` method of WebView and the `javascript` protocol for overriding, which enable arbitrary JavaScript code execution within the scope of the current web page [22, 24]. Third, the code makes the custom function execute a success callback function with degraded geolocation (`dLatitude` and `dLong-itude`) in accordance with a user's choices (❸). Lastly, the code reloads the current web page to activate the injected JavaScript code (❹). The revised code for the `watchPosition()` method is similar.

5.3.2 *Enhancing permission manager*

We implement a custom geolocation permission manager to enhance Android's default geolocation permission manager (`GeolocationPermissions` class). Figure 7a shows a simplified diagram representing how the default geolocation permission manager works. When a web site attempts to use the Geolocation API, the manager first checks whether the permission state of the web site resides in a DB file `GeolocationPermissions.db` (❶). If a corresponding record exists, the manager returns geolocation according to the record. Otherwise, the manager calls `geolocationPermissionsShowPrompt()` (❷) which eventually calls `onGeolocationPermissionsShowPrompt()` to show a permission dialog to obtain a user's decision (❸). The manager receives the decision (❹) and finally records it in the DB (❺).

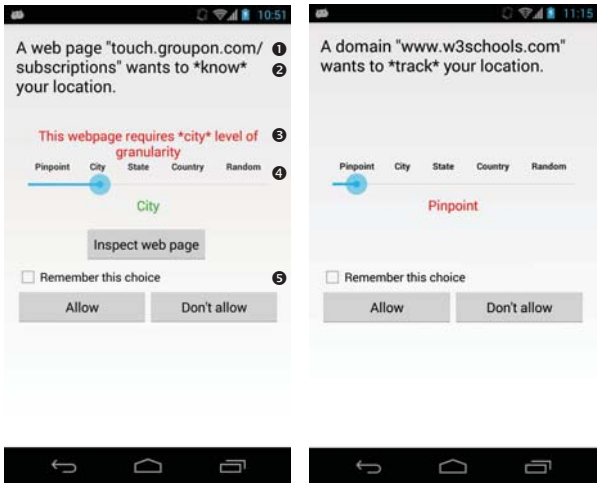
We extend the preceding procedure by adding a custom geolocation permission manager as shown in Figure 7b. Unlike the default manager, the custom manager distinguishes per-page and per-domain permissions, and executes a code to change geolocation (§5.3.1; §5.3.5). The default manager always calls the custom manager when a web site tries to use the Geolocation API because we record no data in the `GeolocationPermissions.db`.

5.3.3 *Designing permission dialog*

We use `onGeolocationPermissionsShowPrompt()` to design a new permission dialog (Figure 8), allowing us to intercept and customize the dialog for requesting geolocation permissions [1]. First, the dialogs show whether a user currently uses per-page (Figure 8a) or per-domain (Figure 8b) permission models (❶). The user can choose one of the permission models in browser settings. Second, they show whether a current web page attempts to use `getCurrentPosition()` or `watchPosition()` (❷). This information is helpful for the user because conventional geolocation dialogs do not distinguish the two different methods. Third, the dialog of the per-page permission model (Figure 8a) displays the estimated location sensitivity of the web page (❸) when the user touches an "Inspect web page" button. The user may refer the result when choosing privacy settings. In contrast, the dialog of the per-domain permission model (Figure 8b) does not have such a button because our scheme does not verify the changes of a domain's content (§5.3.4) Fourth, the dialogs allow the user to choose accuracy options: *pinpoint*, *city*, *state*, *country*, and *random* (❹). If the user selects the random, the geolocation becomes one of well-known cities (e.g., New York, London, or Paris). Lastly, the dialogs allow the user to either temporarily or permanently grant permissions to the web page, or deny providing any geolocation (❺).

5.3.4 *Managing permission DBs*

Our web browser has two permission DBs: a per-page permission DB and a per-domain permission DB. The per-page DB consists of (1) the URL of a web page, (2) whether a user allows or denies the web page to access geolocation, (3) degree of location accuracy the user grants to the web page, (4) the fuzzy hash of the web page for change verification, (5) the geolocation when computing the fuzzy hash, and (6) the (Unix) time when the web browser has visited the



(a) Per-page, position, and inspected.

(b) Per-domain, tracking, and not yet inspected.

Figure 8: Examples of the proposed permission dialogs.

web page lastly to reduce the number of repeated verifications of web page changes.

The per-domain DB is similar to the per-page DB except that it maintains neither fuzzy hash values nor the timestamp of the last visit because we cannot verify changes of all web pages under a domain. To verify whether a domain's content changes, we must compute the fuzzy hash values of all web pages in it. However, this computation has two problems: (1) we cannot enumerate all the web pages under the domain and (2) some of the web pages may change frequently. Moreover, granting permissions to a domain implies that a user trusts all web pages of the domain. Consequently, verifying all web pages under a domain is less meaningful.

Lastly, we must solve a *synchronization problem* between the DBs that occurs when users grant a geolocation permission to a domain whose web pages have already obtained geolocation permissions. To solve this problem, we remove permissions granted to web pages from the per-page DB when their domain obtains a permission.

5.3.5 Decreasing geolocation accuracy

According to a user's choices, our web browser decreases the accuracy of geolocation retrieved from an Android platform. A naïve method to decrease the geolocation accuracy is to adjust GPS coordinates [15], but this approach may lead to address information being wrong. For example, the adjustment can virtually locate a user in a different city, so if the web page that the user is on relies on city-level geolocation, the user receives meaningless information.

Address-aware geolocation manipulation. We propose an address-aware method that cleverly decreases geolocation accuracy, by altering postal address information instead of GPS coordinates. We use the Google Geocoding API [12] that translates GPS coordinates into the corresponding address information and vice versa.

We explain the procedures of the address-aware method with the following conditions: (1) the current GPS coordinates of a device are latitude 40.71751 and longitude -74.00348; and (2) our web browser wants to provide *city-level* geolocation to a web page. First, the web browser attempts to translate the GPS coordinates into the corresponding address information by requesting `http://maps.googleapis.com/maps/api/geocode/json?latlng=40.71751,-74.00348&sensor=true`. When the Google Geocoding API succeeds to process the request, the browser receives a JSON file with estimated geolocation, and extracts detailed address information from the JSON file, e.g., "359 Broadway, New York, NY 10007, USA" stored in the `formatted_address` field.

Next, the web browser truncates the street information and the zip code of the detailed address information, and obtains the corresponding GPS coordinates by requesting `http://maps.googleapis.com/maps/api/geocode/json?address=New+York,+NY,+USA&sensor=true`. When the Google Geocoding API successfully processes the request, the web browser receives a JSON file that includes GPS coordinates, such as latitude 40.71435 and longitude -74.00597, which point to the center of New York City. Lastly, the web browser provides the degraded GPS coordinates to the web page instead of the precise GPS coordinates. As a result, by using the proposed method, users can hide their detailed geolocation while assuring city-level accuracy.

Address caching. Using the Google Geocoding API for manipulating address information has unavoidable network overhead: our web browser should interact with the Google server whenever it visits web pages that demand geolocation. To solve this problem, we use an address caching method. Our web browser caches pinpoint-, city-, state-, and country-level GPS coordinates obtained by the address-aware geolocation manipulation in a DB. When a web page granted either city-, state-, or country-level permissions attempts to access geolocation while our browser's current GPS coordinates are close to some of the cached pinpoint GPS coordinates, the browser uses the corresponding cached information of allowed accuracy. If the browser's current location is not close to any of the cached GPS coordinates, it performs the address-aware geolocation manipulation and updates the DB. We thereby minimize the network overhead of using the Google Geocoding API. A number of studies identify that a user's location history does not suddenly change [26, 28], so address caching is effective.

5.3.6 Suppressing the overflow of permission dialogs

A per-page permission model has a problem: this model can pop up too many permission dialogs when our web browser visits a web site that has many web pages using the Geolocation API. To suppress the overflow of permission dialogs, our web browser automatically applies a per-domain permission model to a web site that pops up per-page permission dialogs too frequently.

5.4 Evaluation

We evaluate our scheme in terms of (1) accuracy, (2) time overhead of the location sensitivity estimation, and (3) overall storage overhead. For the evaluation, we use Galaxy S III and choose 200 web pages among the collected 1196 web pages that are in English and access the geolocation without user interaction.

5.4.1 Accuracy

We verify how accurately our scheme estimates location sensitivity of the selected web pages by comparing the results with manual inspection. The proposed scheme correctly estimates the location sensitivity of 93.5% of the web pages. We further analyze the failed estimations and confirm that the failures are due to web pages that (1) store the obtained geolocation in cookies and retrieve them in future (4.5%), (2) change their content according to both time and geolocation (1.5%), and (3) use IP addresses to inspect locations while ignoring the geolocation that our web browser provides (0.5%). To access the geolocation information stored on such web pages, the proposed scheme may need to manipulate cookies, analyze web page semantics, or cloak IP addresses, respectively.

5.4.2 Time overhead

We measure how much time our scheme spends to inspect location

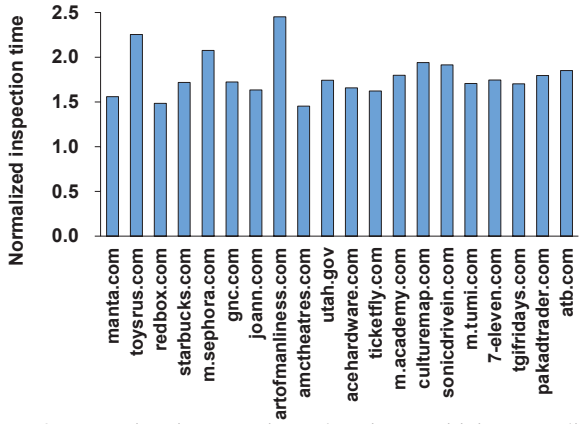


Figure 9: Inspection time to estimate location sensitivity normalized to page loading time.

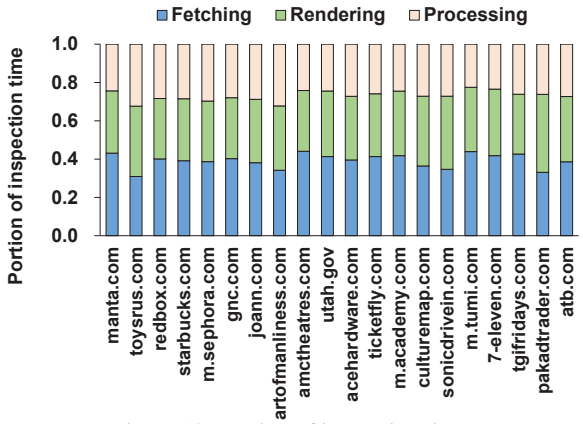


Figure 10: Portion of inspection time.

sensitivity of the selected web pages, which is performed when a user touches an "Inspect web page" button (§5.3.3). Measuring the time overhead, however, demands much time so that we choose 20 web pages out of the selected 200 web pages highly ranked in Alexa Top Sites. Figure 9 shows inspection time to estimate location sensitivity of the 20 web pages normalized to page loading time that consists of fetching and rendering time. On average, the inspection time is 1.8 times longer than the page loading time. Although the time overhead is high, it is tolerable because location-sensitivity inspection would not be frequently performed.

We also measure the processing time that our algorithm spends to estimate location sensitivity (§5.2.3). On average, the processing time only takes ∼27% of the overall inspection time (Figure 10), so the computational overhead of our scheme is acceptable.

5.4.3 Storage overhead

Lastly, we estimate the storage overhead of our scheme. We first check how much storage is necessary to store the per-page information of the 200 web pages in the per-page permission DB: the average size of each record is ∼126 B. Next, we check how much storage is necessary to store address caching information consisting of pinpoint-, city-, state-, and country-level GPS coordinates in a DB. We define an eight-byte field for storing a pair of latitude and longitude values, so the size of each record is 32 B. For example, if our web browser manages 10,000 per-page records and caches up to 10,000 address caching records, storage overhead is only ∼1.5 MB. Therefore, we conclude that our scheme's storage overhead is low.

6. DISCUSSION

6.1 Limitations

We discuss some limitations of the proposed scheme. First, our web browser cannot estimate the location sensitivity of web pages that demand high user interaction (e.g., button clicks or search keywords typings). However, this limitation is less important because such explicit user interaction reminds users that the web sites try to access their geolocation. Users who value their privacy may re-determine whether to grant their geolocation to the web sites.

Second, we cannot apply the proposed scheme to web browsers that disallow JavaScript overriding for security reasons because the proposed scheme should replace the `getCurrentPosition()` and `watchPosition()` methods. If web browser developers accept the proposed scheme and directly modify their web browsers, we can eliminate this limitation.

Third, web sites that recognize geolocation inspection may provide fake web pages to deceive our web browser. Our browser visits a web page several times with various geolocations, so web sites can identify whether the browser inspects their web pages when they compare consecutive HTTP requests. However, this identification can be a huge burden on the web sites, especially when they have a large number of concurrent users. Therefore, we presume that web sites are unwilling to identify our web browser.

Lastly, web sites that only use IP-based geolocation can ignore our scheme. However, a recent study [4] has identified that IP-based geolocation fails when an IP address belongs to cellular networks. For example, the study found that mobile devices hundreds of miles apart can share the same IP address space. Therefore, we anticipate that web sites would use the Geolocation API to obtain correct geolocation instead of using IP-based geolocation.

6.2 Suggestions

We summarize our suggestions to mitigate the privacy problems of the Geolocation API. First, we suggest that the Geolocation API has to allow web sites to control the accuracy of the geolocation that they demand. If the Geolocation API provides accuracy options (e.g., pinpoint, city, state, and country) and returns postal address instead of GPS coordinates, web sites can choose one of the options according to their requirements while eliminating geocoding costs. This improvement mitigates privacy threats while reducing computational and communication overhead of the web sites (§3.4).

Next, we recommend that the Geolocation API needs to employ per-method permission models for separating positioning and tracking because tracking may cause more serious privacy problems than positioning.

Lastly, we propose that the Geolocation API needs to allow web sites to request permissions for either domains or pages. Some web sites serving maps (e.g., Google Maps or Bing Maps) surely demand per-domain permissions because most of their web pages use the Geolocation API. However, other web sites such as shopping mall sites may demand per-page permissions because most of their web pages do not use the Geolocation API.

7. RELATED WORK

Location privacy. Several researchers have proposed various methods to preserve location privacy, which can be used to enhance the location privacy of our scheme. Gruteser and Grunwald [13] establish the concepts of k-anonymity and a trusted-anonymization server in LBSs. Instead of each user's exact GPS coordinates, the server computes a cloaking area including k different users and demands LBSs for the area. Other researchers [7, 23, 31] also consider personalized cloaking areas while assuring k-anonymity.

The k-anonymity-based methods have an important limitation: their cloaking area may contain meaningful locations (e.g., a company, a hospital, and a school), which implies that users are at the locations with a high probability. To solve this problem, Bamba *et al.* [5] propose an l-diversity-based method which enlarges a cloaking area until the area includes l different locations, and Lee *et al.* [20] propose a location-semantics-based method.

Lastly, researchers propose a client-based method to preserve location privacy without a trusted server. Yiu *et al.* [32] allow users to utilize fake locations instead of their precise locations when computing cloaking areas. Ghinita *et al.* [8] use a private information retrieval (PIR) protocol that allows users to retrieve records from a DB without revealing what records they request. Peer-to-peer-based methods [9, 16] also exist.

Geolocation inference attack. Jia *et al.* [17] propose an attack revealing the geolocation of a web browser without a user permission. They exploit a cache-timing attack [6] to recognize what geolocation-dependent resources are stored in the browser cache.

8. CONCLUSION

In this paper, we considered the privacy problems of the HTML5 Geolocation API due to its lack of fine-grained permission and location models. We detected vulnerable web browsers and over-privileged web sites that violate the location privacy of users by conducing case studies. We also proposed a novel scheme to enhance the privacy of using the Geolocation API by supporting fine-grained permission and location models, and by inspecting the location sensitivity of each web page.

In future, we will develop a cloud service to effectively solve the explored problems. We anticipate that the cloud service can minimize inspection overhead by reducing the number of redundant page inspections and can maximize the quality of location sensitivity estimations by using crowdsourcing.

9. REFERENCES

[1] Android Developers. WebChromeClient. `http://developer.android.com/reference/android/webkit/WebChromeClient.html`.

[2] Android Developers. WebView. `http://developer.android.com/reference/android/webkit/WebView.html`.

[3] Apple. UIWebView class reference. `https://developer.apple.com/library/ios/documentation/uikit/reference/UIWebView_Class/Reference/Reference.html`.

[4] M. Balakrishnan, I. Mohomed, and V. Ramasubramanian. Where's that phone?: geolocating IP addresses on 3G networks. In *IMC*, 2009.

[5] B. Bamba, L. Liu, P. Pesti, and T. Wang. Supporting anonymous location queries in mobile environments with PrivacyGrid. In *WWW*, 2008.

[6] E. W. Felten and M. A. Schneider. Timing attacks on web privacy. In *CCS*, 2000.

[7] B. Gedik and L. Liu. Location privacy in mobile systems: A personalized anonymization model. In *ICDCS*, 2005.

[8] G. Ghinita, P. Kalnis, A. Khoshgozaran, C. Shahabi, and K.-L. Tan. Private queries in location based services: Anonymizers are not necessary. In *SIGMOD*, 2008.

[9] G. Ghinita, P. Kalnis, and S. Skiadopoulos. PRIVE: Anonymous location-based queries in distributed mobile systems. In *WWW*, 2007.

[10] globalogiq.com. HTML code search engine - search within HTML source and HTTP headers. `http://globalogiq.com/htmlcodesearch.htm`.

[11] Google. Improve your location's accuracy - maps for mobile help. `https://support.google.com/gmm/answer/3144282?hl=en&ref_topic=3137371`.

[12] Google Developers. Google Geocoding API - Google Maps API web services. `https://developers.google.com/maps/documentation/geocoding`.

[13] M. Gruteser and D. Grunwald. Anonymous usage of location-based services through spatial and temporal cloaking. In *MobiSys*, 2003.

[14] A. T. Holdener. *HTML5 Geolocation*. O'Reilly Media, Inc., 2011.

[15] P. Hornyack, S. Han, J. Jung, S. Schechter, and D. Wetherall. "these aren't the droids you're looking for": Retrofitting Android to protect data from imperious applications. In *CCS*, 2011.

[16] H. Hu and J. Xu. Non-exposure location anonymity. In *ICDE*, 2009.

[17] Y. Jia, X. Dong, Z. Liang, and P. Saxena. I know where you've been: Geo-inference attacks via the browser cache. In *W2SP*, 2014.

[18] J. Jung, A. Sheth, B. Greenstein, D. Wetherall, G. Maganis, and T. Kohno. Privacy oracle: a system for finding application leaks with black box differential testing. In *CCS*, 2008.

[19] J. Kornblum. Identifying almost identical files using context triggered piecewise hashing. *Digital Investigation*, 3:91–97, 2006.

[20] B. Lee, J. Oh, H. Yu, and J. Kim. Protecting location privacy using location semantics. In *KDD*, 2011.

[21] H. Liu, H. Darabi, P. Banerjee, and J. Liu. Survey of wireless indoor positioning techniques and systems. *Systems, Man, and Cybernetics, Part C: Applications and Reviews, IEEE Transactions on*, 37(6):1067–1080, 2007.

[22] T. Luo, H. Hao, W. Du, Y. Wang, and H. Yin. Attacks on WebView in the Android system. In *ACSAC*, 2011.

[23] M. F. Mokbel, C.-Y. Chow, and W. G. Aref. The new Casper: Query processing for location services without compromising privacy. In *VLDB*, 2006.

[24] M. Neugschwandtner, M. Lindorfer, and C. Platzer. A view to a kill: WebView exploitation. In *LEET*, 2013.

[25] A. Restaino. Lightning browser. `https://github.com/anthonycr/Lightning-Browser`.

[26] O. Riva, C. Qin, K. Strauss, and D. Lymberopoulos. Progressive authentication: deciding when to authenticate on mobile phones. In *USENIX Security*, 2012.

[27] U. Shankar and C. Karlof. Doppelganger: Better browser privacy without the bother. In *CCS*, 2006.

[28] E. Shi, Y. Niu, M. Jakobsson, and R. Chow. Implicit authentication through learning user behavior. In *ISC*, 2010.

[29] W3C. Geolocation API specification. `http://www.w3.org/TR/geolocation-API`.

[30] W3Schools. HTML5 geolocation. `http://www.w3schools.com/html/html5_geolocation.asp`.

[31] T. Xu and Y. Cai. Feeling-based location privacy protection for location-based services. In *CCS*, 2009.

[32] M. L. Yiu, C. S. Jensen, X. Huang, and H. Lu. SpaceTwist: Managing the trade-offs among location privacy, query performance, and query accuracy in mobile services. In *ICDE*, 2008.

Differentially Private Data Aggregation with Optimal Utility

Fabienne Eigner[†] Aniket Kate[‡] Matteo Maffei[†]
Francesca Pampaloni[*] Ivan Pryvalov[†]

[†]CISPA, Saarland University, Germany — {eigner,maffei,pryvalov}@cs.uni-saarland.de
[‡]MMCI, Saarland University, Germany — aniket@mmci.uni-saarland.de
[*]francesca.pampaloni@alumni.imtlucca.it

ABSTRACT

Computing aggregate statistics about user data is of vital importance for a variety of services and systems, but this practice has been shown to seriously undermine the privacy of users. Differential privacy has proved to be an effective tool to sanitize queries over a database, and various cryptographic protocols have been recently proposed to enforce differential privacy in a distributed setting, e.g., statical queries on sensitive data stored on the user's side. The widespread deployment of differential privacy techniques in real-life settings is, however, undermined by several limitations that existing constructions suffer from: they support only a limited class of queries, they pose a trade-off between privacy and utility of the query result, they are affected by the answer pollution problem, or they are inefficient.

This paper presents PrivaDA, a novel design architecture for distributed differential privacy that leverages recent advances in secure multiparty computations on fixed and floating point arithmetics to overcome the previously mentioned limitations. In particular, PrivaDA supports a variety of perturbation mechanisms (e.g., the Laplace, discrete Laplace, and exponential mechanisms) and it constitutes the first generic technique to generate noise in a fully distributed manner while maintaining the optimal utility. Furthermore, PrivaDA does not suffer from the answer pollution problem. We demonstrate the efficiency of PrivaDA with a performance evaluation, and its expressiveness and flexibility by illustrating several application scenarios such as privacy-preserving web analytics.

1. INTRODUCTION

Statistics about user data play a significant role in the digital society: they are used daily for improving services, analyzing trends, performing marketing studies, conducting research, and so on. For instance, website owners rely on third-party analytics services to learn statistical information (e.g., gender, age, nationality) about their visitors; electricity suppliers introduced smart meters in order to constantly monitor the user's electricity consumption, which allows them to compute prices based on energy usage trends, to optimize energy distribution, and so on; lastly, service providers often ask their users to evaluate the quality of their services with the goal of publishing the aggregate results.

The acquisition and processing of sensitive information poses serious concerns about the privacy of users. The first problem is how and where user data are aggregated: companies prefer to directly collect and process user data, but this gives them access to a wealth of sensitive information. For instance, web analytics rely on user tracking, allowing aggregators to reconstruct a detailed and precise profile of each individual. The second problem is how to publish aggregate data or statistics in a privacy-preserving manner. For example, researchers demonstrated how precise information about consumer habits can be reconstructed from the electricity consumption information collected by smart meters [36] and how an individual's identity and state of health can be derived from genome wide association studies [40].

Differential Privacy (DP). *Differential privacy* [20] is a popular architecture to define and enforce privacy for statistics on sensitive data. The fundamental idea is that a query on a database is differentially private if the contribution of an individual in the database can only marginally influence the query result. More precisely, the contribution of each single entry to the query result is bounded by a small constant factor, even if all remaining entries are known. A deterministic query can be made differentially private by perturbing the result with a certain amount of noise. The amount of noise depends on the query itself and a variety of perturbation algorithms [22, 33] have been proposed for different queries and datatypes (e.g., numerical and non-numerical data, buckets, histograms, graphs).

Distributed Differential Privacy (DDP). While the original definition of DP focused on a *centralized setting*, in which a database is queried by a curious entity, subsequent work has extended the definition to a *distributed setting* (e.g., [21,35]), in which mutually distrustful, and potentially compromised, parties collaborate to compute statistics about distributed data. In particular, Dwork et al. [21] were the first to suggest the idea of employing *secure multiparty computation (SMPC)* to aggregate and perturb data in a privacy-preserving distributed manner. In general, in a distributed setting, which will be the focus of this paper, the problem to solve is two-fold: (i) how to aggregate data and compute statistics without parties learning each other's data and (ii) how to perturb the result to obtain DP even in the presence of malicious parties deviating from the protocol.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ACSAC '14, December 08 – 12, 2014, New Orleans, LA, USA.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-3005-3/14/12 ...$15.00.
http://dx.doi.org/10.1145/2664243.2664263.

State-of-the-Art. Several specialized ad-hoc cryptographic protocols have been proposed recently to solve the problem of data aggregation in a distributed, privacy-preserving manner, which has enabled the enforcement of DDP in some challenging scenarios, such as smart metering [18] and web analytics [6, 15].

These works can be grouped into two categories: *server-based* and *fully distributed* approaches. The former rely on trusted or *honest-but-curious (HbC)*[1] servers [6, 15, 32] to compute noise perturbation. These servers are assumed not to collude with each other. The latter [5, 14, 38, 39] (see [29] for a comparative survey) propose fully distributed systems to distributively aggregate time-series data, which are directly perturbed by users and then encrypted in such a way that the aggregator is only able to decrypt their noisy sum. Despite the significant progress in this field, the deployment of DDP techniques in real-life systems has so far been undermined by some open challenges, which we discuss below.

Tradeoff between privacy and utility. The existing fully distributed approaches [5,14,38,39] exploit the divisibility properties of certain noise mechanisms and let each party produce a little amount of noise, whose sum yields the noise required to achieve DDP. This solution is affected by a trade-off between privacy and utility, since the amount of noise each user has to add is proportional to the number of tolerated malicious or failing parties: the more malicious parties, the more the noise to be added and, therefore, the less accurate the result. Hence, in order to obtain strong privacy guarantees, each party should assume all others to be malicious, but this leads to an intolerable error ($O(N^2)$, where N is the number of users) as we show in [2]. Relying on a lower honesty threshold, however, not only gives lower privacy guarantees but also leads parties to the paradox of having to agree on how many of them are dishonest!

Lack of generality and scalability. Existing solutions are tailored to individual datatypes and perturbation mechanisms [5–7, 14, 15, 29, 32, 38, 39]. Computing different kinds of queries or employing different perturbation mechanisms requires the usage of different protocols, which rely on different cryptographic schemes, communication patterns, and assumptions. The engineering effort and usability penalty are significant and discourage system administrators from deploying such technologies. Furthermore, existing SMPC-based schemes [7] for two parties cannot simply be extended to a multiparty setting.

Inefficiency. Many schemes [5, 14, 38, 39] involve a significant computational effort on the user's side, making them impractical in several scenarios, e.g., for aggregating data stored on mobile devices with limited computation power.

Answer pollution. Fully distributed schemes [5, 14, 38, 39] in particular, suffer from the answer pollution problem: a single party can substantially pollute the aggregate result by adding excessive noise.

Collusion. Server-based systems [6, 15, 32] rely on strong non-collusion assumptions. In case of collusion, both the noise and the individual user's data are disclosed.

Our Contributions. In this work we present PrivaDA, the first generic architecture for computing differentially private statistics about distributed data. We show how to achieve provable DDP guarantees, while overcoming the previously discussed limitations, by leveraging recently proposed SMPC protocols for floating point numbers [8], fixed point numbers [13], and integers [25]. Our construction refines these schemes, originally designed for the HbC setting, so as to make them secure even in the malicious setting.

The overall differentially private data aggregation computation is organized in two phases: the aggregation phase, in which the clients securely compute the aggregate result, and the perturbation phase, in which this result is perturbed so as to achieve DDP. To improve the performance, the SMPC is actually conducted by computation parties, which collect input shares from each client and perform the required computations. For the perturbation phase, the fundamental idea is to let computation parties jointly compute a random seed (i.e., a random variable in $(0, 1)$), which is then used to produce the required noise.

The distinctive features of our approach are:

Generality. PrivaDA supports a variety of perturbation mechanisms, such as noise drawn from the Laplace and the discrete Laplace (symmetric geometric) distribution as well as the exponential mechanism. Consequently, it is well-suited for a variety of application scenarios. We illustrate generality of our architecture by proposing privacy-preserving designs for web analytics, statistics gathering for anonymity networks, and anonymous surveys.

Strong privacy. As long as at least one of the computation parties is honest, malicious parties can only recover the aggregate result. This is a fundamental difference from other approaches (e.g., [6, 15]), where colluding parties can immediately read the individual's user data. Furthermore, as long as the majority of the computation parties is not colluding, none learns the seed or the noise (i.e., DDP is achieved).

Optimal utility and resistance to pollution attacks. The result is perturbed with the minimal amount of noise required to achieve DDP, irrespectively of the expected number of dishonest users and computation parties. Hence, our protocol provides optimal utility and resistance to answer pollution. We also provide mechanisms to tackle the orthogonal problem of ensuring that the protocol only accepts client inputs that are contained in a set of valid answers.

Efficiency. We implemented the system and conducted a performance evaluation, demonstrating the practicality of our approach. We find the overheads introduced by our privacy-preserving mechanisms to be acceptable for the data aggregation scenarios. Importantly, the client does not have to perform any expensive computation: she just has to provide each computation party with a share of her data and can then go offline, which makes this approach suitable even for mobile devices. Furthermore, PrivaDA supports a large number of clients without any significant performance penalty. Our implementations provide a clear interface for developers to easily integrate PrivaDA in a privacy-preserving data aggregation scenario.

Long version. A long version of this work, containing a detailed comparison of the related works mentioned above. more details, applications, and proofs, is available online [2].

2. BACKGROUND

We now present the concept of differential privacy and the cryptographic building blocks that PrivaDA builds on.

[1] An honest-but-curious party follows the protocol, but tries to learn additional information about the private data.

Differential Privacy (DP). Intuitively, a query is differentially private if it behaves statistically similarly on all databases D, D' differing in one entry, written $D \sim D'$. This means that the presence or absence of each individual database entry does not significantly alter the result of the query. The definition of DP is parameterized by a number ϵ, which *measures* the strength of the privacy guarantee: the smaller ϵ, the smaller the risk to join the database.

Definition 1 (Differential Privacy [20]). *A randomized function $f : \mathcal{D} \to \mathcal{R}$ is* ϵ-differentially private *iff for all databases D, D' such that $D \sim D'$ and every set $S \subseteq \mathcal{R}$, it holds that $Pr[f(D) \in S] \leq e^{\epsilon} \cdot Pr[f(D') \in S]$.*

A deterministic query can be made differentially private by perturbing its result with noise. We describe three popular perturbation mechanisms below. An important insight is that the required amount of noise depends on the query: the more a single entry affects the query result, the stronger the perturbation has to be. This can be expressed using the notion of *sensitivity* of queries, which measures how much a query amplifies the distance between two inputs.

Definition 2 (Sensitivity [22]). *The sensitivity Δf of a query $f : \mathcal{D} \to \mathcal{R}$ is defined as $\Delta f = \max_{\forall D, D' . D \sim D'} |f(D) - f(D')|$.*

Intuitively, queries of low sensitivity map nearby inputs to nearby outputs. For instance, the query *"how many students like the 'Security' lecture?"* has sensitivity 1, since adding or removing one entry affects the result by at most 1.

Laplace noise. The most commonly used sanitization mechanism for queries returning a numerical result is the *Laplace mechanism* [22], i.e., the addition of random noise drawn according to a Laplace distribution $\mathsf{Lap}(\lambda)$ to the correct query result. As shown by Dwork et al. [22], this mechanism provides ϵ-DP, if the parameter λ is set to $\frac{\Delta f}{\epsilon}$. The distribution is both parameterized by the sensitivity of the query and the privacy value ϵ.

Discrete Laplace noise. In some scenarios, it is necessary for a query and its sanitization to return an integer result, e.g., to enable future cryptographic operations based on discrete groups on the result. To this end, Ghosh et al. [28] proposed the *discrete Laplace mechanism* (also called geometric mechanism). It is defined by adding a random integer drawn according to the discrete Laplace distribution $\mathsf{DLap}(\lambda)$ [28], also known as the symmetric geometric distribution, to the correct query result. The discrete Laplace mechanism provides ϵ-DP if the parameter λ is set to $e^{-\frac{\epsilon}{\Delta f}}$ [14]. Again, the distribution is both parameterized by the sensitivity of the query and the privacy value ϵ.

Exponential mechanism. There exist many scenarios in which queries return non-numerical results (e.g., strings or trees). For instance, consider the query "*what is your favorite lecture?*". For such queries, the addition of noise either leads to nonsensical results or is not well-defined. To address this issue, McSherry and Talwar [33] proposed the so-called *exponential mechanism*, which considers queries on databases D that are expected to return a query result a of an arbitrary type $\mathcal{R}$. For our purpose we consider the range $\mathcal{R}$ to be finite, e.g., the set of lectures offered by a university. We refer to each $a \in \mathcal{R}$ as a *candidate*. The mechanism assumes the existence of a *utility function* $q : (\mathcal{D} \times \mathcal{R}) \to \mathbb{R}$ that assigns a real valued score to each possible input-output pair (D, a), which measures the quality of the result a w.r.t. input D. The higher such a score, the better (i.e., more exact) the result. The mechanism $\varepsilon_q^{\epsilon}(D)$ aims at providing the "best" possible result $a \in \mathcal{R}$, while enforcing DP.

Definition 3 (Exponential Mechanism [33]). *For all $q : (\mathcal{D} \times \mathcal{R}) \to \mathbb{R}$ the randomized exponential mechanism $\varepsilon_q^{\epsilon}(D)$ for $D \in \mathcal{D}$ is defined as $\varepsilon_q^{\epsilon}(D) :=$ return $a \in \mathcal{R}$ with probability proportional to $e^{\epsilon q(D,a)}$.*

We now formally state the privacy guarantees of the above mechanisms.

Theorem 1 (DP of Mechanisms [14, 22, 33]). *For all queries $f : \mathcal{D} \to \mathbb{R}$, $g : \mathcal{D} \to \mathbb{Z}$, and $q : (\mathcal{D} \times \mathcal{R}) \to \mathbb{R}$ it holds that the queries $f(x) + \mathsf{Lap}(\frac{\Delta f}{\epsilon})$ and $g(x) + \mathsf{DLap}(e^{-\frac{\epsilon}{\Delta f}})$ and the mechanism $\varepsilon_q^{\frac{\epsilon}{2\Delta q}}(D)$ are ϵ-differentially private.*

Secure Multiparty Computation (SMPC). SMPC enables a set of parties $\mathcal{P} = \{P_1, P_2, \ldots, P_\beta\}$ to jointly compute a function on their private inputs in a privacy-preserving manner [41]. More formally, every party $P_i \in \mathcal{P}$ holds a secret input value x_i, and $P_1, \ldots, P_\beta$ agree on some function f that takes β inputs. Their goal is to compute and provide $y = f(x_1, \ldots, x_\beta)$ to a recipient while making sure that the following two conditions are satisfied: (*i*) *Correctness:* the correct value of y is computed; (*ii*) *Secrecy:* the output y is the only new information that is released to the recipient (see § 5 for a formal definition).

Although the feasibility of SMPC in the computational setting as well as in the information theoretic one is known for more than 25 years, dedicated work to optimize secure realizations of commonly used arithmetic operations has started only in the last few years [8, 11, 13, 17, 25]. However, most of these realizations perform limited SMPC arithmetic operations over input elements belonging only to finite fields [17] or integers [25], and they are thus not well-suited or sufficient for DDP mechanisms. In contrast, we build on SMPCs for algorithms on fixed point numbers [13] and a recent work by Aliasgari et al. [8], who presented SMPC arithmetic algorithms over real numbers represented in floating point form. They also went ahead to propose SMPC protocols for complex tasks such as logarithm and exponentiation computations, and conversion of numbers from floating point form to fixed point or integer form and vice-versa.

Our work starts by observing that their logarithm and exponentiation computations SMPC protocols, combined with the SMPC schemes for the basic integer, fixed point, and floating point number operations, pave the way for a practical design of various perturbation mechanisms for DDP in a completely distributed manner. Nevertheless, to be suitable for our design, we have to enhance and implement this large array of protocols to work against a malicious adversary.

Notations. We assume that secret sharing and the basic SMPC operations take place over a field $\mathbb{F}_q$. Let $[x]$ denotes that the value $x \in \mathbb{F}_q$ is secret-shared among $P_1, \ldots, P_\beta$ such that any $\lceil(\beta+1)/2\rceil$ of those can compute x using the reconstruction Rec protocol. In some steps of our protocols, we employ the sharing $[x]_\beta$, which requires participation from all β parties to reconstruct x. The former is called $(\beta, \lceil(\beta+1)/2\rceil)$ secret sharing, while the latter is called (β, β) secret sharing. Note that $[x] + [y]$, $[x] + c$, and $c[x]$ can be computed by each P_i locally using her shares of

x and y, while the computation of $[x][y]$ is interactive for $(\beta, \lceil(\beta+1)/2\rceil)$ and impossible for (β, β) secret sharing.

Basic SMPC protocols. The following SMPC protocols [8, 13, 17, 25] are used for our DDP mechanisms. The protocol $\mathsf{RandInt}(k)$ allows for the generation of a shared random k-bit value r (i.e., $r \in [0, 2^k)$). The protocols IntAdd and $\mathsf{IntScMul}$ allow for the addition of two shared integers and the multiplication of a shared integer with a scalar, respectively. Analogous operations for fixed point numbers are FPAdd and $\mathsf{FPScMul}$. The protocols FLMul, FLDiv, FLAdd, and FLLT can be used to multiply, divide, add, or compare ($\leq$) two shared floating point numbers respectively. The exponentiation protocol $\mathsf{FLExp2}$ computes the r-th power of 2, the logarithm computation $\mathsf{FLLog2}$ returns the logarithm to base 2. Both operate on shared floating point numbers. The conversion protocols $\mathsf{FL2Int}$ (float-to-int), $\mathsf{Int2FL}$ (int-to-float), $\mathsf{FL2FP}$ (float-to-fixed-point), and $\mathsf{FP2FL}$ (fixed-point-to-float) allow us to convert numbers as described by their names. The protocol $\mathsf{FLRound}$ takes a shared floating point value and depending on an additional argument $\mathsf{mode} \in \{0, 1\}$ outputs the floor or ceil of the argument.

For the exact complexities of the above SMPC protocols, we refer to [2]. Intuitively, additions and scalar multiplications for both integers and fixed point values are non-interactive and consequently very fast, while for floating point values, the algorithms require communication complexity linear in the size of floating point significant. Floating point additions are more costly than even their multiplication counterparts. As expected, the exponentiation and logarithm algorithms are also the most costly and require communication quadratic in the size of significants. The relative efficiency of these SMPC schemes plays a fundamental role in our design.

3. THE PrivaDA ARCHITECTURE

In this section we present the PrivaDA architecture. We first give a general overview of the setup and then present three mechanisms for achieving DP.

The fundamental challenge we had to overcome in order to provide SMPCs for the three noise mechanisms lies in providing SMPC support for drawing random numbers according to various probability distributions. Standard arithmetic SMPC suites we considered only offer support for drawing uniformly random integers. As we will show in § 4, a SMPC for drawing a uniformly random integer can be used to encode the drawing of a uniformly random number between 0 and 1 ($\mathcal{U}_{(0,1]}$). We thus focussed our efforts on finding an encoding that reduces the problem of generating noise according to the distributions employed in three popular sanitization mechanisms to the drawing of a *seed* from $\mathcal{U}_{(0,1]}$.

Setting. We consider a scenario with n users, β computation parties (typically $\beta = 3$, but it can be greater), and an aggregator. Each user P_i has a private input D_i from some domain $\mathcal{D}$. The aggregator would like to compute some aggregate statistics about the users' private inputs, represented by the function $f : \mathcal{D}^n \to \mathcal{R}$. The range $\mathcal{R}$ of f may be a set of numerical values, but not necessarily. The computation parties are responsible for computing and perturbing the aggregate result, which is eventually returned to the data aggregator.

The users communicate with the computation parties through secure and authenticated channels. Furthermore,

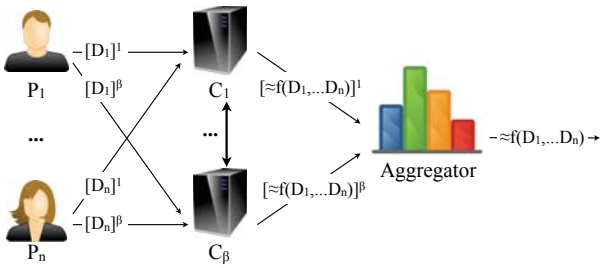


Figure 1: Protocol Flow

the computation parties are pair-wise connected by secure authenticated channels. The users provide the computation parties with shares of their data and can then go offline. The computation parties instead engage in an interactive protocol.

Privacy goals. The aggregate result should be differentially private and neither the aggregator, nor the computation parties, nor the users should learn any further information about the individual users' data.

Attacker model. The data aggregator as well as the users may be corrupted and collude. The SMPC protocols we adopt for the realization of our approach are based on secret sharing: such SMPCs are secure in the malicious setting (i.e., the computation parties may try to deviate from the protocol) but, for certain arithmetic operations, they assume the majority of the computation parties not to collude. We could in principle adopt other kinds of SMPC protocols that do not require this assumption [11, 31, 42], but they are currently less efficient.

Protocol Overview. The protocol proceeds in three steps: (*i*) The users provide the computation parties with shares of their inputs.[2] (*ii*) The computation parties run the SMPC protocol to compute the aggregate statistics and perturb the result. (*iii*) Each computation party gives the aggregator its share of the result, which is reconstructed by the aggregator.

The protocol flow is depicted in Figure 1. In the following we describe three different algorithms to compute queries sanitized with the Laplace, discrete Laplace, and exponential mechanism respectively. For easing the presentation, we consider a class of queries for which $f(D_1, \ldots, D_n) = f(D_1) + \ldots + f(D_n)$. Other arithmetic queries can be implemented in a very similar manner using minor modifications of the presented algorithms, since modern SMPC schemes provide direct support for a large class of arithmetic operations. The algorithms described below do not rely on specific SMPCs: we give one possible efficient instantiation in § 6.

Laplace Mechanism. We now describe an algorithm for calculating the Laplace mechanism (LM) for n inputs. We use the following mathematical results [4,19] that allow us to reduce the problem of drawing a random number according to the Laplace distribution (Lap) to the problem of drawing a uniformly random number between 0 and 1 ($\mathcal{U}_{(0,1]}$) using the exponential distribution (Exp). It holds that $\mathsf{Lap}(\lambda) = \mathsf{Exp}(\frac{1}{\lambda}) - \mathsf{Exp}(\frac{1}{\lambda})$, where $\mathsf{Exp}(\lambda') = \frac{-\ln\ \mathcal{U}_{(0,1]}}{\lambda'}$. Thus,

$$\mathsf{Lap}(\lambda) \quad = \quad \lambda(\ln\ \mathcal{U}_{(0,1]}) - \lambda(\ln\ \mathcal{U}_{(0,1]}). \qquad (1)$$

[2]Our work focusses on guaranteeing the privacy of the user data. To solve the orthogonal problem of pollution attacks and to prevent malicious users from entering wildly incorrect input shares, we can use ZK range proofs [12] (cf. § 4).

In particular, we know that $\lambda = \frac{\Delta f}{\epsilon}$ guarantees DP. We thus define our algorithm for the addition of Laplace noise on n inputs as shown in Table 1a. It takes as input (i) the n real numbers $d_1, \ldots, d_n$ owned by $P_1, \ldots, P_n$ respectively, which correspond to locally executing the query f on each P_i's database D_i ($d_i = f(D_i)$) and (ii) the privacy budget parameter λ, set to $\frac{\Delta f}{\epsilon}$ to guarantee ϵ-DP. The algorithm returns the real $w = (\sum_{i=1}^{n} d_i) + \mathsf{Lap}(\lambda)$ that is computed by summing up all d_i (line 1), drawing a random number according to the distribution $\mathsf{Lap}(\lambda)$ (line 2 - 3) using (1), and adding the sum of the results and the noise (line 4).

Privacy of LM. Since the LM algorithm implements $\sum_{i=1}^{n} d_i + \lambda(\ln\ \mathcal{U}_{(0,1]}) - \lambda(\ln\ \mathcal{U}_{(0,1]}) = \sum_{i=1}^{n} d_i + \mathsf{Lap}(\lambda)$, by Theorem 1 it follows that $\mathsf{LM}(d_1, \ldots, d_n, \lambda)$ is ϵ-differentially private for $\lambda = \frac{\Delta f}{\epsilon}$, where $d_i = f(D_i)$.

Discrete Laplace Mechanism. We now give an algorithm to compute the discrete Laplace mechanism (DLM) on n inputs. The construction follows similar principles as the one for the Laplace mechanism that we presented in the previous section. Additionally, we rely on the following mathematical results [19, 30] to compute the discrete Laplace distribution (DLap) by using the link between DLap, the geometric distribution (Geo), and the exponential distribution (Exp), which can be reduced to the uniform distribution on the interval $(0, 1]$ ($\mathcal{U}_{(0,1]}$) as shown before. It holds that $\mathsf{DLap}(\lambda) = \mathsf{Geo}(1-\lambda) - \mathsf{Geo}(1-\lambda)$, where $\mathsf{Geo}(\lambda') = \lfloor \mathsf{Exp}(-\ln\ (1-\lambda')) \rfloor$. Thus,

$$\mathsf{DLap}(\lambda) = \left\lfloor \frac{1}{\ln\ \lambda} \ln\ \mathcal{U}_{(0,1]} \right\rfloor - \left\lfloor \frac{1}{\ln\ \lambda} \ln\ \mathcal{U}_{(0,1]} \right\rfloor. \quad (2)$$

In particular, for $\lambda = e^{-\frac{\epsilon}{\Delta f}}$, ϵ-DP is guaranteed. The algorithm to add discrete Laplace noise to n inputs is shown in Table 1b. It takes as input (i) the n integer numbers $d_1, \ldots, d_n$ owned by $P_1, \ldots, P_n$ respectively, which correspond to locally executing the query f on each P_i's database D_i ($d_i = f(D_i)$) and (ii) the privacy budget parameter λ, which will be set to $e^{-\frac{\epsilon}{\Delta f}}$ to guarantee ϵ-DP. The algorithm returns the integer $w = (\sum_{i=1}^{n} d_i) + \mathsf{DLap}(\lambda)$, which is computed analogously to the Laplace mechanism, using (2).

Privacy of DLM. The DLM algorithm implements $\sum_{i=1}^{n} d_i + \left\lfloor \frac{1}{\ln\ \lambda} \ln\ \mathcal{U}_{(0,1]} \right\rfloor - \left\lfloor \frac{1}{\ln\ \lambda} \ln\ \mathcal{U}_{(0,1]} \right\rfloor = \sum_{i=1}^{n} d_i + \mathsf{DLap}(\lambda)$. By Theorem 1, $\mathsf{DLM}(d_1, \ldots, d_n, \lambda)$ is ϵ-differentially private for $\lambda = e^{-\frac{\epsilon}{\Delta f}}$, where $d_i = f(D_i)$.

Exponential Mechanism. Our algorithm to compute the exponential mechanism [33] (EM) for n inputs, is inspired by [7], which is however an ad-hoc protocol that is constrained to a 2-party setting. Here, the challenge lies in computing the exponential mechanism in an n-party setting and make it work in the adversarial setting we consider.

Inputs and outputs. The algorithm to compute the EM on the join of n databases is presented in Table 1c. It outputs the candidate $a \in \mathcal{R}$ (where $|\mathcal{R}| = m \in \mathbb{N}$), which is the result of locally executing the desired query f on the databases $D_1, \ldots, D_n$ that are under the control of the participants $P_1, \ldots, P_n$ respectively and sanitizing the joint result using the exponential mechanism. The algorithm takes the following inputs: (i) the data sets $d_1, \ldots, d_n$ belonging to the participants $P_1, \ldots, P_n$ respectively, (ii) the list of candidates $a_1, \ldots, a_m$, and (iii) the privacy parameter λ, which will be set to $\frac{\epsilon}{2\Delta q}$ in order to guarantee ϵ-DP. For the sake of simplicity, we assume each data set $d_i \in \mathcal{D}$ to be a histogram that is the result of locally executing $f(D_i)$. Each histogram is a sequence of m natural numbers $z_1, \ldots, z_m$ that correspond to the frequency of candidates $a_1, \ldots, a_m \in \mathcal{R}$. For instance, for the query $f :=$ "*What is your favorite lecture?*" the sequence of candidates $a_1, \ldots, a_5$ might be Algebra, Logic, Security, Cryptography, Java and the individual data set d_2 of student P_2 who prefers the lecture Security is a histogram of the form $0, 0, 1, 0, 0$. The algorithm outputs the winning candidate a_k drawn according to $\varepsilon_q^{\epsilon}(d_1, \ldots, d_m)$.

Utility function. Our approach is general and can support any arithmetic utility function. For the sake of presentation, we focus on the following utility function $q((z_1, \ldots, z_m), a_i) = z_i$ for all histograms $d = (z_1, \ldots, z_m)$ and candidates $a_1, \ldots, a_m$, returning the frequency z_i of candidate a_i stored in d. For instance, in the above example $q(d_2, \mathsf{Security}) = 1$ and $q(d_2, a_i) = 0$ for all candidates a_i, where $i \in \{1, 2, 4, 5\}$. Since $\Delta q = 1$ it can be omitted from the algorithm. The privacy parameter λ is thus set to $\frac{\epsilon}{2}$.

Random variable. Our goal is to compute the exponential mechanism $\varepsilon_q^{\epsilon}(D)$ for a discrete range $\mathcal{R}$, where $|\mathcal{R}| = m$. The probability mass [7, 33] for the exponential mechanism is defined as $\Pr\left[\varepsilon_q^{\epsilon}(D) = a\right] = \frac{e^{\epsilon q(D,a)}}{\sum_{j=1}^{m} e^{\epsilon q(D,a_j)}}$. As pointed out by Alhadidi et al. [7], drawing a random value according to this distribution corresponds to mapping the above defined probability mass onto the interval $(0, 1]$ and drawing a random number r in $(0, 1]$ to select the interval of the winning candidate. Formally, $r \leftarrow \mathcal{U}_{(0,1]}$ and $r \in (\sum_{k=1}^{j-1} \Pr\left[\varepsilon_q^{\epsilon}(D) = a_k\right], \sum_{k=1}^{j} \Pr\left[\varepsilon_q^{\epsilon}(D) = a_k\right]]$ corresponds to $a_j \leftarrow \varepsilon_q^{\epsilon}(D)$. For instance, assume $\Pr\left[\varepsilon_q^{\epsilon}(D) = a_1\right] = 0.3$ and $\Pr\left[\varepsilon_q^{\epsilon}(D) = a_2\right] = 0.7$. We draw a random number r from $(0, 1]$ and check whether r is in interval $(0, 0.3]$ or in interval $(0.3, 1]$. In this example, the drawing of $0.86 \leftarrow \mathcal{U}_{(0,1]}$ corresponds to $a_2 \leftarrow \varepsilon_q^{\epsilon}(D)$.

It is easy to see that by multiplying with $S := \sum_{j=1}^{m} e^{\epsilon q(D,a_j)}$ the check $r \in (\sum_{k=1}^{j-1} \Pr\left[\varepsilon_q^{\epsilon}(D) = a_k\right], \sum_{k=1}^{j} \Pr\left[\varepsilon_q^{\epsilon}(D) = a_k\right]]$ is equivalent to $r \cdot S \in (\sum_{k=1}^{j-1} e^{\epsilon q(D,a_k)}, \sum_{k=1}^{j} e^{\epsilon q(D,a_k)}]$, since $\Pr\left[\varepsilon_q^{\epsilon}(D) = a\right] \cdot S = e^{\epsilon q(D,a)}$. To optimize complexity, our algorithm will compute the exponential mechanism using the latter version, i.e., by drawing a random number $r \leftarrow \mathcal{U}_{(0,1]}$ and then checking $r \cdot S \in (\sum_{k=1}^{j-1} e^{\epsilon q(D,a_k)}, \sum_{k=1}^{j} e^{\epsilon q(D,a_k)}]$ and returning the candidate a_k for which this checks succeeds. Thus, our main effort lies in computing the necessary interval borders $(\sum_{k=1}^{j-1} e^{\epsilon q(D,a_k)}, \sum_{k=1}^{j} e^{\epsilon q(D,a_k)}]$.

Algorithm. Our algorithm consists of the following steps[3]: (i) Initialize the interval border I_0 (line 1). (ii) Compute the joint histogram $d = d_1 + \ldots + d_n$ (line 3) by adding the frequencies for each individual candidate. (iii) Compute interval borders for candidates (line 4 - 5). (iv) Draw a random value r in $(0, 1]$ (line 6) and multiply this value by $I_n = \sum_{j=1}^{m} e^{\epsilon q(D,a_j)}$, resulting in the scaled random value r'. (v) Check into which of the intervals $(I_{j-1}, I_j]$ the random value r' falls (line 7) by using binary search that returns k such that $I_{k-1} < r' \leq I_k$. (vi) Return the winning candidate a_k (line 8).

[3]Depending on the instantiation of the SMPCs, the steps might be slightly modified, or type conversions added, to provide the best efficiency. The steps presented in this work are designed so as to get the best efficiency based on SMPC schemes employed in PrivaDA.

In: $d_1, \ldots, d_n$; $\lambda = \frac{\Delta f}{\epsilon}$
Out: $(\sum_{i=1}^{n} d_i) + \mathsf{Lap}(\lambda)$

1: $d = \sum_{i=1}^{n} d_i$
2: $r_x \leftarrow \mathcal{U}_{(0,1]}$; $r_y \leftarrow \mathcal{U}_{(0,1]}$
3: $r_z = \lambda(\ln\ r_x - \ln\ r_y)$
4: $w = d + r_z$
5: **return** w

(a) LM

In: $d_1, \ldots, d_n$; $\lambda = e^{-\frac{\epsilon}{\Delta f}}$
Out: $(\sum_{i=1}^{n} d_i) + \mathsf{DLap}(\lambda)$

1: $d = \sum_{i=1}^{n} d_i$
2: $r_x \leftarrow \mathcal{U}_{(0,1]}$; $r_y \leftarrow \mathcal{U}_{(0,1]}$
3: $\alpha = \frac{1}{\ln\ \lambda} = -\frac{\Delta f}{\epsilon}$
4: $r_z = \lfloor \alpha(\ln\ r_x) \rfloor - \lfloor \alpha(\ln\ r_y) \rfloor$
5: $w = d + r_z$
6: **return** w

(b) DLM

In: $d_1, \ldots, d_n$; $a_1, \ldots, a_m$; $\lambda = \frac{\epsilon}{2}$
Out: winning a_k

1: $I_0 = 0$
2: **for** $j = 1$ to m **do**
3: $\quad z_j = \sum_{i=1}^{n} d_i(j)$
4: $\quad \delta_j = e^{\lambda z_j}$
5: $\quad I_j = \delta_j + I_{j-1}$
6: $r \leftarrow \mathcal{U}_{(0,1]}$; $r' = rI_m$
7: $k = \mathsf{binary_search}(r', \leq, I_0, \ldots, I_m)$
8: **return** a_k

(c) EM

Table 1: Algorithms: Sanitization Mechanisms

Privacy of EM. The EM algorithm implements the join of the individual n histograms, the utility function q as defined above, and the drawing of a random value according to $\varepsilon_q^{\lambda}(d_1 + \ldots + d_n)$, which is soundly encoded as explained above. Thus, $\mathsf{EM}(d_1, \ldots, d_n, a_1, \ldots, a_m, \lambda)$ computes $\varepsilon_q^{\lambda}(d_1 + \ldots + d_n)$, where q has sensitivity 1 and by Theorem 1 it follows that $\mathsf{EM}(d_1, \ldots, d_n, a_1, \ldots, a_m, \lambda)$ is ϵ-differentially private for $\lambda = \frac{\epsilon}{2}$, where $d_i = f(D_i)$.

4. INSTANTIATION

In this section, we instantiate the three mechanisms described in the previous section. A technical challenge we had to face was to identify, among all possible instantiations (based, e.g., on different type conversions), the most efficient one based on the currently available SMPC schemes (cf. § 2). The protocols we propose to compute the distributed Laplace, the distributed discrete Laplace, and the distributed exponential mechanism are given in Tables 2, 3, and 4 respectively, and are explained below.

Number Representation. For floating point form, each real value u is represented as a quadruple (v, p, z, s), where v is an ℓ-bit significand, p is a k-bit exponent, z is a bit which is set to 1 when the value $u = 0$, s is a sign bit, and $u = (1-2s) \cdot (1-z) \cdot v \cdot 2^p$. Here, the most significant bit of v is always set to 1 and thus $v \in [2^{\ell-1}, 2^{\ell})$. The k-bit signed exponent p is from the range $\mathbb{Z}_{\langle k \rangle} = (-2^{k-1}, 2^{k+1})$. We use γ to denote the bit-length of values in either integer or fixed point representation, and f to denote the bitlength of the fractional part in fixed point values. Every integer value x belongs to $\mathbb{Z}_{\langle \gamma \rangle} = (-2^{\gamma-1}, 2^{\gamma+1})$, while a fixed point number x is represented as $\bar{x}$ such that $\bar{x} \in \mathbb{Z}_{\langle \gamma \rangle}$ and $x = \bar{x}2^{-f}$. Finally, it is required that $k > \max(\lceil \log(\ell + f) \rceil, \lceil \log(\gamma) \rceil)$ and $q > \max(2^{2\ell}, 2^{\gamma}, 2^{k})$. For ease of exposition, we assume that $\gamma = 2\ell$ for integers and fixed point numbers, and that $f = \frac{\gamma}{2}$ for fixed point numbers.

Input Distribution and Output Reconstruction. We assume that prior to the computation the users $P_1, \ldots, P_n$ create β shares of their respective integer or fixed point inputs $d_1, \ldots, d_n$ in the (β, β)-sharing form and distribute them amongst the β computation parties $C_1, \ldots C_\beta$, so that each party C_k holds a share of each input value $[d_i]$, for $k \in \{1, \ldots, \beta\}$ and $i \in \{1, \ldots, n\}$.

Note that the input values are either integers or fixed point numbers that are only subject to addition operations. Therefore, for security against $\beta - 1$ (instead of $\lfloor \frac{\beta-1}{2} \rfloor$) compromised parties, we perform (β, β) sharing for input values to obtain $[\cdot]_\beta$. For DP noise generation, we still rely on the honest majority assumption and thus use the usual $(\beta, \lceil \frac{\beta+1}{2} \rceil)$ sharing. After the parties $C_1, \ldots, C_\beta$ jointly computed the shared result $[w]_\beta$ of the noise mechanism, the parties collaborate to reconstruct the result $w = \mathsf{Rec}([w]_\beta)$.

In: Shared fixed point form (γ, f) inputs $[d_1]_\beta, \ldots, [d_n]_\beta$; $\lambda = \frac{\Delta f}{\epsilon}$
Out: $w = (\sum_{i=1}^{n} d_i) + \mathsf{Lap}(\lambda)$ in the fixed point form

1: $[d]_\beta = [d_1]_\beta$
2: **for** $i = 2$ to n **do**
3: $\quad [d]_\beta = \mathsf{FPAdd}([d]_\beta, [d_i]_\beta)$
4: $[r_x] = \mathsf{RandInt}(\gamma + 1)$; $[r_y] = \mathsf{RandInt}(\gamma + 1)$
5: $\langle [v_x], [p_x], 0, 0 \rangle = \mathsf{FP2FL}([r_x], \gamma, f = \gamma, \ell, k)$
6: $\langle [v_y], [p_y], 0, 0 \rangle = \mathsf{FP2FL}([r_y], \gamma, f = \gamma, \ell, k)$
7: $\langle [v_{x/y}], [p_{x/y}], 0, 0 \rangle = \mathsf{FLDiv}(\langle [v_x], [p_x], 0, 0 \rangle, \langle [v_y], [p_y], 0, 0 \rangle)$
8: $\langle [v_{ln}], [p_{ln}], [z_{ln}], [s_{ln}] \rangle = \mathsf{FLLog2}(\langle [v_{x/y}], [p_{x/y}], 0, 0 \rangle)$
9: $\langle [v_z], [p_z], [z_z], [s_z] \rangle = \mathsf{FLMul}(\frac{\lambda}{\log_2 e}, \langle [v_{ln}], [p_{ln}], [z_{ln}], [s_{ln}] \rangle)$
10: $[z] = \mathsf{FL2FP}(\langle [v_{z_1}], [p_{z_1}], [z_{z_1}], [s_{z_1}] \rangle, \ell, k, \gamma)$
11: $[w]_\beta = \mathsf{FPAdd}([d]_\beta, [z])$
12: **return** $w = \mathsf{Rec}([w]_\beta)$

Table 2: Protocol: Distributed LM

General Overview. Intuitively, the instantiation for the most part unfolds the mathematical operations used in the algorithms presented in § 3 and replaces them by the corresponding SMPCs for arithmetic operations listed in § 2.

Additions for both integers and fixed point numbers are very fast, while for floating point values, the protocol is costly. We thus choose the n shared data inputs $[d_1], \ldots, [d_n]$ to the mechanisms to be fixed point or integer numbers respectively to lower the cost of adding them together to yield the joint unperturbed query result $[d_1] + \ldots + [d_n]$. We compute the noise values in floating point form as the required logarithm and exponentiation operations are only available for distributed floating point arithmetic. We use the conversion operations FP2FL, FL2Int, Int2FL whenever required.

Random Number Generation. As we have seen in the previous section, our algorithms rely heavily on the generation of a random number in the interval $(0, 1]$ drawn according to the uniform distribution $\mathcal{U}_{(0,1]}$. Unfortunately, the SMPC suite we consider does not include such a function. Hence we devised an SMPC protocol that is based on the idea of encoding such a random number generation using the primitive RandInt for the generation of a random integer (e.g., cf. steps 4 and 5 in Table 2. We first generate a shared $(\gamma + 1)$-bit integer $[r_x]$ using the SMPC primitive RandInt. We then consider this integer to be the fractional part of fixed point number, whose integer part is 0 (by choosing $f = \gamma$). Afterwards, the fixed point number is converted to floating point by using the function FP2FL and disregarding the shared sign bit. Notice that strictly speaking,

In: Shared integer number (γ) inputs $[d_1]_\beta, \ldots, [d_n]_\beta$; $\lambda = e^{-\frac{\epsilon}{\Delta f}}$; $\alpha = \frac{1}{\ln \lambda \cdot \log_2 e}$

Out: integer $w = (\sum_{i=1}^{n} d_i) + \mathsf{DLap}(\lambda)$

1: $[d]_\beta = [d_1]_\beta$
2: **for** $i = 2$ to n **do**
3: $\quad [d]_\beta = \mathsf{IntAdd}([d]_\beta, [d_i]_\beta)$
4: $[r_x] = \mathsf{RandInt}(\gamma + 1)$; $[r_y] = \mathsf{RandInt}(\gamma + 1)$
5: $\langle [v_x], [p_x], 0, 0 \rangle = \mathsf{FP2FL}([r_x], \gamma, f = \gamma, \ell, k)$
6: $\langle [v_y], [p_y], 0, 0 \rangle = \mathsf{FP2FL}([r_y], \gamma, f = \gamma, \ell, k)$
7: $\langle [v_{lnx}], [p_{lnx}], [z_{lnx}], [s_{lnx}] \rangle = \mathsf{FLLog2}(\langle [v_x], [p_x], 0, 0 \rangle)$
8: $\langle [v_{lny}], [p_{lny}], [z_{lny}], [s_{lny}] \rangle = \mathsf{FLLog2}(\langle [v_y], [p_y], 0, 0 \rangle)$
9: $\langle [v_{\alpha lnx}], [p_{\alpha lnx}], [z_{\alpha lnx}], [s_{\alpha lnx}] \rangle =$
$\quad \mathsf{FLMul}(\alpha, \langle [v_{lnx}], [p_{lnx}], [z_{lnx}], [s_{lnx}] \rangle)$
10: $\langle [v_{\alpha lny}], [p_{\alpha lny}], [z_{\alpha lny}], [s_{\alpha lny}] \rangle =$
$\quad \mathsf{FLMul}(\alpha, \langle [v_{lny}], [p_{lny}], [z_{lny}], [s_{lny}] \rangle)$
11: $\langle [v_{z_1}], [p_{z_1}], [z_{z_1}], [s_{z_1}] \rangle =$
$\quad \mathsf{FLRound}(\langle [v_{\alpha lnx}], [p_{\alpha lnx}], [z_{\alpha lnx}], [s_{\alpha lnx}] \rangle, 0)$
12: $\langle [v_{z_2}], [p_{z_2}], [z_{z_2}], [s_{z_2}] \rangle =$
$\quad \mathsf{FLRound}(\langle [v_{\alpha lny}], [p_{\alpha lny}], [z_{\alpha lny}], [s_{\alpha lny}] \rangle, 0)$
13: $[z_1] = \mathsf{FL2Int}(\langle [v_{z_1}], [p_{z_1}], [z_{z_1}], [s_{z_1}] \rangle, \ell, k, \gamma)$
14: $[z_2] = \mathsf{FL2Int}(\langle [v_{z_2}], [p_{z_2}], [z_{z_2}], [s_{z_2}] \rangle, \ell, k, \gamma)$
15: $[w]_\beta = \mathsf{IntAdd}([d]_\beta, \mathsf{IntAdd}([z_1], -[z_2]))$
16: **return** $w = \mathsf{Rec}([w]_\beta)$

Table 3: Protocol: Distributed DLM

this generates a random number in $[0, 1)$. We can achieve a transition to the expected interval $(0, 1]$ by slightly modifying the conversion primitive FP2FL such that the shared $[0]$ is replaced by the sharing of $[1]$ in step 3 [8, § 5]. We could avoid the modification of FP2FL and instead transition into the desired interval by subtracting the random number from 1, but this requires an additional costly addition step.

Exponentiation and Logarithm. The work by Aliasgari et al. [8] provides SMPCs for computing exponentiation with base 2 (FLExp2) and logarithm to base 2 (FLLog2). Since we often require exponentiation and logarithm to a base $b \neq 2$, we use the following mathematical properties $b^a = 2^{a(\log_2 b)}$ and $\log_b x = \frac{\log_2 x}{\log_2 b}$ to compute exponentiation and logarithm for any base b. For instance, steps 8 - 9 in Table 2 and steps 7 - 8 in Table 4 use the above equations to compute logarithm and exponentiation to base e respectively.

Distributed (Discrete) Laplace Mechanism. The protocols to compute the distributed Laplace and distributed discrete Laplace mechanism are shown in Tables 2 and 3 respectively. While the former expects fixed point numbers as inputs, the latter expects integers. Both protocols follow along the same lines, but while the Laplace mechanism can use the simplification $\ln r_x - \ln r_y = \ln \frac{r_x}{r_y}$ and thus reduce the number of necessary logarithm operations FLLog2 as well as the number of follow-up operations, this is not possible for its discrete counterpart due to the floor operations FLRound.

Distributed Exponential Mechanism. The protocol to compute the distributed exponential mechanism using SMPCs is presented in Table 4. Each shared input $[d_i]$ consists of an integer array of size m, representing the histogram of participant P_i. The instantiation follows the steps of the algorithm presented in Table 1c in § 3 by using the insights and techniques we presented in this section. We straightforwardly implement the binary search to find the winning interval/candidate in steps 13-17. Note that we need a slightly simplified version of the FLLT protocol that outputs a value $\{0, 1\}$ that does not need to be shared, thus allowing us to output $w_{j_{\mathsf{min}}}$ immediately without reconstruction, which

In: $[d_1], \ldots, [d_n]$; the number m of candidates; $\lambda = \frac{\epsilon}{2}$

Out: m-bit w, s.t. smallest i for which $w(i) = 1$ denotes winning candidate a_i

1: $I_0 = \langle 0, 0, 1, 0 \rangle$
2: **for** $j = 1$ to m **do**
3: $\quad [z_j]_\beta = 0$
4: $\quad$ **for** $i = 1$ to n **do**
5: $\quad\quad [z_j]_\beta = \mathsf{IntAdd}([z_j]_\beta, [d_i(j)]_\beta)$
6: $\quad \langle [v_{z_j}], [p_{z_j}], [z_{z_j}], [s_{z_j}] \rangle = \mathsf{Int2FL}([z_j]_\beta, \gamma, \ell)$
7: $\quad \langle [v_{z'_j}], [p_{z'_j}], [z_{z'_j}], [s_{z'_j}] \rangle =$
$\quad\quad \mathsf{FLMul}(\lambda \cdot \log_2 e, \langle [v_{z_j}], [p_{z_j}], [z_{z_j}], [s_{z_j}] \rangle)$
8: $\quad \langle [v_{\delta_j}], [p_{\delta_j}], [z_{\delta_j}], [s_{\delta_j}] \rangle =$
$\quad\quad \mathsf{FLExp2}(\langle [v_{z'_j}], [p_{z'_j}], [z_{z'_j}], [s_{z'_j}] \rangle)$
9: $\quad \langle [v_{I_j}], [p_{I_j}], [z_{I_j}], [s_{I_j}] \rangle = \mathsf{FLAdd}(\langle [v_{I_{j-1}}], [p_{I_{j-1}}],$
$\quad\quad [z_{I_{j-1}}], [s_{I_{j-1}}] \rangle, \langle [v_{\delta_j}], [p_{\delta_j}], [z_{\delta_j}], [s_{\delta_j}] \rangle)$
10: $[r] = \mathsf{RandInt}(\gamma + 1)$
11: $\langle [v_r], [p_r], 0, 0 \rangle = \mathsf{FP2FL}([r], \gamma, f = \gamma, \ell, k)$
12: $\langle [v'_r], [p'_r], [z'_r], [s'_r] \rangle =$
$\quad \mathsf{FLMul}(\langle [v_r], [p_r], 0, 0 \rangle, \langle [v_{I_m}], [p_{I_m}], [z_{I_m}], [s_{I_m}] \rangle)$
13: $j_{\mathsf{min}} = 1$; $j_{\mathsf{max}} = m$
14: **while** $j_{\mathsf{min}} < j_{\mathsf{max}}$ **do**
15: $\quad j_{\mathsf{M}} = \lfloor \frac{j_{\mathsf{min}} + j_{\mathsf{max}}}{2} \rfloor$
16: $\quad$ **if** $\mathsf{FLLT}(\langle [v_{I_{j_{\mathsf{M}}}}], [p_{I_{j_{\mathsf{M}}}}], [z_{I_{j_{\mathsf{M}}}}], [s_{I_{j_{\mathsf{M}}}}] \rangle, \langle [v'_r], [p'_r], [z'_r], [s'_r] \rangle)$
$\quad$ **then**
17: $\quad\quad j_{\mathsf{min}} = j_{\mathsf{M}} + 1$ **else** $j_{\mathsf{max}} = j_{\mathsf{M}}$
18: **return** $w_{j_{\mathsf{min}}}$

Table 4: Protocol: Distributed EM

would require additional interactions. We can also improve performance by running m instances of steps 3-8 in parallel.

Mechanisms in the Malicious Setting. In order to achieve DP against malicious computation parties, we had to strengthen the SMPC protocols so as to make them resistant to computation parties deviating from the protocol [12, 27]. Intuitively, to maintain security, one has to enforce the following TWO additional properties: (i) The protocol-instance observations of honest parties are consistent with each other; (ii) Every party proves that each step of its computation was performed correctly.

Given the real-world impracticality of information-theoretically secure channels and subsequently information-theoretically secure SMPC protocols, in the malicious setting we shift to the computational setting. In particular, we employ a computational verifiable secret sharing scheme (VSS) [10, 16, 37] instead of the basic secret sharing scheme to achieve the first property. For the second property, we introduce zero-knowledge (ZK) proofs such that a party can prove that a correct secret value is shared among the parties [27] and that shared secret values satisfy some mathematical conditions (e.g., they are in a pre-defined range) [12]. We note that these two changes are not sufficient to maintain *liveness*: compromised parties may crash to stop the computation. Although a stronger resiliency condition of $\beta \geq 3t + 1$ (instead of $\beta \geq 2t + 1$) can ensure the protocol completion, the honest parties can always determine which parties crashed during a computation and replace those before restarting the protocol. Therefore, for our implementation, we stick to $\beta \geq 2t + 1$ parties.

Limitations of Finite-Precision Instantiations. While the theoretical definition of sanitization mechanisms for DP operates on reals $r \in \mathbb{R}$ (or integers $z \in \mathbb{Z}$), the implementations of such mechanisms have to approximate these mathematical abstractions by finite-precision representations due to the physical limitations of actual machines. This mis-

match has been shown to give rise to several attacks, as shown by Mironov [34] and Gazeau et al. [26] (cf. [2] for more details). The techniques proposed in these works to prevent such attacks rely on arithmetic operations that can be implemented using our arithmetic SMPCs. This allows us to make PrivaDA immune to these attacks. For the sake of simplicity, we omitted this extension from our presentation.

5. SECURITY ANALYSIS

We first recall the standard notion of t-secrecy for SMPC, which is formulated as in [8] except for a small modification to accommodate the computation parties. The following definitions refer to computation parties $\mathcal{C} = \{C_1, \ldots, C_\beta\}$ engaging in a protocol Π that computes function $y = f(D)$, where $D = D_1, \ldots, D_n$ and D_i denotes the input of party P_i and $y \in \mathcal{R}$ is the output.

Definition 4 (View). *C_i's view consists of its shares $\{[D]\}_{C_i}$ and its internal random coin tosses r_i, as well as the messages M exchanged with the other parties during the protocol execution induced by the other parties' random coin tosses h: i.e., $\mathsf{VIEW}_{\Pi(D,h)}(C_i) = (\{[D]\}_{C_i}, r_i, M)$. $\mathsf{VIEW}_{\Pi(D)}(C_i)$ denotes the corresponding random function conditioned to the other parties' coin tosses.*

Definition 5 (t-Secrecy). *A protocol Π is t-private in the presence of HbC adversaries if for each coalition $I = \{C_{i_1}, C_{i_2}, \ldots, C_{i_t}\} \subset \mathcal{C}$ of HbC computation parties of size $t < \beta/2$, there exists a probabilistic polynomial time simulator S_I such that $\{S_I(\{[D]\}_I, f(D))\} \equiv \{\mathsf{VIEW}_{\Pi(D,h)}(I), y\}$. Here, $\{[D]\}_I = \bigcup_{C \in I}\{[D]\}_C$, $\equiv$ denotes indistinguishability, $\mathsf{VIEW}_{\Pi(D,h)}(I)$ the combined view of the parties in I, and h the coin tosses of the parties in $\mathcal{C} \backslash I$.*

Let $\mathcal{V}_\Pi$ be the set of all possible views for the protocol Π. We now formally define the notion of DDP for protocols, close in spirit to the one introduced in [23]. Here, two vectors $D, D' \in \mathcal{D}^n$ are said to be *neighbors* if they differ in exactly one coordinate, which corresponds to the scenario in which exactly one user changes her input.

Definition 6 (ϵ-DDP). *We say that the data sanitization procedure implemented by a randomized protocol Π among β computation parties $\mathcal{C} = \{C_1, \ldots, C_\beta\}$ achieves ϵ-DDP w.r.t. a coalition $I \subset \mathcal{C}$ of HbC computation parties of size t, if the following condition holds: for any neighboring input vectors $D, D' \in \mathcal{D}^n$ and any possible set $\mathcal{S} \subseteq \mathcal{V}_\Pi$, $Pr[\mathsf{VIEW}_{\Pi(D)}(I) \in \mathcal{S}] \leq e^\epsilon Pr[\mathsf{VIEW}_{\Pi(D')}(I) \in \mathcal{S}]$ holds.*

For the malicious setting, the coalition I of HbC parties in Definitions 5 and 6 is replaced by an equal-sized coalition I^{M} of malicious computationally-bounded (for a security parameter κ) parties and the protocol Π is replaced by a computational protocol Π^{M} fortified against malicious attackers with the same t-secrecy property. The above DDP relation changes to an indistinguishability-based computational DDP (IND-DDP) [14, 35] relation with a negligible function $\mathrm{negl}(\kappa)$ such that $Pr[\mathsf{VIEW}_{\Pi^{\mathsf{M}}(D)}(I^{\mathsf{M}}) \in \mathcal{S}] \leq e^\epsilon Pr[\mathsf{VIEW}_{\Pi^{\mathsf{M}}(D')}(I^{\mathsf{M}}) \in \mathcal{S}] + \mathrm{negl}(\kappa)$.

We now state our main theorems on ϵ-DDP in the HbC model, and ϵ-IND-DDP in the malicious model and refer the readers to the full version [2] for proof sketches.

Theorem 2 (ϵ-DDP). *Let $\epsilon > 0$. In the HbC setting, our distributed LM, DLM, and EM protocols achieve ϵ-DDP w.r.t. any HbC coalition $I \subset \mathcal{C}$ of size $t < \beta/2$.*

Theorem 3 (ϵ-IND-DDP). *Let $\epsilon > 0$ and κ be a sufficiently large security parameter. In the malicious setting, our distributed LM, DLM, and EM protocols achieve ϵ-IND-DDP w.r.t. any malicious coalition $I^{\mathsf{M}} \subset \mathcal{C}$ of size $t < \beta/2$, under the strong RSA and decisional Diffie-Hellman assumptions for parameter κ.*

6. PERFORMANCE ANALYSIS

Aliasgari et al. [8] microbenchmarked the performance for most of the required arithmetic SMPC protocols in the HbC setting for three computation parties. However, we could not successfully execute several library functions and their library does not handle the malicious setting. Hence we developed the complete SMPC library for both the HbC and malicious setting from scratch. Here, we present our SMPC implementation for integer, fixed point, and floating point arithmetic and measure the performance costs for the distributed LM, DLM, and EM protocols in both settings.

Implementation. We implement all SMPC protocols discussed in §2 as well as our DDP mechanisms as a multi-threaded object-oriented C++ code to support any number (≥ 3) of computation parties in the HbC and malicious settings. Our implementation uses the GMP library [1] for all finite field computations, the Relic toolkit [9] for elliptic curve cryptographic (ECC) constructions, and the Boost [3] and OpenSSL libraries for secure communication. Our numeric SMPC libraries can be of independent interest to other distributed computation scenarios, and our complete code base is available online [2] in an anonymized form.

Experimental Setup. The experiments are performed over an 3.20 GHz (Intel i5) Linux machine with 16 GB RAM, using a 1 Gbps LAN. We run experiments for the 3-party (i.e., $\beta = 3$ and $t = 1$), and 5-party (i.e., $\beta = 5$ and $t = 2$) computation setting. The floating point numbers employed in the experiments have a bit-length of $\ell = 32$ for significands and $k = 9$ for (signed) exponents, which gives a precision of up to 2^{-256}. For integers and fixed point numbers, we use a bit-length of $\gamma = 64$, where $f = 32$ for fixed point numbers. It gives a precision of 2^{-32} for the latter. The experiments use finite fields of size 177 bits for integers, 208 bits for fixed point numbers, and 113 bits for floating point significands. For floating point exponents, as well as sign and zero bits, significantly smaller fields suffice. In contrast to [8], we do not employ batching (which improves average computations times) since our distributed mechanisms call the individual arithmetic SMPC functions only a few times. To determine an average performance, we run the experiments ten times for both parameter sets. In Table 5, we show our results for all required SMPC functionalities in the HbC and malicious settings; in particular, we include the computation time for single 3-party and 5-party arithmetic SMPC operations measured in seconds.

Cost Analysis (HbC setting). As expected, the logarithm and exponentiation SMPC are the most expensive operations, and they will drive our distributed mechanism cost analysis. Our protocols also use Rec, IntAdd, FPAdd, RandInt SMPC, but we do not include them in Table 5 as they are local operations that can be performed significantly faster than the rest of the protocols. Next, we determine the average performance costs for our distributed LM, DLM, and EM protocols for ($\beta = 3$, $t = 1$) computation parties and $100,000$ users. The distributed LM protocol has a compu-

Type	Protocol	HbC		Malicious	
		$\beta = 3$, $t = 1$	$\beta = 5$, $t = 2$	$\beta = 3$, $t = 1$	$\beta = 5$, $t = 2$
Float	FLAdd	0.48	0.76	14.6	29.2
	FLMul	0.22	0.28	3.35	7.54
	FLScMul	0.20	0.28	3.35	7.50
	FLDiv	0.54	0.64	4.58	10.2
	FLLT	0.16	0.23	2.82	6.22
	FLRound	0.64	0.85	11.4	23.4
Convert	FP2FL	0.83	1.21	25.7	50.9
	Int2FL	0.85	1.22	25.7	50.9
	FL2Int	1.35	1.91	26.3	54.3
	FL2FP	1.40	1.96	26.8	55.3
Log	FLLog2	12.0	17.0	274	566
Exp	FLExp2	7.12	9.66	120	265

Table 5: Performance of SMPC operations measured in sec

tation cost of 15.5 sec, while the distributed DLM protocol requires around 31.3 sec. The better efficiency of the LM mechanism is due to the fact that we halved the number of costly logarithm operations FLLog2 and necessary follow-up operations by using the property $\ln r_x - \ln r_y = \ln \frac{r_x}{r_y}$, which is not possible for its discrete counterpart due to the necessary floor operations FLRound. The computation cost of the distributed EM protocol linearly depends on the number $m = |\mathcal{R}|$ of result candidates. For instance, for $m = 5$, the cost of computation is 42.3 sec.

For larger numbers of computation parties β, one can extrapolate the performance from our analysis for $(\beta = 3, t = 1)$ and $(\beta = 5, t = 2)$. Even for $\beta \approx 100$, we expect the distributed LM and DLM protocols to take about a few hundred seconds in the HbC setting. We also compared our experimental results with [8]. We could not reproduce their results, possibly due to the introduced memory management and correctness verifications.

Cost Analysis (Malicious Setting). As expected, the computations times for the SMPC operations secure against an active adversary are significantly higher (around 15-20 times) than those of the operations secure against an HbC adversary. The average performance costs for our distributed LM, DLM, and EM protocols for ($\beta = 3$, $t = 1$) computation parties and $100,000$ users in the malicious setting are as follows: The distributed LM protocol has an average computation cost of 344 sec, while the distributed DLM protocol requires 477 sec. The cost of the distributed EM protocol, for $m = 5$ result candidates, is 652 sec.

We stress that these operations are performed by computation parties, and that there are no critical timing restrictions on DDP computations in most real-life scenarios, such as web analytics. Nevertheless, we expect 1 order of magnitude performance gain in the HbC as well as the malicious setting by employing high performance computing servers. Furthermore, since users have to simply forward their shared values to the computation parties, which is an inexpensive operation ($<$ 1 msec in the HbC setting and a couple of milliseconds in the malicious setting), we believe that these numbers demonstrate the practicality of PrivaDA even in a setting where clients are equipped with computationally limited devices, such as smartphones.

7. APPLICATION SCENARIOS

We showcase the flexibility of our architecture by briefly discussing how PrivaDA can be used to improve the state-of-the-art in two different application scenarios. An additional scenario (anonymous surveys) is given in the long version [2].

Web Analytics. Web analytics consist in the measurement, collection, analysis, and reporting of Internet data about users visiting a website. For instance, data can include user demographics, browsing behavior, and information about the clients' systems. This information is important for publishers, because it enables them to optimize their site content according to the users' interests, for advertisers, because it allows them to target a selected population, and many other parties, which we will refer to as *analysts*.

State-of-the-Art. In order to obtain aggregated user information websites commonly use third party web analytics services, called *aggregators*, which however track individual users' browsing behavior across the web, thereby violating their privacy. Newer systems, e.g., a series of non-tracking web analytics systems [6,15] proposed by Chen et al., provide users with DP guarantees but rely on strong non-collusion assumptions. Should a collusion happen, not only the noise but also the individual user's data would be disclosed.

Protocol design in PrivaDA. The computation parties are operated by third-parties, which are possibly paid by the aggregator. In order to avoid multiple responses by each user without relying on a public key infrastructure, which is unrealistic in this setting, we add an initial step to the protocol. The publisher signs and gives each visiting user a different token, along with one or more queries and an associated expiry time (within which the result has to be computed). The user sends the tokens to the computation parties, together with their answer shares, so that the computation parties are able to detect duplicates and to discard them before the aggregation. The users have just to submit their shares and can then go offline. Finally, the support for a variety of perturbation mechanisms enables the execution of different kinds of analytical queries.

Traffic Statistics for Anonymous Communication Networks (ACNs). Given their anonymous nature, it is hard to collect egress traffic statistics from ACNs, such as Tor, without violating the privacy of users. Such statistics are interesting to both designers and researchers, which might for instance want to know how much of the network traffic is made up by people trying to circumvent censorship.

State-of-the-Art. Elahi et al. recently proposed PrivEx [24], a system for collecting differentially private statistics on ACNs traffic in predefined slots of time (epochs). Their work provides two ad-hoc protocols that rely on secret sharing and distributed decryption respectively. Nevertheless, to tolerate even an HbC adversary PrivEx has to compromise on the utility or the epoch duration.

Protocol design in PrivaDA. We can easily apply PrivaDA to the problem of collecting anonymous traffic statistics: we simply let the ACN egress nodes, which relay traffic between the ACN and the destination websites, count the accesses to the different destinations that they relayed. After a fixed epoch, they then share their individual counts among mutually distrustful computation parties (e.g., privacy organizations, research centers, and service providers), which jointly compute the overall egress traffic in a privacy-preserving manner with optimal utility.

8. CONCLUSION AND FUTURE WORK

Although it is a long-held belief that SMPCs may be used to generically design differentially private data aggregation

protocols, such an approach has not been undertaken so far due to the inefficiency of generic constructions. In this work we demonstrated the viability of such an approach by designing an SMPC architecture that constitutes not only a generic, but also a practical building block for designing a variety of privacy-preserving data aggregation protocols. In particular, the computational effort on the client side is negligible, making PrivaDA suitable even for computationally limited devices, e.g., smartphones. In contrast to previous works, PrivaDA supports a variety of perturbation mechanisms, offers strong privacy guarantees as well as optimal utility, and is resistant to answer pollution attacks. Furthermore, PrivaDA can support a large number of clients without any significant performance penalty.

For the security of certain arithmetic operations, the SMPC schemes we use assume that the majority of the computation parties are not colluding. This assumption is present in any secret sharing-based SMPC scheme. We mention that stronger assumptions are common in the DP literature and deemed appropriate in many realistic scenarios. There exist SMPCs based on other techniques (homomorphic encryption, oblivious transfer, etc.) that do not assume an honest majority (e.g., [31,42]), but that are currently less efficient. Nevertheless, since PrivaDA is parameterized over the underlying arithmetic SMPCs, it can take immediate advantage of the rapid progress in this research field.

As a future work, we indeed intend to investigate the usage of alternative SMPC schemes and to explore the integration of more sanitization mechanisms. To foster further progress in this field, we made the implementation of PrivaDA publicly available [2]: to the best of our knowledge, this is the first publicly available SMPC implementation that supports a variety of arithmetic operations in the malicious setting.

Acknowledgements. This work was supported by the German research foundation (DFG) through the Emmy Noether program and the Cluster of Excellence on Multimodal Computing and Interaction (MMCI), and by the German Federal Ministry of Education and Research (BMBF) through the Center for IT-Security, Privacy and Accountability (CISPA).

9. REFERENCES

[1] GMP: The GNU Multiple Precision Arithmetic Library. `http://gmplib.org`.

[2] PrivaDA Project Page: Full Version + Our SMPC Library. `http://crypsys.mmci.uni-saarland.de/projects/ArithmeticSMPC`.

[3] The Boost C++ Libraries. `http://www.boost.org`.

[4] M. Abramowitz and I. A. Stegun. *Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables*. Dover, 1964.

[5] G. Ács and C. Castelluccia. I have a DREAM! (DiffeRentially privatE smArt Metering). In *IH'11*, pages 118–132, 2011.

[6] I. E. Akkus, R. Chen, M. Hardt, P. Francis, and J. Gehrke. Non-tracking Web Analytics. In *CCS'12*, pages 687–698, 2012.

[7] D. Alhadidi, N. Mohammed, B. C. M. Fung, and M. Debbabi. Secure Distributed Framework for Achieving ϵ-Differential Privacy. In *PETS'12*, pages 120–139, 2012.

[8] M. Aliasgari, M. Blanton, Y. Zhang, and A. Steele. Secure Computation on Floating Point Numbers. In *NDSS'13*, 2013.

[9] D. F. Aranha and C. P. L. Gouvêa. RELIC is an Efficient LIbrary for Cryptography. `http://code.google.com/p/relic-toolkit/`.

[10] M. Backes, A. Kate, and A. Patra. Computational Verifiable Secret Sharing Revisited. In *ASIACRYPT'11*, pages 590–609, 2011.

[11] A. Ben-David, N. Nisan, and B. Pinkas. FairplayMP: A System for Secure Multi-party Computation. In *CCS'08*, pages 257–266, 2008.

[12] F. Boudot. Efficient Proofs that a Committed Number Lies in an Interval. In *EUROCRYPT'00*, pages 431–444, 2000.

[13] O. Catrina and A. Saxena. Secure Computation With Fixed-Point Numbers. In *FC'10*, pages 35–50, 2010.

[14] T.-H. H. Chan, E. Shi, and D. Song. Privacy-Preserving Stream Aggregation with Fault Tolerance. In *FC'12*, pages 200–214, 2012.

[15] R. Chen, A. Reznichenko, P. Francis, and J. Gehrke. Towards Statistical Queries over Distributed Private User Data. In *NSDI'12*, pages 13–13, 2012.

[16] B. Chor, S. Goldwasser, S. Micali, and B. Awerbuch. Verifiable Secret Sharing and Achieving Simultaneity in the Presence of Faults. In *Proc. 26th IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 383–395, 1985.

[17] R. Cramer, I. Damgård, and Y. Ishai. Share Conversion, Pseudorandom Secret-Sharing and Applications to Secure Computation. In *TCC'05*, pages 342–362, 2005.

[18] G. Danezis, M. Kohlweiss, and A. Rial. Differentially Private Billing with Rebates. In *IH'11*, pages 148–162, 2011.

[19] L. Devroye. Non-Uniform Random Variate Generation, 1986.

[20] C. Dwork. Differential Privacy. In *ICALP'06*, pages 1–12, 2006.

[21] C. Dwork, K. Kenthapadi, F. McSherry, I. Mironov, and M. Naor. Our Data, Ourselves: Privacy Via Distributed Noise Generation. In *EUROCRYPT'06*, pages 486–503, 2006.

[22] C. Dwork, F. McSherry, K. Nissim, and A. Smith. Calibrating Noise to Sensitivity in Private Data Analysis. In *TCC'06*, pages 265–284, 2006.

[23] F. Eigner and M. Maffei. Differential Privacy by Typing in Security Protocols. In *CSF'13*, 2013.

[24] T. Elahi, G. Danezis, and I. Goldberg. PrivEx: Private Collection of Traffic Statistics for Anonymous Communication Networks. In *CCS'14*, 2014. to appear.

[25] S. L. From and T. Jakobsen. Secure Multi-Party Computation on Integers. Master's thesis, University of Aarhus, 2006.

[26] I. Gazeau, D. Miller, and C. Palamidessi. Preserving differential privacy under finite-precision semantics. In *QAPL'13*, pages 1–18, 2013.

[27] R. Gennaro, M. O. Rabin, and T. Rabin. Simplified VSS and Fact-Track Multiparty Computations with Applications to Threshold Cryptography. In *PODC'98*, pages 101–111, 1998.

[28] A. Ghosh, T. Roughgarden, and M. Sundararajan. Universally Utility-Maximizing Privacy Mechanisms. In *STOC'09*, pages 351–360, 2009.

[29] S. Goryczka, L. Xiong, and V. Sunderam. Secure Multiparty Aggregation with Differential Privacy: A Comparative Study. In *EDBT/ICDT'13*, pages 155–163, 2013.

[30] S. Inusah and T. J. Kozubowski. A Discrete Analogue of the Laplace Distribution. *JSPI*, 136(3):1090–1102, 2006.

[31] Y. Ishai, M. Prabhakaran, and A. Sahai. Secure Arithmetic Computation with No Honest Majority. In *TCC'09*, pages 294–314, 2009.

[32] M. Jawurek and F. Kerschbaum. Fault-Tolerant Privacy-Preserving Statistics. In *PETS'12*, pages 221–238, 2012.

[33] F. McSherry and K. Talwar. Mechanism Design via Differential Privacy. In *FOCS'07*, pages 94–103, 2007.

[34] I. Mironov. On Significance of the Least Significant Bits for Differential Privacy. In *CCS'12*, pages 650–661, 2012.

[35] I. Mironov, O. Pandey, O. Reingold, and S. P. Vadhan. Computational Differential Privacy. In *Crypto'09*, pages 126–142, 2009.

[36] A. Molina-Markham, P. Shenoy, K. Fu, E. Cecchet, and D. Irwin. Private Memoirs of a Smart Meter. In *BuildSys'10*, pages 61–66, 2010.

[37] T. P. Pedersen. Non-Interactive and Information-Theoretic Secure Verifiable Secret Sharing. In *Crypto'91*, pages 129–140, 1991.

[38] V. Rastogi and S. Nath. Differentially Private Aggregation of Distributed Time-Series with Transformation and Encryption. In *SIGMOD'10*, pages 735–746, 2010.

[39] E. Shi, T.-H. H. Chan, E. G. Rieffel, R. Chow, and D. Song. Privacy-Preserving Aggregation of Time-Series Data. In *NDSS'11*, 2011.

[40] 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 *CCS'09*, pages 534–544, 2009.

[41] A. C.-C. Yao. Protocols for Secure Computations (Extended Abstract). In *FOCS'82*, pages 160–164, 1982.

[42] C.-H. Yu, S. S. Chow, K.-M. Chung, and F.-H. Liu. Efficient Secure Two-Party Exponentiation. In *CT-RSA'11*, pages 17–32. 2011.

On the Privacy Provisions of Bloom Filters in Lightweight Bitcoin Clients

Arthur Gervais[†], Ghassan O. Karame[‡], Damian Gruber[⫰], Srdjan Capkun[†]
[†⫰]ETH Zurich, Switzerland, [‡]NEC Laboratories Europe, Germany

[†]firstname.lastname@inf.ethz.ch, [‡]firstname.lastname@neclab.eu, [⫰]gruberda@student.ethz.ch

ABSTRACT

Lightweight Bitcoin clients are gaining increasing adoption among Bitcoin users, owing to their reduced resource and bandwidth consumption. These clients support a simplified payment verification (SPV) mode as they are only required to download and verify a part of the block chain—thus supporting the usage of Bitcoin on constrained devices, such as smartphones. SPV clients rely on Bloom filters to receive transactions that are relevant to their local wallet. These filters embed all the Bitcoin addresses used by the SPV clients, and are outsourced to more powerful Bitcoin nodes which then only forward to those clients transactions relevant to their outsourced Bloom filters.

In this paper, we explore the privacy of existing SPV clients. We show analytically and empirically that the reliance on Bloom filters within existing SPV clients leaks considerable information about the addresses of Bitcoin users. Our results show that an SPV client who uses a modest number of Bitcoin addresses (e.g., < 20) risks revealing almost all of his addresses. We also show that this information leakage is further exacerbated when users restart their SPV clients and/or when the adversary has access to more than one Bloom filter pertaining to the same SPV client. Motivated by these findings, we propose an efficient countermeasure to enhance the privacy of users which rely on SPV clients; our proposal can be directly integrated within existing SPV client implementations.

1. INTRODUCTION

Bitcoin has already witnessed a wider adoption and attention than any other digital currency proposed to date. Bitcoin implements a distributed time-stamping service [25], which operates on top of the Bitcoin Peer-to-Peer (P2P) network and ensures that all transactions and their order of execution are visible to all Bitcoin users.

Currently, a typical Bitcoin installation requires more than 18 GB of disk space, and requires considerable time to download and locally index blocks and transactions that are contained in the block chain. In addition to disk space usage, the continuous growth of Bitcoin's transactional volume [1] incurs considerable overhead on the Bitcoin clients that need to verify the correctness of broadcasted blocks and transactions in the network. This problem becomes even more evident when users wish to perform/verify Bitcoin payments using resource-constraint devices, such as mobile devices.

To remedy this, the Bitcoin developers released a lightweight client, BitcoinJ [2], which supports a simplified payment verification (SPV) mode where only a small part of the block chain is downloaded— thus supporting the typical usage of Bitcoin on constrained devices (e.g., smartphones, cheap virtual private servers). SPV clients were originally proposed by Nakamoto in [25] and were later extended to rely on Bloom filters [17] in order to receive transactions that are relevant to their local wallet. These Bloom filters embed all the addresses used by the SPV clients, and are outsourced to more powerful Bitcoin nodes; these nodes will then forward to the SPV clients those transactions relevant to their Bloom filters.

Bloom filters can be defined with a target false positive rate; by appropriately setting the target false positive rate of Bloom filters, Bitcoin developers aim to provide a suitable anonymity set to hide the addresses of SPV clients. As far as we are aware, the information leakage associated with the reliance on Bloom filters has not been yet thoroughly analyzed in the context of Bitcoin [2].

In this paper, we address this problem, and explore the privacy provisions due to the use of Bloom filters in existing SPV client implementations. We show analytically and empirically that the current integration of Bloom filters within Bitcoin leaks considerable information about the addresses of Bitcoin users. More specifically, we show that the information leakage due to Bloom filters significantly depends on the number of addresses that each user possesses; notably, users who have a modest number of addresses (< 20) risk leaking all of their addresses by embedding them in a Bloom filter. This information leakage is further exacerbated when nodes restart their client, or generate additional Bitcoin addresses to their SPV clients; in these cases, the SPV clients re-compute new Bloom filters. We show that the computation of new Bloom filters considerably reduces the privacy of SPV clients.

Our work therefore motivates a careful assessment of the current implementation of SPV clients, prior to any large-scale deployment. In this work, we make the following contributions:

- We show that considerable information about users who possess a modest number of Bitcoin addresses (e.g., < 20) is leaked by a single Bloom filter in existing SPV clients.
- We show that an adversary can easily link different Bloom filters which embed the same elements—irrespective of the target false positive rate. This also enables the adversary to link, with high confidence, different Bloom filters which pertain to the same originator.
- We show that a considerable number of the addresses of users are leaked if the adversary can collect at least two Bloom fil-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ACSAC'14, December 08 - 12 2014, New Orleans, LA, USA.
Copyright 2014 ACM 978-1-4503-3005-3/14/12 ...$15.00.
http://dx.doi.org/10.1145/2664243.2664267 .

ters issued by the same SPV client—irrespective of the target false positive rate and of the number of user addresses.

- Finally, we propose a lightweight and efficient countermeasure to enhance the privacy offered by SPV clients. Our countermeasure can be integrated with minimum modifications within existing SPV client implementations.

The remainder of this paper is organized as follows. In Section 2, we briefly overview the operation of SPV clients. In Section 3, we introduce our system and attacker model. In Section 4, we analyze the privacy provisions of existing SPV client implementations when the adversary captures a single Bloom filter of an SPV client. In Section 5, we discuss the information leakage when the adversary can acquire multiple Bloom filters of an SPV node. In Section 6, we propose an efficient solution to enhance the privacy of users in SPV clients. In Section 7, we review related work in the area and we conclude the paper in Section 8.

2. BACKGROUND

Bitcoin enables its users to perform payments by issuing transactions to Bitcoin addresses. A Bitcoin address corresponds to a public key, whose corresponding secret key enables the address owner to spend the BTCs stored in the respective address. In order to spend BTCs, a user first creates a transaction, which typically takes as inputs the outputs of earlier transactions addressed to his addresses, and specifies as outputs the Bitcoin addresses (or their corresponding public keys) which will collect the resulting BTCs. Finally, the user signs the transaction and broadcasts it into the Bitcoin peer-to-peer (P2P) network.

Transactions are included and stored into Bitcoin blocks. Blocks are generated (or mined) by solving a hash-based proof-of-work (PoW) scheme. Bitcoin requires all peers in the system to verify all broadcasted transactions and blocks. Clearly, this comes at the expense of storage and computational overhead. Currently, a typical Bitcoin installation requires more than 18 GB of disk space, and considerable time to download and locally index blocks and transactions that are contained in the block chain. Additionally, the continuous growth of Bitcoin transactional volume incurs significant computational overhead on the Bitcoin clients, when verifying the correctness of broadcasted blocks and transactions in the network. This problem becomes even more evident when users wish to perform/verify Bitcoin payments from resource-constrained devices such as mobile devices, tablets, etc.

To remedy that, lightweight client implementations (or simplified payment verification, SPV) have been proposed in [25]; SPV clients do not store the entire block chain, nor do they validate all transactions in the system. Notably, SPV clients only perform a limited amount of verifications, such as the verification of block difficulty and the presence of a transaction in the Merkle tree, etc., and offload the verification of all transactions and blocks to the full Bitcoin nodes. Note that in order to calculate their own balance, SPV clients request full blocks from a given block height on; here, the full Bitcoin nodes can also provide "filtered blocks" to the SPV client that only contain relevant transactions from each block.

Unlike full Bitcoin nodes, SPV clients do not receive all the transactions that are broadcasted within the Bitcoin P2P network, but instead receive a subset of transactions, filtered for them by the full nodes to which they are connected[1]. To reduce bandwidth consumption, SPV clients make use of Bloom filters [17]. A Bloom filter [12] is a space-efficient probabilistic data structure which is used to test membership of an element. An SPV client constructs

[1]Currently, SPV clients connect to a default of four different randomly chosen nodes

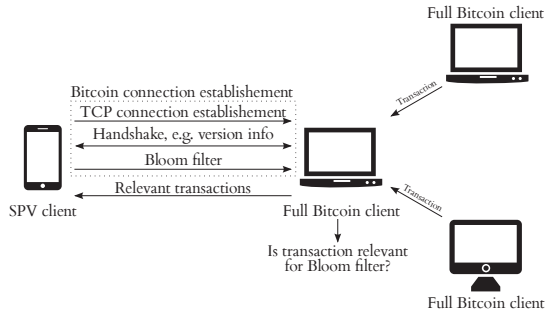


Figure 1: Sketch of the operation undergone by an SPV client. SPV clients connect to a full Bitcoin node, which only forwards to the SPV clients the transactions relevant to their Bloom filters.

a Bloom filter by embedding all the Bitcoin addresses which appear in its wallets. Upon connection to a full Bitcoin node, the constructed Bloom filter is outsourced to the full node following an initial handshake protocol (cf. Figure 1). Whenever the full node receives a transaction, it first checks to see if its input and/or output addresses match the SPV client's Bloom filter. If so, the full node forwards the received transaction to the SPV client.

Bloom filters have been first proposed by Bloom in 1970; we refer the readers to [13] for detailed information on Bloom filters. In SPV clients [2], a Bloom filter B of an SPV client is specified by the maximum number of elements that it can fit, denoted by M, without exceeding its target false-positive rate P_t. Let $m \leq M$ denote the number of elements that are inserted in $B(M, P_t)$. In SPV clients, the size of the filter n is computed as follows:

$$n = -\frac{M \ln(P_t)}{(\ln(2))^2} \tag{1}$$

A Bloom filter B basically consists of an array $B[.]$ of n bits accessed by k independent hash functions $H_1(.), \ldots, H_k(.)$, each of which maps an input string $x \in \{0,1\}^*$ to one of the n bits of the array. In SPV clients, k is computed as follows:

$$k = \ln(2)\frac{n}{M} \tag{2}$$

To insert an element $x \in \{0,1\}^*$ to a Bloom filter B, then $\forall j \in \{1, \ldots, k\}, B[H_j(x)] \leftarrow 1$. Similarly, to query the presence of an element $x \in \{0,1\}^*$ in B, then this entails computing $\wedge_{j=1}^{k} B[H_j(x)]$ (thus returning 1, if all corresponding bits are 1).

Bloom filters can generate a number of false positives, but cannot result in false negatives. The literature features a number of techniques to estimate the false positive rate of Bloom filters [13, 21]. In this paper, we focus on the proposal due to Mullin *et al.* [21] to estimate the false positive rates in Bloom filters (which is computed over all possible inputs to the Bloom filter). More specifically, we compute the false positive rate of a filter $B(M, P_t)$ which has m elements, $P_f(m)$, as follows [13]:

$$P_f(m) = \left(1 - \left(1 - \frac{1}{n}\right)^{km}\right)^k \tag{3}$$

We acknowledge that there might be more accurate techniques for computing the false positive rates (e.g., [13]); our findings, however, show that the difference in false positives resulting from [13] and [21] was negligible and did not affect our results. We therefore elected to rely on the estimation of P_f which appears in [21].

Here, note that $P_f(M) \approx P_t$. That is, the target false positive rate of a Bloom filter is only reached when the number of elements contained in the filter matches M.

3. MODEL

We assume that lightweight SPV clients connect to the Bitcoin P2P network through full Bitcoin nodes. As described earlier, full nodes only inform the SPV clients about transactions specific to their corresponding Bloom filters.

We assume that the adversary can compromise one or more full Bitcoin nodes and eavesdrop on communication links in order to acquire one or more Bloom filters pertaining to an SPV client. Here, the goal of the adversary is to identify the Bitcoin addresses that are inserted within a Bloom filter created by a particular SPV client. The addresses inserted in the Bloom filter typically correspond to addresses that the SPV client is interested in receiving information about (e.g., these addresses typically belong to the wallet of the SPV client). For example, the adversary might be connected to the node which generated the Bloom filter or might try to assign an identity to nodes according to their addresses, etc. Note that since the Bitcoin network provides currently less than 10,000 reachable full Bitcoin nodes, it is likely that regular nodes receive one or more filter pertaining to each SPV client over a sufficiently long period of time.

In our analysis, we assume that the adversary knows the parameters used to create a Bloom filter. This includes, for instance, the target false positive rate, P_t, that the Bloom filter is designed to achieve, and the number of hash functions k used in the Bloom filter. Since Bitcoin is an open payment system, we also assume that the adversary has access to all addresses/transactions which appear in the block chain, and to their respective order of execution.

Clearly, we assume that the adversary is computationally bounded. However, since the Bitcoin network only contains a bounded number of addresses (around 33 million Bitcoin addresses), the adversary can simply check whether all existing addresses match to a given Bloom filter.

Note that an adversary who is connected to an SPV client can see the transactions issued by the client and could potentially use this in order to learn the clients' addresses. This can be countered e.g., by SPV clients using TOR whenever they issue Bitcoin transactions. Instead, in this work, we focus on analyzing the privacy issues of the use of Bloom filters within existing SPV client implementations, which as we show, cannot be solved by simply relying on anonymizing networks.

Let $\mathbb{B}$ refer to the set of all existing Bitcoin addresses. Furthermore, let $\mathcal{B}_i$ refer to the set of all elements y in $\mathbb{B}$ that are members of Bloom filter B_i (i.e., for which a query in B_i returns true), and $\mathcal{F}_i \subset \mathcal{B}_i$ denote the set of false positives of B_i.

In our analysis, we assume that all elements map uniformly at random to the bits of the Bloom filter; that is, we assume that all addresses $y \in \mathcal{B}_i$ are equally likely to be a true member of B_i.

We further assume that the adversary can collect additional information from the system which can help her classify (a fraction of) the false positives exhibited by Bloom filters; for example, the adversary can try to identify some false positives generated by filters by analyzing two different Bloom filters which embed the same true positives (cf. Section 5). We capture this additional knowledge using the set $\mathbb{K} \subseteq \mathcal{F}_i$ which contains addresses in $\mathcal{F}_i$ that the adversary can correctly classify. We acknowledge that the adversary could also try to gather prior knowledge about the addresses inserted in the Bloom filter; in this work, we focus however on the case where the adversary does not have any prior knowledge about the true positives of the filter.

Our analysis throughout the rest of the paper does not exploit the fact that the adversary knows the public keys of Bitcoin addresses. Indeed, in current implementations of SPV clients, both the addresses and their public keys are inserted in the outsourced Bloom filter. As such, if the adversary knows both the address and its public key, then she can trivially test whether an address is a true positive of the filter by checking whether both the address and its public key are inserted within the filter. If not, then it is highly likely that the address is a false positive of the filter. We believe that the inclusion of both the address and its public key in the Bloom filter is a severe flaw in the current SPV client implementations—and can be easily countered; we thus do not exploit this flaw in our analysis. In fact, more than 99% of all Bitcoin transactions consist of payments to Bitcoin addresses (or the public key hash); moreover, only 4587 out of 33 million total addresses in the system received transactions destined for *both* their public keys and their public key hashes[2]. This means that for the vast majority of Bitcoin clients, there is no need to include both the public keys and their hashes (i.e., the Bitcoin addresses) in the Bloom filters; inserting one or the other would suffice (in more than 99% of the cases)[3].

Privacy Metric.

We quantify the privacy offered by a Bloom filter using the probability, $P_{h_{(j)}}$, that the adversary correctly guesses any j true positives of a Bloom filter among all positives that match the filter and which are not included in the knowledge of the adversary. More specifically, we measure $P_{h_{(j)}}$ achieved by a Bloom filter B_i, as follows: $P_{h_{(j)}} = \prod_{k=0}^{j-1} \frac{N-k}{N+S-k} = \frac{N}{N+S} \cdot \frac{N-1}{N+S-1} \cdots$. Here, N refers to the number of Bitcoin addresses inserted into B_i, and S denotes the cardinality of the set $\{\mathcal{F}_i - \mathbb{K}\}$; S therefore corresponds to all false positives that match B_i, but for which the adversary does not have any knowledge about. Therefore, the probability that the adversary correctly guesses *all* the addresses of B_i is given by: $P_{h_{(N)}} = \prod_{k=0}^{N-1} \frac{N-k}{N+S-k} = \frac{N!S!}{(N+S)!}$. Clearly, the higher is $P_{h_{(.)}}$, the smaller is the privacy of the SPV node. Note that if the adversary is able to identify an SPV client (e.g., by some side channel information), then simply identifying any address pertaining to that client would be a considerable violation of its privacy. Otherwise, if the adversary can link a number of addresses to the same anonymous client, then the information offered by the clustering of these addresses offers the adversary considerable information about the profile of the client, such as its purchasing habits, etc.

Note that our privacy metric only assesses the probability of guessing addresses inserted within the Bloom filter. As mentioned earlier, these are the addresses that the SPV client is interested in receiving information about. $P_{h_{(.)}}$, however, does not necessarily capture the probability of guessing addresses which belong to the user of the SPV client. Indeed, addresses inserted within the filter do not necessarily have to belong to the user; for instance, the privacy of users—who only embed in their Bloom filters $L \leq N$ addresses—can be quantified by computing $\frac{L}{N} P_{h_{(.)}}$.

Table 1 summarizes the notation used throughout the paper.

4. INFORMATION LEAKAGE DUE TO A SINGLE BLOOM FILTER

In this section, we start by analyzing the privacy provisions of existing SPV clients when the adversary acquires a single Bloom filter pertaining to an SPV client. In Section 5, we address the case when multiple Bloom filters are acquired by the adversary.

[2]We obtained these numbers by parsing the Bitcoin block chain till block # 296000.

[3]Note that only inserting the addresses in the Bloom filter would suffice since regular nodes can easily hash the public keys and check whether they match the Bloom filter. However, this clearly incurs computational overhead on regular Bitcoin nodes.

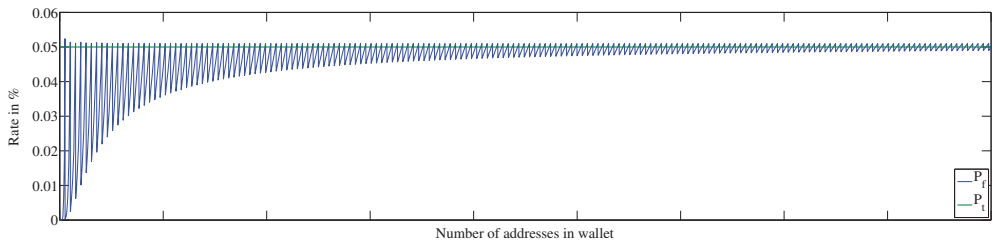


Figure 2: P_f, and P_t computed analytically with respect to the number of addresses N. Here, we assume that the SPV client did not restart since initialization.

$\mathbb{B}$	All existing Bitcoin addresses
B_i	Bloom filter i
$\mathcal{B}_i$	Positives of the Bloom filter
$\mathcal{F}_i$	False positives of the Bloom filter
N	Number of addresses inserted in the Bloom filter
S	False positives, the adversary has no knowledge about
M	Maximum number of elements the Bloom filter fits
m	Number of elements inserted in the Bloom filter
n	Size of the Bloom filter (in bits)
k	Number of hash functions of the Bloom filter
P_f	Actual false positive rate
P_t	Target false positive rate
$P_{h_{(j)}}$	Probability of correctly guessing any j true positives
X	Number of bits in the Bloom filter set to one

Table 1: Notations used throughout the paper.

4.1 Leakage under a Single Bloom Filter

In existing SPV clients (which use the Bitcoinj library), a node initializes its Bloom filter B_i with a random nonce r, and specifies its target false positive rate P_t which can be achieved when a number of elements M have been inserted in the filter. M is equal to $M = m + 100 = 2N + 100$, where N is the number of Bitcoin addresses inserted in B_i.[4] Note that a Bitcoin address is inserted into the Bloom filter by adding both the corresponding public key and the public key hash to the filter; therefore $m = 2N$.

By default, the target false positive rate of the Bloom filter P_t is set to 0.05%[5]. The size of the filter n and the number of hashes k are computed given Equations 1 and 2, respectively.

Note that at initialization time, the SPV client is only equipped with $j = 1$ Bitcoin address. The corresponding Bloom filter will then be initially created to fit $M = 102$ elements, and will only contain $m = 2$ elements. However, if the SPV client restarts (e.g., mobile phone reboots, mobile application is restarted), then the Bloom filter will be re-computed with $M = 2j + 100$ (the SPV client stores the Bloom filter in volatile memory). In either cases, when the user acquires 50 or more additional addresses such that $m > M$, then the SPV client will resize the Bloom filter by re-computing $M = 2N + 100$, and will send the updated Bloom filter to the full Bitcoin nodes that it is connected to. This process re-iterates whenever the number of addresses inserted in B_i increase such that $m > M$.

In Figure 2, we analytically compute P_t with respect to the number of addresses N that an SPV client is equipped with. Here, we assume that the adversary has access to only one Bloom filter pertaining to the SPV client, and that the adversary has no prior knowledge about addresses in Bitcoin (i.e., $\mathbb{K} = \phi$). Note that given n and k, the number of elements contained in a Bloom filter can be estimated by the adversary as follows [26]:

$$m \approx -n\frac{ln(1-\frac{X}{n})}{k} \tag{4}$$

Here, X corresponds to the number of bits of the Bloom filter set to one. Given n and P_t, M can also be computed by the adversary from Equation 1.

Since $\mathbb{K} = \phi$, $S + N = |\mathcal{B}_i|$ (i.e., the number of all existing Bitcoin addresses which match B_i)[6]. Note that, in April 2014, the Bitcoin block chain comprised nearly $|\mathbb{B}|$ =33 million addresses. This means that an adversary can simply try all possible addresses in the Bitcoin system in order to compute $\mathcal{B}_i$. In doing so, it is straightforward to see that:

$$P_{h_{(j)}} = \prod_{k=0}^{j-1} \frac{N-k}{N+S-k} \approx \prod_{k=0}^{j-1} \frac{N-k}{N+|\mathbb{B}-N|P_f(2N)-k}, \tag{5}$$

where $N \ll |\mathbb{B}|$, and $m = 2N$ is the number of elements contained in the Bloom filter seen by the adversary. In analyzing our findings, we distinguish two cases:

1) $2N/M \leq 0.4$ *and* $N < 100$.

In Figure 2, we compute P_t and P_f with respect to the number of addresses N. Our results show that $P_{h_{(1)}}$ (the probability of correctly guessing one address as a true positive) is large when $2N/M \leq 0.4$, as long as $N < 100$. Given a modest number of addresses in the Bloom filter (i.e, $N < 100$), $P_f(m = 2N)$ is small when $\frac{m}{M}$ is small. As $\frac{m}{M}$ increases, $P_f(m = 2N)$ increases (and $P_{h_{(1)}}$ decreases). As shown in Figure 2, $P_f(m = 2N)$ is less affected by $\frac{m}{M}$ for larger M. For example, when $N = 10$, $P_{h_{(10)}}$= 0.99 which corresponds to the probability of correctly guessing all the true positives when the SPV client has 10 addresses.

This means that the information leakage in SPV clients is considerable for new Bitcoin users, and for Bitcoin users which restart

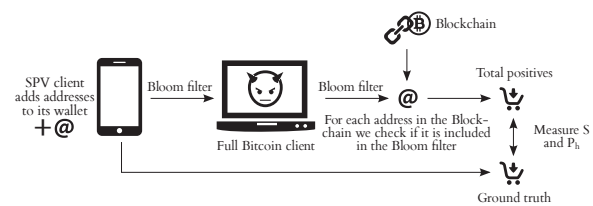


Figure 3: Experimental setup. We constructed around 18,060 different Bloom filters of various sizes and pertaining to 10 different wallets.

[4]The additional number '100' is added to m by the Bitcoin developers in order to avoid the recomputation of a new filter in the case where a user inserts up to 50 additional Bitcoin addresses.

[5]The Bitcoin developers claim that a target false positive rate of 0.1% should provide "very good privacy" [7].

[6]$|X|$ denotes the cardinality of set X.

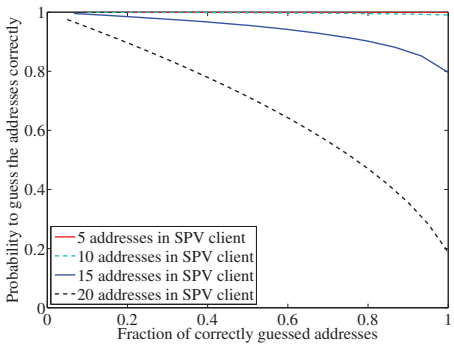


Figure 4: $P_{h_{(j)}}$ with respect to the number of addresses inserted in the Bloom filter. Given an SPV client with 5 addresses, all addresses can be guessed; when the SPV client has 20 addresses, 20% of the addresses can be guessed with almost 0.90 probability. Here, we assume that the user restarts its SPV client.

their SPV clients. For instance, at initialization time, the Bloom filter of SPV clients is typically instantiated using $M = 102$. Our results show that if the user is new in the Bitcoin system and only has 1 Bitcoin address, $P_{h_{(1)}} \approx 1$, which signals complete lack of privacy. Recall that this observation also holds when the SPV client restarts and $N < 100$; for instance, when $N = 20$, $m = 40$, $P_{h_{(1)}} \approx \frac{N}{N+|\mathbb{B}|P_f(m)} \approx 0.98$ and $P_{h_{(20)}} \approx 0.20$ (cf. figure 4). We validate our analysis experimentally in Section 4.2.

2) $2N/M > 0.4$.

Conforming with our previous analysis, when $2N/M > 0.4$, $P_f(m = 2N)$ is close to P_t. Recall that in this case, the probability of correctly guessing one address reaches a local minimum when $N = \frac{M}{2}$ (cf. Figure 2). Our results also show that the global minimum achieved by $P_{h_{(1)}}$ is 0.002961 and is reached when the user has $N = 51$ addresses.

In addition, we analytically compute $P_{h_{(j)}}$ when the SPV client has 5, 10, 15 and 20 addresses. Our results in Figure 4 show that guessing all addresses given one filter which embeds less than 15 addresses can be achieved with almost 0.80 probability. This probability decreases as the number of addresses embedded within the filter increases beyond 15.

4.2 Experimental Evaluation

We now proceed to validate our analytical results empirically, by means of implementation. For that purpose, we parsed the block chain from the genesis block mined in 2009 until the beginning of April 2014 using the parser in [5] and collected nearly 33 million distinct addresses. In our evaluation, we rely on *Bitcoin Wallet* [6], which builds upon the standard Bitcoinj library (0.12-SNAPSHOT of end of April 2014).

As mentioned earlier, Bitcoinj initializes by constructing one address by default. The user can subsequently add addresses to his wallet. Our implementation setup is depicted in Figure 3. Here, we construct 10 different Bitcoin wallets, and we gradually increase the number of Bitcoin addresses which populate each wallet from 1 address to 9,000 Bitcoin addresses. Whenever new addresses are added to the wallets, we compute the modified Bloom filters; we increase the number of addresses by a step of 2 addresses to reach 19 addresses, and then by a step of 5 addresses to reach 8,999 addresses. As such, our experiments resulted in the evaluation of a total of 18,060 different Bloom filters which contain various number of elements, and pertain to 10 different SPV clients. For each Bloom filter, we compute the matching set of Bitcoin addresses, we compare this set with the actual addresses inserted in each Bloom filter in order to compute S, and $P_{h_{(.)}}$. Since we assume here that the adversary has no a priori knowledge about Bitcoin addresses, S corresponds to the number of existing Bitcoin addresses (among the 33,000,000 total Bitcoin addresses) that match the Bloom filter of the SPV client, and that are not the addresses of the SPV client (i.e., S corresponds to the false positives generated by the Bloom filter).

Each data point in our experiments corresponds to the average of 10 independent measurements obtained from each of the 10 wallets. Our results (cf. Table 2) confirm our analysis in Section 4.1. More specifically, our results show that given few addresses (small N), $P_{h_{(.)}}$ is large; for example, when $N = 19$, B_i results in an average of 6.1 false positives among all 33,000,000 addresses which corresponds to $P_{h_{(1)}} = 0.76$ and $P_{h_{(N/2)}} = 0.0433$. As N increases to 49 addresses, $P_{h_{(1)}}$ and $P_{h_{(N/2)}}$ decrease to 0.004 and 0, respectively. Indeed, when $N = 49$, $N/M \approx 0.5$, since initially B_i is computed with $M = 102$. Here, B_i incurs $P_f(98) \approx P_t$ false positive rates. When N increases beyond 51, then $m > M$, and the Bloom filter is resized to fit $M = 2N + 100$ elements. In this case, when $2N/M$ is small, $P_{h_{(.)}}$ is large. For instance, when $N = 54$, $P_{h_{(1)}}$ is 0.36, which corresponds to a total of 96 false positives seen by the adversary among all addresses in $\mathbb{B}$. This process re-iterates whenever B_i is resized, however as N increases, we observe that the fluctuations in $P_{h_{(.)}}$ decrease as $P_f(2N)$ converges towards P_t.

In Figure 5, we also measure $P_{h_{(1)}}$ and the number of false positives when the SPV client restarts and has to re-compute its Bloom filter B_i. As mentioned earlier, our results show that the restarting of an SPV client causes $P_{h_{(1)}}$ to significantly increase when compared to the case where the client is not restarted. As mentioned earlier, at any restart and given N addresses, the SPV client recomputes B_i with a size of $M = 2N + 100$. Therefore, for modest values of N (e.g., $N < 100$), $2N/M$ becomes small, thus resulting in modest $P_f(2N)$ and a large $P_{h_{(1)}}$. Our findings also show that as N increases above 500 addresses, $P_{h_{(1)}}$ also increases. This is the case since the number of false positives provided by B_i is bounded by $P_t|\mathbb{B}|$, and does not depend on N.

Summary.

- We show that the probability $P_{h_{(.)}}$ that the attacker correctly guesses the addresses of the SPV client is large when the number N of addresses inserted in the filter is smaller than 20. For example, if the SPV client has 15 addresses in its wallet and restarts, correctly guessing all of its address can be achieved with a probability of at least $P_{h_{(15)}} = 0.8$.
- Due to addition of the constant '100' to M (the maximum number of elements the Bloom filter is designed for), the closer m (the actual number of elements in the filter) is to M, the higher is the privacy offered by the Bloom filter.

5. INFORMATION LEAKAGE DUE TO MULTIPLE BLOOM FILTERS

In Section 4.1, we analyzed the information leakage in the presence of an adversary who has access to a single Bloom filter instance of an SPV client. In this section, we extend our analysis to cover a more powerful adversary who can acquire multiple Bloom filters pertaining to SPV clients.

5.1 Leakage under Multiple Bloom Filters

In what follows, we assume that the adversary can acquire $b > 1$ Bloom filters pertaining to different users. For example, the adver-

N / P_t	$P_{h_{(1)}}$ (0.05%)	$P_{h_{(1)}}$ (0.1%)	$P_{h_{(1)}}$ (0.5%)	$P_{h_{(\lceil N/2 \rceil)}}$ (0.05%)	$P_{h_{(\lceil N/2 \rceil)}}$ (0.1%)	$P_{h_{(\lceil N/2 \rceil)}}$ (0.5%)	$P_{h_{(N)}}$ (0.05%)	$P_{h_{(N)}}$ (0.1%)	$P_{h_{(N)}}$ (0.5%)
1	1(±0)	1(±0)	1(±0)	-	-	-	1	1	1
3	1(±0)	1(±0)	1(±0)	1	1	1	1	1	1
19	0.76(±0.03)	0.42(±0.03)	0.03(±0.002)	0.0433	0.000026	0	0	0	0
49	0.004(± 0.00032)	0.0021(±0.00019)	0.00035(±0.00002)	0	0	0	0	0	0
54	0.36(±0.02)	0.14(±0.0059)	0.01(±0.00089)	0	0	0	0	0	0
8,999	0.35(±0.002)	0.21(±0.00075)	0.05(±0.00032)	0	0	0	0	0	0

Table 2: $P_{h_{(.)}}$ with respect to P_t and N. Each data point is averaged over 10 independent runs; we also show the corresponding 95% confidence intervals.

sary might be connected to SPV clients for a long period of time, and receive their updated Bloom filter. Alternatively, the adversary can acquire additional Bloom filters by compromising/colluding with other full Bitcoin nodes. Note that in our analysis, we do not assume that the adversary knows the correct association of the Bloom filters to the respective users.

Two Bloom Filters.

We start by analyzing the case where the adversary acquires two different Bloom filters B_1 and B_2. In the sequel, we focus on computing $P_{h_{(.)}}$ corresponding to filter B_1, which we assume to be the smallest of the two filters (in size). In analyzing the information leakage due to the acquisition of two Bloom filters, we distinguish two cases.

1) B_1 and B_2 pertain to different users.

Recall that each Bloom filter is initialized with a random seed, chosen uniformly at random from $\{0,1\}^{64}$. Therefore, if B_1 and B_2 pertain to different users, then it is highly likely that they are initialized with different random seeds. This means that the false positives generated by each filter are highly likely to correspond to different addresses. Moreover, since different users will have different Bitcoin addresses, B_1 and B_2 will contain different elements. Therefore, $\mathcal{B}_1 \cap \mathcal{B}_2$ is likely to be comprised of only few addresses, if any. Notably, when B_1 and B_2 pertain to different users, then $|\mathcal{B}_1 \cap \mathcal{B}_2|$ can be computed as follows:

$$E[|\mathcal{B}_1 \cap \mathcal{B}_2|] \approx (|\mathcal{B}_1| - N_1)|\mathcal{B}_2| \frac{1}{|\mathbb{B}| - N_1} \tag{6}$$

$$\approx \frac{P_f(m_1)P_f(m_2)|\mathbb{B}|^2}{|\mathbb{B}| - N_1} \tag{7}$$

$$\approx P_f(m_1)P_f(m_2)|\mathbb{B}|, \tag{8}$$

where N_1 corresponds to the number of elements inserted in B_1. $E[|\mathcal{B}_1 \cap \mathcal{B}_2|]$ is the expected number of elements which match $\mathcal{B}_2$ and $\mathcal{B}_1$. The number of elements in $\mathbb{B}$ which match $\mathcal{B}_2$ is given by $P_f(m_2)|\mathbb{B}|$. Then, $E[|\mathcal{B}_1 \cap \mathcal{B}_2|]$ can be computed by assuming a binomial distribution with success probability $P_f(m_2)$, and with $P_f(m_2)|\mathbb{B}|$ number of trials.

Note that the adversary can compute m_1 (using Equation 4); if $m_1 > |\mathcal{B}_1 \cap \mathcal{B}_2|$, then this offers a clear distinguisher for the adversary that the two acquired Bloom filters B_1 and B_2 pertain to different user wallets.

2) B_1 and B_2 pertain to the same user.

On the other hand, in the case where B_1 and B_2 correspond to the same SPV client, three sub-cases emerge:

B_1 and B_2 use the same size/seed: This is the case when users, e.g., create additional Bitcoin addresses and need to update their outsourced Bloom filters to include those addresses. In this case, $\mathcal{B}_1$ and $\mathcal{B}_2$ are likely to comprise of similar Bitcoin addresses. This includes both the actual elements of the filters, i.e., the Bitcoin addresses of the user, and the false positives generated by the Bloom filter. In this case, $|\mathcal{B}_1 \cap \mathcal{B}_2|$ can be computed as follows:

$$E[|\mathcal{B}_1 \cap \mathcal{B}_2|] \approx N_1 + P_f(2N_1)|\mathbb{B}| \tag{9}$$

$$P_{h_{(j)}} \approx \prod_{k=0}^{j-1} \frac{N_1 - k}{N_1 + P_f(2N_1)|\mathbb{B}| - k} \tag{10}$$

Notice that $|\mathcal{B}_1 \cap \mathcal{B}_2| = S_1 + N_1 = |\mathcal{B}_1|$ (since $\mathcal{B}_1 \subset \mathcal{B}_2$); therefore, $P_{h_{(j)}}$ is not affected by the acquisition of the second filter B_2.

B_1 and B_2 use different seeds: In existing SPV clients, the random nonce r used to instantiate the Bloom filter is stored in volatile memory. Therefore, each time the SPV client is restarted (e.g., smartphone reboots), a new filter will be created with a new seed chosen uniformly at random. If the adversary acquires two Bloom filters of the same user which are initialized with different seeds, then these filters are likely to exhibit different false positives. B_1 and B_2 will however comprise of a number of identical elements (which map to the Bitcoin addresses of the user). More specifically,

$$E[|\mathcal{B}_1 \cap \mathcal{B}_2|] \approx N_1 + (|\mathcal{B}_1| - N_1)\frac{|\mathcal{B}_2|}{|\mathbb{B}| - N_1} \tag{11}$$

$$\approx N_1 + P_f(m_1)P_f(m_2)|\mathbb{B}| \tag{12}$$

$$P_{h_{(j)}} \approx \prod_{k=0}^{j-1} \frac{N_1 - k}{N_1 + P_f(m_1)P_f(m_2)|\mathbb{B}| - k} \tag{13}$$

As we show in Section 5.2, the obtained $P_{h_{(j)}}$ is considerably large in this case, when compared to the case where the adversary has access to only one filter.

B_1 and B_2 use the same seed, but have different sizes: This is the case when users, e.g., create additional Bitcoin addresses

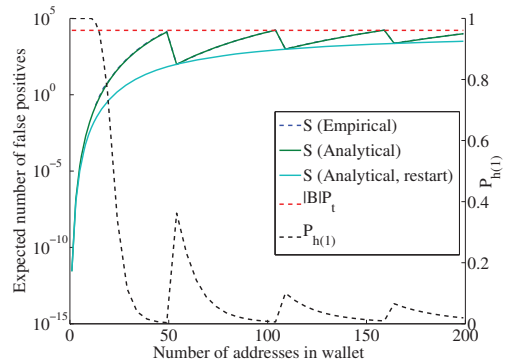


Figure 5: $P_{h_{(.)}}$ and S computed experimentally for the first 200 insertion of Bitcoin addresses into the Bloom filter (with and without restart of the SPV client).

beyond the capacity of their current Bloom filters. SPV clients therefore need to resize their Bloom filters. Note that filter resizing typically shuffles the bits of the Bloom filters; the resulting distribution of bits in the new resized filter is not necessarily pseudo-random (since the same seed is used) and depends on the sizes of the filters. As such, only the lower bound on $|\mathcal{B}_1 \cap \mathcal{B}_2|$ can be estimated using Equation 12 (which estimates the worst case where filter resizing causes a pseudo-random permutation of the bits of the filters); in Section 5.2, we confirm our analysis by means of experiments.

Recall that since there are only few tens of millions of addresses in Bitcoin, the adversary can brute-force search the entire list of Bitcoin addresses in order to acquire $\mathcal{B}_1$ and $\mathcal{B}_2$ and compute $\mathcal{B}_1 \cap \mathcal{B}_2$. Given any two Bloom filters B_1 and B_2, the adversary can easily guess whether these two Bloom filters contain Bitcoin addresses from the same wallet. Indeed, if $|\mathcal{B}_1 \cap \mathcal{B}_2|$ is small, then it is highly likely that B_1 and B_2 map to different elements (if m_1 and m_2 are not small), and therefore pertain to different users. On the other hand, when $|\mathcal{B}_1 \cap \mathcal{B}_2| \gg 0$, it is highly likely that all the Bitcoin addresses in the set $\mathcal{B}_1 \cap \mathcal{B}_2$ belong to the same SPV client.

Multiple Bloom Filters.

In the previous paragraphs, we discussed the case where the adversary is equipped with only two Bloom filters. Note that our analysis equally applies to the case where the adversary possesses any number $b > 2$ of Bloom filters pertaining to the same entity.

As mentioned earlier, by computing the intersection between each pair of filters, the adversary can find common elements to different filters; this also enables the adversary to guess with high confidence whether different filters have been generated by the same client. Given b filters which belong to the same SPV client, the adversary can compute the number of elements inserted within each filter using Equation 4. In the sequel, we assume that filters $B_1, \dots, B_b$ are sorted by increasing number of elements (i.e., B_b contains the largest number of elements), and that filters are constructed using different seeds. Let $\mathcal{K}_j = \mathcal{B}_j \cap \dots \cap \mathcal{B}_{(b-1)}, \forall j \in [1, b-1]$.

Note that $|\mathcal{K}_1| \leq |\mathcal{K}_2| \dots \leq |\mathcal{K}_{(b-1)}|$. Here, the larger the number of Bloom filters at the disposal of the adversary, the smaller is the error of the adversary in correctly classifying the genuine addresses of the SPV client, and the larger is $P_{h_{(.)}}$. That is, the larger is b, the smaller are the number of common false positives that are exhibited by the different filters, and the higher is the confidence of the adversary in identifying the false positives of B_j. Following Equation 12,

$$E[|\mathcal{K}_1|] = \min(|\mathcal{B}_1|, |\mathcal{B}_2|, \dots) \approx N_1 + |\mathbb{B}| \prod_{\forall j} P_f(m_j) \quad (14)$$

$$P_{h_{(j)}} \approx \prod_{k=0}^{j-1} \frac{N_i - k}{N_i - k + |\mathbb{B}| \prod_{\forall j} P_f(m_j)} \quad (15)$$

Moreover, as j increases, $\mathcal{K}_j$ will contain more false positives, and $P_{h_{(j)}}$ will decrease.

5.2 Experimental Evaluation

In what follows, we experimentally validate our analysis in Section 5.1. For that purpose, we rely on a similar setup to the one used in Section 4.2, and we perform four different experiments. We perform all four experiments by setting the target false positive rate P_t to 0.05%, 0.1% and 0.5%, respectively.

In our first experiment (Experiment 1), we create 10 different user wallets, each generating five different Bloom filter B_1,B_2,..., B_5 with the same size and using the same random seed r, but each having a different number of elements $N = \{25, 30, 35, 40, 45\}$. This corresponds to the case where 10 different users constantly insert new Bitcoin addresses and update their outsourced Bloom filters. Here, we assume that our adversary captures all 50 Bloom filters and applies our analysis described in Section 5.1 to learn additional information about the user addresses. In this experiment, we compute $I_1 = \mathcal{B}_1 \cap \mathcal{B}_5$ for each SPV client; we then report the average intersection size for all 10 wallets in Table 3.

Our results in Table 3 show that Bloom filters pertaining to the same SPV client, and which share the same initial seed, are likely to exhibit the same false positives (in addition to the elements inserted in the Bloom filter). Indeed, our results show that $|I_1|$—measured experimentally—matches Equation 9, irrespective of the target P_t; moreover, in this case, $P_{h_{(.)}}$ obtained using $b = 2$ Bloom filters (cf. Equation 13) is similar to $P_{h_{(.)}}$ when $b = 1$ one Bloom filter.

In our second experiment (Experiment 2), we extend our evaluation to account for the case where Bloom filters originating from the same user have different sizes. Here, we also create 10 different user wallets, each generating five different Bloom filters B_1,B_2,..., B_5 using the same random seed, but each having a different number of elements (respectively $N = \{70, 120, 170, 220, 270\}$) and different sizes ($M$ ranges between 3224 and 9680 bits). Analogously to Experiment 1, we compute $I_2 = \mathcal{B}_1 \cap \mathcal{B}_5$ for each SPV client; we then report the average intersection size for all 10 wallets in Table 3. Additionally, we compute the intersection set I_3 shared by B_1 from the first wallet, with a randomly chosen Bloom filter from the remaining 9 wallets. In Table 5, we report the average intersection set size over the 9 wallets. Our results in Tables 3 and 5 confirm our aforementioned analysis. Indeed, $|I_2|$ matches the values derived using Equation 12 when the Bloom filters pertain to the same user. Otherwise, $|I_3|$ matches Equation 6.

In our third experiment (Experiment 3), we investigate the case where Bloom filters pertaining to the same user are initialized with different random seeds. As mentioned earlier, this, e.g., corresponds to the case when the user restarts the SPV client. In this experiment, we create 10 different user wallets, each generating 16 different Bloom filters constructed using different initial seeds as follows: 4 filters B_1,B_2,B_3,B_4 with $N = 10$, $N = 100$, $N = 1,000$, and $N = 10,000$, respectively. For each wallet, and all filters of the same size, we compute $I_4 = \mathcal{B}_1 \cap \mathcal{B}_4$ and we report the average $|I_4|$ for each filter sizes in Table 3. Additionally, for each filter size, we also compute the intersection set I_5 using B_1 from the first wallet, with a randomly chosen Bloom filter from all other 9 wallets. In Table 5, we report the average intersection set size over the 9 wallets. Similar to Experiment 2, our results in Table 3 show that, irrespective of the filter size, and of the target false positive rate, $P_{h_{(.)}}$ significantly decreases when the adversary acquires $b = 2$ Bloom filters pertaining to the same SPV client.

Finally, in our final experiment (Experiment 4), we investigate the impact of having $b > 2$ Bloom filters pertaining to the same SPV client. For that purpose, we use 5 Bloom filters B_1,B_2,...,B_5 generated using different seeds with $N = \{3070, 3120, 3170, 3220, 3270\}$. We then compute $\mathcal{K}_j = \mathcal{B}_1 \cap \dots \cap \mathcal{B}_{(j+1)}, \forall j \in [1, b-1]$, and the corresponding $P_{h_{(.)}}$ using Equation 15. Our findings are depicted in Table 4; our results validate our aforementioned analysis and show that the larger the number b of acquired Bloom filters of the same SPV client, the larger is $P_{h_{(.)}}$, and the smaller is the privacy of the user's addresses.

Summary.

- In Experiments 1,2, and 3, notice that $|I_3| \approx |I_5| \ll \frac{\min(m1,m2)}{2}$. This means that given any two Bloom filters B_1 and B_2, if $|B_1 \cap B_2| \ll \frac{\min(m1,m2)}{2}$ (estimated using Equation 1), then

N_i, N_j	P_t (%)	$\lvert\mathcal{B}_i \cap \mathcal{B}_j\rvert$	$P_{h_{(1)}}$ $(b=2)$	$P_{h_{(1)}}$ $(b=1)$	$P_{h_{(N/2)}}$ $(b=2)$	$P_{h_{(N)}}$ $(b=2)$	N_i, N_j	P_t (%)	$\lvert\mathcal{B}_i \cap \mathcal{B}_j\rvert$	$P_{h_{(1)}}$ $(b=2)$	$P_{h_{(1)}}$ $(b=1)$	$P_{h_{(N/2)}}$ $(b=2)$	$P_{h_{(N)}}$ $(b=2)$
Experiment 1 (Same client, same seed, same size)							**Experiment 2 (Same client, same seed, different size)**						
25,45	0.05	83.6(±10.88)	0.2990	0.2910	0	0	70,270	0.05	70.3(±0.28)	0.9957	0.0678	0.8145	0.2502
25,45	0.1	245.0(±15.36)	0.1020	0.1070	0	0	70,270	0.1	71.70(±0.48)	0.9762	0.0266	0.3177	0.0011
25,45	0.5	3192.90(±230.79)	0.0078	0.0075	0	0	70,270	0.5	671.20(±46.54)	0.1043	0.0031	0	0
Experiment 3 (Same client, different seed)													
10, 10	0.05	10.0(±0.0)	1	0.9997	1	1	10, 10	0.5	10.0(±0.0)	1	0.7448	1 1	1
100, 100	0.05	100.0(±0.0)	1	0.1164	1	1	100, 100	0.5	110.90(±1.40)	0.9017	0.0052	0.0009	0
1,000, 1,000	0.05	1004.70(±1.54)	0.9953	0.0785	0.0390	0	1,000, 1,000	0.5	1499.70(±22.7)	0.6668	0.0077	0	0
5,000, 5,000	0.05	5007.30(±1.96)	0.9985	0.0003	0.0064	0	5,000, 5,000	0.5	5755(±15.05)	0.8688	0.0308	0	0
10, 10	0.1	10.0(±0.0)	1	0.9969	1	1	1,000, 1,000	0.1	1015.60(±2.37)	0.9846	0.0395	0	0
100, 100	0.1	100.30(±0.28)	0.9970	0.0464	0.8138	0.225	5,000, 5,000	0.1	5032.40(±3.15)	0.9936	0.1376	0	0

Table 3: Measuring $|\mathcal{B}_i \cap \mathcal{B}_j|$ and $P_{h_{(.)}}$ in Experiments 1,2, and 3, using filters of the same SPV client with respect to N and P_t. Here, $m_i \leq m_j$. Each data point is averaged over 10 independent runs; we also present the 95% confidence intervals.

b	$P_{h_{(1)}}$ $(P_t = 0.05\%)$		$P_{h_{(1)}}$ $(P_t = 0.1\%)$		$P_{h_{(N/2)}}$ $(P_t = 0.05\%)$		$P_{h_{(N/2)}}$ $(P_t = 0.1\%)$		$P_{h_{(N)}}$ $(P_t = 0.05\%)$		$P_{h_{(N)}}$ $(P_t = 0.1\%)$	
	(Analy.)	(Emp.)	(Analy.)	(Emp.)	(Analy.)	(Emp.)	(Analy.)	(Emp.)	(Analy.)	(Emp.)	(Analy.)	(Emp.)
1	0.1713	0.1715	0.0926	0.0916	0	0	0	0	0	0	0	0
2	0.9978	0.9977	0.9911	0.9908	0.0091	0.0079	0	0	0	0	0	0
3	1	1	1	1	1	1	1	1	1	1	1	1
4	1	1	1	1	1	1	1	1	1	1	1	1
5	1	1	1	1	1	1	1	1	1	1	1	1

Table 4: Experiment 4: $P_{h_{(.)}}$ w.r.t. the number b of Bloom filters which pertain to same SPV client. Here, we assume that each filter is generated using a different seed.

an adversary can easily tell whether any two Bloom filters pertain to the same SPV client.

- If the two Bloom filters acquired by the adversary belong to the same SPV client, the adversary can identify whether the SPV client has restarted while generating his Bloom filters; here, notice that $\frac{\min(m1,m2)}{2} \leq |I_4| \ll |I_2| \ll |I_1|$.
- $P_{h_{(.)}}$ corresponding to $b = 2$ filters is considerably larger when compared to the case where the adversary has access to one Bloom filter. This means that an adversary which can acquire more than one Bloom filter pertaining to an SPV client can learn considerable information about the addresses of the node—irrespective of the size of the Bloom filter and P_t. In this case, our results show that $P_{h_{(N)}}$ approaches 1, which signals full leakage of the addresses of the SPV client. $P_{h_{(.)}}$ increases to 1 as the number b of Bloom filters of the same SPV client captured by the adversary increases.

6. OUR PROPOSED SOLUTION

Given our findings, we propose in what follows a solution that enhances the privacy of SPV clients which rely on Bloom filters. Before presenting our solution, we briefly summarize our observations from Sections 4 and 5.

Observation 1: The number of elements inserted within a Bloom filter significantly affects the resulting false positive rate of the filter. This is especially true when the filter's size is modest (e.g., < 500). Indeed, the number of elements inserted in the filter should match at all times the filter's size in order to achieve the target false positive rate (i.e., $P_f(m) = P_t$).

Observation 2: The acquisition of multiple Bloom filters considerably reduces the privacy of SPV clients. The construction of multiple Bloom filters per SPV client should be avoided. Otherwise, different Bloom filters should be constructed with different initial seeds, and should contain different elements. In this way, an adversary does not gain considerable advantage when acquiring two or more Bloom filters.

N_i	P_t (%)	$E[\lvert\mathcal{B}_i \cap \mathcal{B}_j\rvert]$ (Analytical)	$\lvert\mathcal{B}_i \cap \mathcal{B}_j\rvert$ (Empirical)
Experiment 2 (Different client, different seed, different size)			
-	0.05	-	0.0(±0)
-	0.1	-	1.22(±0.60)
-	0.5	-	31.60(±6.34)
Experiment 3 (Different client, different seed, same size)			
10	0.05	0	0.0(±0.0)
10	0.1	0	0.0(±0.0)
10	0.5	0	0.0(±0.0)
100	0.05	0	0.0(±0.0)
100	0.1	0.13	0.0(±0.0)
100	0.5	11.27	13.89(±1.64)
1,000	0.05	4.21	5.0(±0.97)
1,000	0.1	18.06	21.11(±2.35)
1,000	0.5	523.20	512.89(±23.21)
5,000	0.05	7.37	11.89(±1.73)
5,000	0.1	30.01	40.11(±3.40)
5,000	0.5	753.41	789.78(±18.92)

Table 5: Measuring $|\mathcal{B}_1 \cap \mathcal{B}_2|$ in Experiments 1,2, and 3, using filters pertaining to different SPV clients with respect to P_t. Each data point is averaged over 10 independent runs.

Observation 3: SPV clients should keep state about their outsourced Bloom filters (i.e., on persistent storage) to avoid the need to re-compute a filter which contains the same elements using different parameters.

Observation 4: As mentioned earlier, inserting both the public key and the public key hash (the address) in the Bloom filter provides a sufficient distinguisher for the adversary in guessing whether an address is a true positive or not. Note that for the most common transaction type *Pubkey Hash* (P2PKH), inserting the hash of the public key is sufficient. However, there might be other transaction types where it is beneficial to also store the public key in the Bloom filter. In this case, the client can insert either a Bitcoin address or its corresponding public key (but not both) in the same Bloom filter. Indeed, sample experimental results that we conducted show that for almost 99% of all addresses in the network, it suffices to in-

sert either the public key or the public key hash within the same Bloom filter in order to receive all the relevant transactions destined to the address.

Our Solution.

In what follows, we describe a solution which leverages Observations 1, 2, 3, and 4. In our solution, each SPV client generates N Bitcoin addresses at the start, and embeds them in a Bloom filter which can fit $M = m = N$. Here, the Bloom filter is constructed with a realistic target false positive rate P_t, which combined with N and M, results in a target privacy level (cf. Equation 3). Note that since $M = m$, then we ensure that the Bloom filter's false positive rate matches P_t (cf. Section 4). Here, we assume that only the address is inserted within each filter.

Clearly, since the user might not directly use all N addresses, some of his Bitcoin addresses will not be revealed and will remain in his wallets. Whenever users run out of their N addresses and need to get additional addresses, they repeat the aforementioned setup process. That is, users create an additional set of N addresses and embed them in a new Bloom filter—constructed with a new initial seed— with $M = m = N$, and using the previously chosen P_t. Here, the advantage of an adversary which captures one or more Bloom filters pertaining to the same SPV client is negligible, since these filters do not have any element in common.

Additionally, our solution requires the SPV clients to keep state, e.g., about each Bloom filter, to avoid the need of re-computation of the same filter if the client restarts at any point in time. We point out that the required storage overhead to maintain information about the state of each Bloom filter is negligible. Indeed, to keep all the necessary state to reconstruct a Bloom filter, an SPV client needs to locally store: *(i)* the number of addresses embedded in the filer (4 bytes), *(ii)* the used target false positive rate P_t (8 bytes), *(iii)* the used seed value (8 bytes), *(iv)* the *BloomUpdate* flag (2 bytes), and *(v)* the addresses inserted in the filter. The SPV client can add a pointer in the *ECKey* class of Bitcoinj in order to link each Bitcoin address to the Bloom filter that embeds it; the size of the pointer is roughly 2 bytes per address. This amounts to a total storage of $2N + 20$ bytes per Bloom filter. For example, when $N = 100$, the SPV client needs to store an additional 220 bytes per Bloom filter in order to reconstruct the filter upon restart; clearly, this overhead can be easily tolerated in existing SPV client implementations.

Clearly, our proposed solution can be directly integrated within existing SPV clients, and only incurs in small modifications to existing client implementations. Moreover, our solution does not incur additional overhead on the SPV clients—apart from the pre-generation of N Bitcoin addresses (which is only done at setup time), and the storage space required for each generated Bloom filter. Note that our solution requires SPV clients to outsource all their filters to full Bitcoin nodes since different filters embed different sets of their addresses. Therefore, N should be carefully chosen. If a user needs many addresses, choosing a larger N would avoid the generation of a larger number of Bloom filters. We leave the task of finding the best parameters when constructing a Bloom filter as an interesting direction for future work. Since users typically have few hundred Bitcoin addresses, we argue that this overhead can be largely tolerated given the current usage patterns in Bitcoin. Notice that the larger is N, the smaller are the number of Bloom filters created by the SPV client. Note that, in our solution, an SPV client should not outsource more than one Bloom filter to each regular node that it connects to. Indeed, if the client outsources several filters to the same node, then the regular node can correlate these filters (even if they do not embed the same elements).

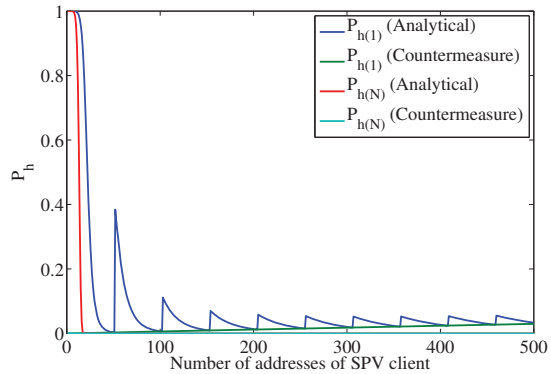


Figure 6: Evaluation of our countermeasure. Here, we measure $P_{h_{(1)}}$ and $P_{h_{(N)}}$ given $P_t = 0.05$ and assuming the current SPV client implementation. In computing $P_{h_{(1)}}$, we assume that the SPV client did not restart since initialization.

In Figure 6, we analytically compute P_h given our countermeasure. Clearly, our countermeasure considerably decreases P_h, especially for clients that have a small number of addresses.

Additional Insights.

We point out that an adversary which has access to the Bitcoin block chain can observe that Bitcoin addresses which recently were used by an SPV client do match the Bloom filter of that client. This knowledge clearly reduces the privacy of newly used addresses. Here, we stress that every newly used address of an SPV client is likely to match the Bloom filters of a fraction P_t of all other SPV clients. In our solution, P_t therefore defines the minimum anonymity set size of each address.

One alternative would be to embed existing Bitcoin addresses that do not belong to the node, when constructing each Bloom filter. Subsequently, whenever an SPV client needs to generate a new Bitcoin address, it will choose a Bitcoin address which matches its existing Bloom filter[7]. However, this alternative also results in an anonymity set defined by P_t; it also comes at the expense of computational overhead incurred on the SPV clients. In an experiment that we conducted using a Bloom filter B_t of size 1608 bits with $P_t = 0.05\%$ and $m = 102$, we were able to generate three new Bitcoin addresses which match B_t; these addresses were found after 240,351, 415,877, and 5,767,346 tries, respectively.

Moreover, we point out that our analysis and solution do not address the case where the adversary can link addresses by using side-channel information from the Bitcoin block chain, namely:

Filtering false positives by date: SPV clients only appeared in the second half of 2011, which is 1.5 years after Bitcoin started. Therefore, if a Bitcoin address which was created before 2011 matches the Bloom filter of an SPV client, the adversary can label it, with high confidence, to be a false positive, and not a Bitcoin address of the node. However, we note that the number of addresses created from 2011 exceeds by far that corresponding to the period from 2009-2011, during which Bitcoin was still not widely used at the time.

Clustering Bitcoin addresses: The adversary can make use of techniques such as [9, 15, 19] to link/cluster certain Bitcoin addresses based on user behavior, transaction amounts, etc., and assess whether an address matching a Bloom filter is a

[7]That can be achieved, e.g., by constantly generating a new Bitcoin address until it matches its existing Bloom filter.

false positive or a true positive. Here, we point out that our proposed solution can be used in conjunction with existing solutions such as [8, 20] to prevent the linking/clustering of addresses using such techniques.

7. RELATED WORK

User Privacy in Bitcoin: Bitcoin has received considerable attention from the research community [3, 10, 14–16, 18].

In [9], Androulaki *et al.* evaluate user privacy in Bitcoin and show that Bitcoin leaks considerable information about the profiles of user. In [4], Elias investigates the legal aspects of privacy in Bitcoin. In [23], Reid and Harrigan explore user anonymity limits in Bitcoin. In [24], Ron and Shamir investigate how users move BTCs between their various accounts in order to better protect their privacy. In [19], the authors investigated the possibility of linking addresses of the same user together by utilizing the Bitcoin peers network address information (IPs). Miers *et al.* introduced in [20] ZeroCoin, a cryptographic extension to Bitcoin that augments the protocol to prevent the tracing of coin expenditure. In [8], Androulaki and Karame proposed an extension of ZeroCoin to hide the transaction values and address balances in the system.

Privacy in Bloom Filters: As far as we are aware, Mullin *et al.* [21] were the first to propose an estimate the false positive rate of Bloom filters. In [13], Christensen *et al.* propose a novel technique for computing the false positive rate, which results in tighter estimates when compared to [21]. In [11], Bianchi *et. al* quantify the privacy properties of Bloom filters; their analysis, however, does not address the privacy provisions when the adversary has access to multiple Bloom filters originating from the same entity. In [22], Nojima *et. al*, propose a cryptographically secure privacy-preserving Bloom-filtering protocol based on blind signatures; this proposal, however, incurs additional computational load on SPV nodes.

8. CONCLUSION

In this paper, we explored the privacy provisions due to the integration of Bloom filters in SPV clients. Our results show that Bloom filters incur serious privacy leakage in existing SPV client implementations. More specifically, we show that a considerable number of the addresses of users of SPV clients which possess a modest number of Bitcoin addresses (e.g., < 20) are leaked by a single Bloom filter. Moreover, we show that a considerable number of the addresses of users is leaked if the adversary can collect two different Bloom filters issued by the same node, irrespective of the target false positive rate of the filter, and of the number of addresses owned by the user. Finally, we propose a lightweight solution that enhances the privacy offered by Bloom filters; our proposal can be integrated within existing SPV client implementations with minimum modifications.

9. REFERENCES

[1] Core Development Status Report # 1 - Bitcoin, Available from `https://bitcoinfoundation.org/2012/11/01/core-development-status-report-1/`.

[2] BitcoinJ, Available from `http://bitcoinj.github.io/`.

[3] Bitcoin Gateway, A Peer-to-peer Bitcoin Vault and Payment Network, 2011. Available from `http://arimaa.com/bitcoin/`.

[4] Bitcoin: Tempering the Digital Ring of Gyges or Implausible Pecuniary Privacy, 2011. Available from `http://ssrn.com/abstract=1937769ordoi:10.2139/ssrn.1937769`.

[5] Bitcoin Blockchain parser, 2013. Available from: `https://github.com/znort987/blockparser`.

[6] Bitcoin Wallet, Android, 2014. Available from: `https://play.google.com/store/apps/details?id=de.schildbach.wallet`.

[7] BitcoinJ, privacy assumptions, 2014. Available from: `https://github.com/bitcoinj/bitcoinj/blob/ee2a91010e5cf66299684160d6a48a108ff2299b/core/src/main/java/com/google/bitcoin/core/PeerGroup.java#L250`.

[8] Elli Androulaki and Ghassan Karame. Hiding transaction amounts and balances in bitcoin. In *Proceedings of TRUST*, 2014.

[9] Elli Androulaki, Ghassan Karame, and Srdjan Capkun. Evaluating user privacy in bitcoin. 2013. `http://eprint.iacr.org/2012/596.pdf`.

[10] S. Barber, X. Boyen, E. Shi, and E. Uzun. Bitter to Better - How to Make Bitcoin a Better Currency. In *Proceedings of Financial Cryptography and Data Security*, 2012.

[11] Giuseppe Bianchi, Lorenzo Bracciale, and Pierpaolo Loreti. Better than nothing privacy with bloom filters: To what extent? In *Privacy in Statistical Databases*, pages 348–363. Springer, 2012.

[12] Burton H Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

[13] Ken Christensen, Allen Roginsky, and Miguel Jimeno. A new analysis of the false positive rate of a bloom filter. *Information Processing Letters*, 110(21):944–949, 2010.

[14] C. Decker and R. Wattenhofer. Information Propagation in the Bitcoin Network. In *13-th IEEE International Conference on Peer-to-Peer Computing*, 2013.

[15] Meiklejohn et al. A fistful of bitcoins: Characterizing payments among men with no names. In *Proceedings of the 2013 Conference on Internet Measurement Conference*, IMC '13, pages 127–140, New York, NY, USA, 2013. ACM.

[16] Arthur Gervais, Ghassan Karame, Srdjan Capkun, and Vedran Capkun. Is bitcoin a decentralized currency? *IEEE Security and Privacy Magazine, 2014*.

[17] Mike Hearn. Connection bloom filtering, 2012. Available from: `https://github.com/bitcoin/bips/blob/master/bip-0037.mediawiki`.

[18] Ghassan O. Karame, Elli Androulaki, and Srdjan Capkun. Double-spending fast payments in bitcoin. In *Proceedings of the 2012 ACM conference on Computer and communications security*, CCS '12, pages 906–917, New York, NY, USA, 2012. ACM.

[19] Philip Koshy, Diana Koshy, and Patrick McDaniel. An analysis of anonymity in bitcoin using p2p network traffic. 2014. `http://fc14.ifca.ai/papers/fc14_submission_71.pdf`.

[20] Ian Miers, Christina Garman, Matthew Green, and Aviel D. Rubin. Zerocoin: Anonymous Distributed E-Cash from Bitcoin. 2013.

[21] James K Mullin. A second look at bloom filters. *Communications of the ACM*, 26(8):570–571, 1983.

[22] Ryo Nojima and Youki Kadobayashi. Cryptographically secure bloom-filters. *Transactions on Data Privacy*, 2(2):131–139, 2009.

[23] F. Reid and M. Harrigan. An Analysis of Anonymity in the Bitcoin System. *CoRR*, 2011.

[24] Dorit Ron and Adi Shamir. Quantitative analysis of the full bitcoin transaction graph. 2013. `http://eprint.iacr.org/2012/584.pdf`.

[25] S. Nakamoto. Bitcoin: A Peer-to-Peer Electronic Cash System, 2009.

[26] S Joshua Swamidass and Pierre Baldi. Mathematical correction for fingerprint similarity measures to improve chemical retrieval. *Journal of chemical information and modeling*, 47(3):952–964, 2007.

OSPF Vulnerability to Persistent Poisoning Attacks: A Systematic Analysis

Gabi Nakibly
CS Department
Technion - IIT
gnakibly@cs.technion.ac.il

Adi Sosnovich
CS Department
Technion - IIT
sadisos@cs.technion.ac.il

Eitan Menahem*
Cyber Security Labs
Ben-Gurion University
eitanme@il.ibm.com

Ariel Waizel
Cyber Security Labs
Ben-Gurion University
arielw@post.bgu.ac.il

Yuval Elovici
Cyber Security Labs
Ben-Gurion University
elovici@bgu.ac.il

ABSTRACT

Open Shortest Path First (OSPF) is one of the most widely deployed interior gateway routing protocols on the Internet. The most common attack vector against OSPF is spoofing of routing advertisements on behalf of a remote router. OSPF employs a self-defense "fight-back" mechanism that quickly reverts the effects of such attacks. Nonetheless, some attacks that evade the fight-back mechanism have been discovered, making it possible to persistently falsify routing advertisements. This type of attacks are the most serious threat to a routing protocol since they allow an attacker to gain persistent control over how traffic is routed throughout the network. This shows that despite its maturity, the OSPF specification is not without security flaws and may have still-unknown vulnerabilities. In this work we systematically analyze – manually and by formal verification – the OSPF specification for additional vulnerabilities in the fight-back mechanism. Our analysis uncovered a fundamental security flaw in OSPF that allows a simple means for an attacker to evade the fight-back mechanism. Most major router vendors acknowledged the existence of this vulnerability in their products. Fortunately, our analysis strongly indicates that no other vulnerabilities in the fight-back mechanism are likely to exist.

Categories and Subject Descriptors

C.2.2 [**Computer Systems Organization**]: Computer Communications Networks—*Network Protocols, Routing Protocols*; C.2.0 [**Computer Systems Organization**]: General—*Security and Protection*; D.2.4 [**Software**]: Software Engineering—*Software Verification, Model Checking*

General Terms

Security, Verification

Keywords

Routing Security, OSPF, Formal Verification

*Eitan is currently at IBM's Cyber Security Center of Excellence, Israel.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ACSAC December 08 - 12 2014 New Orleans, Louisiana USA
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
Copyright 2014 ACM 978-1-4503-3005-3/14/12
http://dx.doi.org/10.1145/2664243.2664278 ...$15.00

1. INTRODUCTION

Open Shortest Path First (OSPF) is one of the most popular interior gateway routing protocols on the Internet. Its aim is to allow routers within a single autonomous system (AS) to construct their routing tables, while dynamically adapting to changes in the autonomous system's topology. OSPF is currently used within many autonomous systems on the Internet. It was developed and standardized by the IETF's OSPF working group. This work is concerned with version 2 of the protocol [15], which was specifically designed for IPv4 networks; hence, it is essentially the only version used today. Version 3 [8] has been standardized to accommodate IPv6 networks but has seen little deployment to date.

OSPF is a link-state routing protocol, which means that each router advertises its links (and their costs) to neighboring routers and networks. These advertisements are termed Link State Advertisements (LSAs). The cost of a link is usually statically configured by the network administrator. Each LSA is flooded throughout the AS, where a router receiving an LSA from one of its neighbors resends it to all its other neighbors. Each router compiles an identical database of the LSAs of all the routers in the AS and uses it to obtain a complete view of the AS topology. It can thus employ Dijkstra's algorithm [9] to calculate the least cost paths between it and every other advertised network or router. As a result, a next hop is derived for each destination, thus forming the router's routing table.

Figure 1 illustrates the flooding of an LSA throughout the AS while the routers build their LSA database to construct their view of the AS topology.

The most common attack vector against OSPF is LSA falsification, in which an attacker advertises an LSA with false link information. This allows an attacker to poison the routers' view of the AS topology, thereby affecting their routing tables. The simplest attack of this type is to falsify the LSA of a router the attacker already controls. However, such an attack may have limited effectiveness since only one

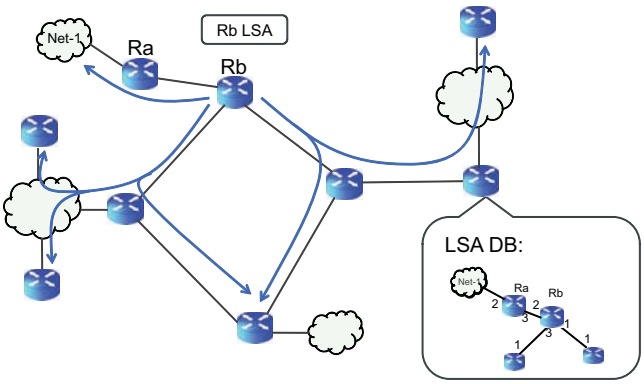


Figure 1: An example of LSA flooding (taken from [16])

LSA is falsified, thereby poisoning a small part of the AS topology – its immediate neighborhood. It would thus be difficult for an attacker to gain real control over the routing tables of other routers.

A much more powerful attack is falsifying an LSA on behalf of routers the attacker does *not* control (e.g., [19]). Using this technique, the attacker can poison a large portion of the view routers have of the AS topology, thereby changing their routing tables so that traffic flow is controlled by the attacker. Since the false LSA is flooded throughout the AS, it will eventually be received by the router on whose behalf the false LSA has been advertised. Once that router receives this LSA, it immediately triggers what is known as the "fight-back" mechanism. This mechanism advertises a newer instance of the LSA, which cancels out the false one, thereby reversing the effects of the attack. Hence, although such attacks cause instability and performance degradation to the AS, they do not have a lasting effect.

However, were a vulnerability to be found in the specification of the fight-back mechanism that would allow it to be circumvented, an attacker could persistently control the routing tables of all the routers in the AS. The attacker could thus gain complete control over the routing of traffic within the AS. The attacker could, for example, divert remote traffic to pass through a router or a network the attacker has access to in order to eavesdrop on the traffic or alter it. Furthermore, such an attack could also affect the routing of external traffic of other ASes routed through the current AS. In some cases, a well-crafted attack on a high volume transit AS may have a devastating effect on the network connectivity of many ASes throughout the Internet.

Two such vulnerabilities in the design of the fight-back mechanism were indeed discovered in the last few years. The details are given in Section 4. These discoveries show that the OSPF specification, despite its maturity, is still not without security weaknesses. We believe that a systematic and thorough analysis is called for in order to vet out other possible vulnerabilities.

In this work we present the results of a systematic search to uncover vulnerabilities in the OSPF specification that would allow an attacker to evade the fight-back mechanism. The search was conducted in two stages: manual analysis followed by a formal verification. The manual stage began by characterizing the general properties a vulnerability must have to allow an attacker to successfully evade the fight-back mechanism. This was followed by a targeted manual analysis of the specification to find such properties. This analysis uncovered a new security weakness in the design of the fight-back mechanism. In the second stage we aim to formally validate that no other fight-back evasion vulnerabilities exist in OSPF. We formally modeled the fight-back mechanism. Then we used a model checker to search for the same properties we found in the manual stage. The model checker was able to find all the previously known vulnerabilities as well as the newly discovered one. Moreover, no other vulnerabilities were found. This gives a strong indication (albeit not a complete proof) that there are no other weaknesses in the OSPF specification that allow an attacker to evade the fight-back mechanism.

Vendor	Advisory
Brocade	TSB 2013-165
Check Point	sk94490
Cisco	cisco-sa-20130801-lsaospf
D-Link	DLINK-2013-VUL0213
IBM	N1010309
Juniper	PSN-2013-08-987
NEC	NV13-006
Oracle	CPU-Oct2013

Table 1: The major vulnerable vendors and the advisories they published.

The newly discovered attack. The weakness uncovered by our analysis turns out to be even more powerful and simpler than the previously discovered vulnerabilities (see comparison in Section 5.1). We have successfully verified the vulnerability on Cisco and RouterOS (a software router). It has been reported to CERT/CC, who relayed it to all known OSPF router vendors in the market. Consequently, many router vendors acknowledged the existence of this vulnerability in their products. In most cases, the flaw was found in their entire line of OSPF-related products. The affected vendors are some of the most popular on the Internet. Table 1 lists the major affected vendors and the advisories they published. All these vendors released appropriate patches. The vulnerability is identified by CVE-2013-0149.

The paper is organized as follows. Section 2 gives a brief overview of the OSPF specification and principal functionality. Section 3 discusses the threat model we consider. Section 4 reviews known attacks that exploit vulnerabilities in the design of OSPF's fight-back mechanism. Section 5 discusses our manual analysis, presents the newly found vulnerability, evaluates its severity, and proposes mitigation measures. Section 6 reviews our formal verification analysis and its results. Finally, Section 7 concludes the paper.

2. OSPF BASICS

We begin with a brief overview of the OSPF protocol as specified in [15]. OSPF is a link state routing protocol: each router advertises an LSA containing the links to neighboring networks and routers and their associated costs. Each LSA is flooded throughout the AS. Routers construct a complete view of the AS topology by compiling all the LSAs they receive into a single database. From this global view routers compute their routing tables. Each router is identified by a parameter called router ID. A router ID is usually the IP address of one of the router's interfaces.

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|            LS age             |    Options    |    LS type    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                        Link State ID                          |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                     Advertising Router                        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                     LS sequence number                        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|         LS checksum           |             length            |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figure 2: The structure of an LSA header

A local network having two or more routers directly attached to it is called a transit network. A router connected to a transit network advertises a link to the network rather than to the neighboring routers. In addition, one of the neighboring routers is chosen to act as a designated router. This router advertises an LSA on behalf of the local network, in addition to its own LSA, advertising links back from the network to all the routers attached to the network (including itself).

Each LSA is advertised periodically every 30 minutes, by default. An LSA includes a Sequence Number field, which is incremented for every new instance. A fresh LSA instance with a higher sequence number will always take precedence over an older instance with a lower sequence number. In addition, an LSA includes an Age field indicating the elapsed time since the LSA's origination. When it reaches 1 hour, the LSA instance is removed from the LSA database.

For scalability reasons, OSPF allows an AS to be partitioned into areas. Flooding is confined to a single area while routing information is disseminated between different areas through a special backbone area. For simplicity of the presentation we shall consider only the case where an AS is composed of a single area.

The OSPF defines four basic types of LSA: Router-LSA, Network-LSA, Summary-LSA and AS-external-LSA. The most common type is Router-LSA, which is used to advertise the links of a given router. For brevity, throughout the paper we refer to a Router-LSA as simply LSA.

All LSAs begin with a common 20 byte header. Figure 2 depicts the LSA header. A description of the LSA header fields follows.

- LS age – The time in seconds since the LSA was originated.
- Options – Optional capabilities supported by the described portion of the routing domain.
- LS type – The type of LSA.
- Link State ID – Identifies the part of the AS that is being described by the LSA.
- Advertising Router – Identifies the router that originated the LSA.
- LS sequence number – The sequence number of the LSA.
- LS checksum – The checksum of the complete contents of the LSA.
- Length – The length in bytes of the LSA.

The standard specifies (Sec. 12.1 of [15]) that an LSA is identified by the three fields: LS type, Link State ID, and Advertising Router.

2.1 OSPF's Security Strengths

We next discuss the dominant security strengths of OSPF and explain why it is difficult for an attacker to persistently and significantly poison the routing table of a router it does not control. In order for an attacker to substantially alter a router's routing table, the attacker needs to poison large portions of the LSA database of that router. Since an LSA holds only a small part of the AS topology – only a single router and the links to its immediate neighbors – an attacker needs to falsify LSAs of multiple routers. In most cases the attacker have no control over the routers it wishes to falsify their LSAs.

In the following we list the properties of OSPF that make it impossible, in the general case, for an attacker to falsify LSAs of routers it does not control.

1. Flooding – Every LSA is flooded throughout the AS. In particular, this means that a false LSA advertised by the attacker will eventually be received by the router on whose behalf the false LSA has been advertised.
2. fight-back – Once a router receives a false instance of its own LSA, it immediately advertises a newer instance which cancels out the false one.

The fight-back mechanism, combined with the flooding property, should prevent an attacker from persistently and stealthily falsifying an LSA of a router the attacker does not control. Due to the flooding property we are assured that in the general case the fight-back LSA will be received by all routers in the AS.

3. THREAT MODEL

We adopt the common threat model found in the literature ([18, 19, 11, 16]). This model assumes the attacker has the ability to send LSAs to at least one valid router in the AS and that router processes them as valid LSAs. This assumption can be trivially achieved by an insider, namely an attacker who gained control over just a *single* router in the AS. We assume nothing about the location of the compromised router. In particular, we do not assume the attacker compromised a core or a centrally located router.

The attacker can gain control of a router, for example, by remotely exploiting an implementation vulnerability on the router. Several such vulnerabilities have been published in the past (e.g., CVE-2010-0581, CVE-2010-0580, and CVE-2009-2865). Note that we do not assume that the attacker can compromise an arbitrary number of routers within the AS, since in many cases it is difficult for an attacker to do that even if he has already comprised one router in the AS. One reason for this is the routers' firmware diversity. For example, it has been pointed out ([12]) that there are over 300,000 unique variants of Cisco's IOS. This diversity stems from a mixture of various hardware platforms, feature-sets, licensing agreements, and build procedures. This firmware diversity has largely made the deployment of reliable attacks difficult in practice. Moreover, usually with every attempt to take-over a router inevitably the chances of discovery

increase, hence the attacker may prefer to launch a single OSPF attack rather than compromise multiple routers.

OSPF employs per-link LSA authentication where each link is associated with a secret shared by all the routers directly attached to that link. A router authenticates a received LSA using the secret associated with the link on which it was received. Note that no end-to-end authentication is employed in OSPF. The threat model assumes that the attacker knows only the secrets associated with the links that are directly attached to the router it controls. It does not know the secrets associated with other links in the AS. Consequently, the attacker may only be able to send false LSAs to its immediate neighbors (which they in turn flood to their neighbors). Specifically, the attacker can not send an LSA directly to a remote router.

We also note that in order to realize the assumptions of the threat model it is not necessary for the attacker to compromise a router in the AS. It is sufficient to know the secret of one of the links in the AS. This would allow an attacker to remotely send false LSAs that would be successfully authenticated by the routers attached to that link.

The attacker's goal is to significantly poison the routing tables of other routers within the routing domain of the compromised router. Because the attacker would like to control the routing domain for an extended period of time, the poisoning should be persistent and stealthy while reducing the attack's footprints and the trails leading to the attacker.

4. KNOWN ATTACKS

There are two previously published attack techniques that exploit design vulnerabilities of OSPF in order to evade the fight-back mechanism, thereby allowing the attacker to persistently falsify an LSA of a route it does not control.

Jones et al. [11] introduced the first known attack that evades the fight-back mechanism – called 'periodic injection'. The attack exploits a vulnerability in which the fight-back mechanism is triggered only *after* the router has flooded the false LSA (which was advertised on its behalf) to its neighbors. Since the OSPF specification prevents a router from originating two instances of its own LSA within less than 5 seconds. The victim router must wait for that time period before it is able to send the fight-back LSA. Hence, if the attacker periodically advertises the false LSA at least once every 5 seconds, it will ultimately prevent the attacker from issuing any fight-back LSAs.

Nakibly et al. [16] presented another attack technique that evaded the effect of the fight-back mechanism called 'Disguised LSA'. This attack is more severe than the attack of [11] since only two malicious packets are required to persistently falsify an LSA of a router. The technique exploited a vulnerability in the fight-back mechanism which does not advertise a correcting LSA if the received LSA is "identical" to last valid LSA advertised by that router. The OSPF spec considers two instances of an LSA to be identical if they have the same values in the following three fields: Sequence Number, Checksum, and Age. In fact, two LSAs are considered identical even if their Age fields differ by up to 15 minutes (and the Sequence Number and Checksum fields are the same). The key point is that the spec considers these two LSAs to be the same even if the actual advertised links in the LSAs differ. The attack exploited this by advertising a false LSA that seems to be identical to a future fight-back LSA sent by a victim router. During the launch of the attack the fight-back mechanism is triggered, however other routers ignore this fight-back LSA by considering it to be a duplicate of the false LSA. Thereby, rendering the fight-back LSA ineffective.

5. MANUAL ANALYSIS

We note that there is a general lesson to be learned from the vulnerabilities outlined above: a defense mechanism must always be fully aligned with the functionality that it protects. In other words, a defense mechanism must cover the entire relevant attack surface of the functionality it aims to defend. The slightest misalignment jeopardizes the entire functionality. In our case the defense mechanism is the fight-back mechanism, the protected functionality is the routing table calculation, and the attack surface is the receipt of LSAs. The two fight-back evasion attacks described above are successful examples of techniques that exploit such misalignments, enabling the attacker to send a false LSA that does not trigger a fight-back while still affecting the routing table calculation of the routers that install it.

In the periodic injection attack the misalignment is due to the fight-back being triggered after the flooding step. In the disguised LSA attack the misalignment is due to the fight-back being triggered only on the basis of the LSA's header fields and not on its content.

Note that not every false LSA that does not trigger a fight-back results in a successful attack. For example, an LSA that is advertised on behalf of a nonexistent – phantom – router will never trigger a fight-back; however, it will not affect the routing table calculation of routers that install that LSA[1].

One may well ask whether the OSPF specification has other security weaknesses in which the fight-back mechanism is not fully aligned with the routing table calculation. To answer this question, we reviewed, in detail, both the conditions which trigger the fight-back mechanism, as described in Section 13.4 of [15], and the procedures of the routing table calculation, as described in Section 16 of [15]. Then we systematically searched for misalignments between them.

To find such misalignments we first enumerated all the fields in the LSA header which may potentially affect the routing table calculation and the values which they must hold in order for an LSA to actually influence this calculation. This resulted in the following list.

1. LS age must be lower than 3600 (the maximum value age value).
2. Link State ID must belong to an existing router or network in the AS.
3. LS sequence number must be higher than the value of the valid LSA to be replaced.

As a second stage of our search we closely inspected the cases that may trigger a fight-back. These cases can then be easily translated into conditions on LSA header fields. Then we asked what should be the header of an LSA that, on one hand, obey the conditions to influence the routing

[1]The OSPF specification mandates that only links advertised by both their ends can affect the routing table. Since no real router will advertise a link to the phantom router, the false links advertised on behalf of it will be ignored during the routing table calculation.

table calculation, but, on the other hand, does *not* obey the conditions of the fight-back. This analysis led us to the new vulnerability we uncovered. The following section explains the vulnerability in detail.

5.1 The New Attack

The Link State ID field of a Router-LSA equals the ID of the router whose links are described by the LSA. Moreover, since a Router-LSA is originated by the router described by that LSA, the Advertising Router field of the LSA will also equal the router's ID. Therefore, we note that in every Router-LSA, the above two fields – Link State ID and Advertising Router – are expected to have the same exact value. The first part of the vulnerability arises because the standard neglects to specify a sanity check to verify this equality on LSA reception. Thus, a Router-LSA is considered valid even if it has different values in these two fields. In the following we explain how an attacker can benefit from this.

According to Section 13.4 of [15], a router is triggered to send a fight-back LSA only if it receives a false LSA in which

> "the Advertising Router is equal to the router's own Router ID."

This means that no fight-back shall be triggered by the victim router as long as the Advertising Router field of a received LSA is *not* equal to the victim router's ID. This is true even if the Link State ID field of that LSA equals the victim router's ID. Namely, no fight-back is triggered even if the false LSA claims to describe the links of the victim router itself.

An attacker can exploit the above vulnerability as follows. Let us assume the attacker wishes to advertise a false LSA on behalf of some victim router, R_v. The attacker will advertise on its local links the false LSA while having a header in which:

- Link State ID = ID of R_v,
- Advertising Router $\neq$ ID of R_v.

The false LSA shall be flooded throughout the AS as usual and will eventually be received by all routers, including the victim router R_v. The OSPF standard guarantees that such false LSA will not trigger fight-back by R_v. Consequently, all routers within the AS – including R_v – install the false LSA in their LSA databases. Thus, the attacker achieves complete poisoning of the LSA databases of all routers within the AS.

However, does this LSA database poisoning indeed poison the routing tables as well? Recall that an LSA is identified by the combination of the following three fields:

1. LS type,
2. Advertising Router,
3. and Link State ID.

This means that the false LSA advertised by the attacker has a different identifier than that of the valid LSA of the victim router. They differ in the Advertising Router field's value. The value of a valid LSA equals that of the victim router's ID while the value of the false LSA does not. Therefore, the false LSA should *not* replace the valid LSA in the LSA databases. Consequently, following the attack, both LSAs are expected to reside in the LSA databases of all routers.

Now let us turn to the second part of the vulnerability. The OSPF standard (Section 16.1) specifies that during the routing table calculation phase LSAs are looked up in the LSA database

> "based on the Vertex ID".

The vertex ID refers in the standard to the LSA's Link State ID field. This means that while a router calculates its routing table, it identifies LSAs on the basis of their Link State ID field only and not on the basis of the full LSA identifier, which also includes the Advertising Router and LS type fields. That the lookup is based on the partial identifier is explicitly reiterated in footnote 14 of the standard, which also explains the motivation behind this:

> "[14] There is one instance where a lookup must be done based on partial information. This is during the routing table calculation, when a network-LSA must be found based solely on its Link State ID. The lookup in this case is still well defined, since no two network-LSAs can have the same Link State ID."

Namely, the standard motivates this lookup by saying that during the routing table calculation the full identifier of Network-LSAs is not known. However, it seems that the standard unduly generalized this partial information lookup to Router-LSAs as well, despite the full identifier of a Router-LSA being known during the routing calculation phase[2].

The above gives rise to the following question: which LSA will be fetched from the LSA database during the routing table calculation: the valid LSA of the victim router or the false LSA advertised by the attacker? Remember, both LSAs reside side by side in the LSA database of each router in the AS. Both LSAs have the exact same value in their Link State ID field – the ID of the victim router. Unfortunately, the standard fails to answer this question; it does not consider the case where two different LSAs have the same Link State ID field. Because this question is not addressed in the standard, the answer is thus dependent on the implementation. An OSPF implementation that fetches the valid LSA during the routing table calculation is oblivious to the attack. However, an OSPF implementation that fetches the false LSA is completely vulnerable to the attack.

As noted in Table 1 the attack is indeed successful against many OSPF implementations which were found to be vulnerable. These vendors and others have acknowledged their routers are vulnerable to the attack and released suitable patches.

In Table 2 we compare the attack techniques of [11] and of [16] with the new attack technique we uncovered. All three attacks persistently falsify an LSA of a victim router not controlled by the attacker. However, it seems that our attack is more powerful that the others. It does not trigger the victim router to originate a fight-back LSA during the attack. It requires only a single malicious packet to carry out the attack. Finally, it can be used to poison the entire

[2]Note that for Router-LSAs, the Advertising Router field must be equal to the Link State ID field and hence must also be known.

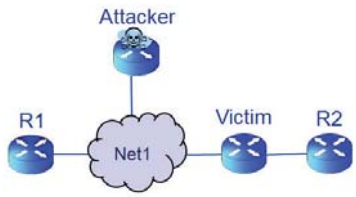


Figure 3: The topology used for the experiments

AS: *all* the routers within the attacked AS will install the false LSA in their LSA databases. This includes the victim router on whose behalf the false LSA has been advertised. This last property is unique to our attack.

5.2 Attack's Effects on Commercial Routers

We demonstrate the attack's effects on two commercial routers: a hardware router and a software router. We chose a Cisco 7200-series router as a hardware router. For the software router we chose MikroTik's RouterOS [4]. On both routers the attack was successful and reliable. Figure 3 depicts the sample network topology we used for both experiments. We labeled the routers as follows:

- Victim – The router on whose behalf a false LSA is advertised
- Attacker – The router that was compromised by the attacker, and which originates the false LSA
- R1 – The router co-located on the same local network as the attacker and victim
- R2 – A router connected to the victim router

The victim, attacker and R1 reside on the same local network – Net1. The victim and R2 reside on a different local network. The attacker sends out a false LSA onto Net1 on behalf of the victim . The false LSA has the following fields in its header:

- Link State ID = victim's ID
- Advertising Router $\neq$ victim's ID (some arbitrary value)
- LS sequence number $>$ valid LSA's sequence number

The attacker advertises an LSA on behalf of the victim saying that the victim has only a single link which is connected to Net1. In particular, *no* link is advertised which connects the victim to R2. The purpose of this false LSA is to poison the LSA database of R1 so that no route to R2 will exist in its routing table. The attacker floods the false LSA on Net1. Note that we put R1 and the attacker on the same local network to simplify the presentation. R1's LSA database would have been poisoned even if R1 had not resided on the local network of the attacker.

Cisco.

We used network simulation software called GNS3 [3] that builds on top of a Cisco IOS emulator called Dynamips [1]. We emulated a Cisco 7200 router with a commercial image of a stable release of IOS – 15.0(1)M[3]. We configured the topology described above in the GNS3 software. The attack was successful. Here are our main findings:

1. The false LSA replaces the valid LSA. The false LSA is not only installed in the LSA databases of all the routers but also replaces the valid LSA. This happens in all routers, including the victim router. Consequently, all routers *always* consider the false LSA during their routing table calculation.
2. Routing tables of all routers except the victim router are poisoned. All routers except the victim build their routing tables according to the false LSA advertised by the attacker. In particular, the route to R2 in R1's routing table is gone.
3. The routing table of the victim is erased. Probably due to the bogus value of the Advertising Router field in the false LSA, the victim does not install routes in its routing table. Consequently, all the entries in the victim's routing table which are sourced from the OSPF process are deleted. This essentially empties the routing table, effectively making the victim unable to perform its routing function. This erasure of the victim's routing table is permanent. The victim is not able to spontaneously recover from this. To recover, the OSPF process must be reinitialized.
4. The attack can be undone. The attacker can easily undo the effects of the attack by sending another false LSA on behalf of the victim but this time with an Advertising Router that equals the victim's ID. This LSA must have a higher sequence number than that of the previous attack LSA. The new false LSA triggers a fight-back by the victim, which originates a newer instance of the valid LSA that replaces the false LSA in the LSA databases of all routers. In particular, the route to R2 is reinstated in R1's routing table and the victim's routing table reverts to normal.

RouterOS.

RouterOS is a software router that can be run on an x86 PC. We tested the newest version of RouterOS – version 6.2. We set up on a single PC four virtual machines running Ubuntu version 10.4 Desktop. Each virtual machine ran a single instance of RouterOS. The virtual machines were configured in the topology described above. The attack was successful. Here are our main findings:

1. The false LSA is installed in the LSA databases of all routers. Note that unlike the Cisco case, here the valid LSA still remains in the LSA databases.
2. All routers *always* consider the false LSA during their routing table calculation. Routing tables of all routers except the victim router are poisoned, as in the Cisco case. We assume that the false LSA is always fetched rather than the valid one since it was installed in the database on a later time, however we can not verify this.
3. The routing table of the victim is erased, as in the Cisco case.
4. Triggering a fight-back using a second false LSA does not undo the attack's effects.

[3]IOS 15.0(1)M was released on November 2009. Cisco acknowledged that IOS's latest release – 15.3 – is also vulnerable as well as most other versions of IOS (around 600!).

Attack property	Periodic injection [11]	Disguised LSA [16]	Our attack
Persistence	Yes	Yes	Yes
Triggers fight-back	No	Yes	No
Number of packets to send	Many (1 packet per 5 sec.)	2	1
Entire AS is poisoned	No	No	Yes
Victim router is poisoned	No	No	Yes

Table 2: A comparison of our attack to previously published persistent attacks. The green cells in each row contain the best value for the corresponding attack property

5.3 Mitigation Measures

The attack we presented relies on the following three vulnerabilities in the standard:

1. There is no sanity check on Router-LSA reception to verify that the Link State ID field equals the Advertising Router field.
2. A fight-back is triggered only when a false LSA is received having an Advertising Router field that equals the ID of current router (Section 13.4).
3. During the routing table calculation phase, Router-LSAs are looked up in the LSA database using only on the Link State ID field (Section 16.1).

Elimination of any one of the three vulnerabilities shall successfully mitigate the attack. Here are possible mitigation measures for each vulnerability:

1. Specify a check on Router-LSA reception to verify that the Link State ID field equals the Advertising Router field; otherwise, the LSA should be discarded.
2. A fight-back shall be triggered if a false LSA is received having either an Advertising Router or a Link State ID that equals the ID of current router.
3. During the routing table calculation phase, Router-LSAs are looked up using their full identifier.

Probably the easiest mitigation measure for a vendor to implement is the first one. It is least likely to have unexpected affects on other parts of the OSPF implementation. Indeed, this measure was implemented by many vendors found to be vulnerable to the attack. We note, however, that the third measure should ideally be implemented as well, since the third vulnerability is due to an oversight of the protocol's designers rather than an intended functionality.

6. FORMAL VERIFICATION

Model checking is a widely used formal verification method. Its goal is to automatically determine whether a system satisfies a desired specification. A model of the system is systematically searched for a violation of the specification. If such a violation is found, the system behavior that caused this violation is returned to the user. Model checking has shown notable success in verifying software systems, hardware designs, communication protocols, controllers, and more [6, 17].

Some past works [14, 10, 13, 5] have analyzed the security of OSPF and other routing protocols using model checking. Their goal was to check OSPF functionality in general. Since OSPF is a complex protocol, none of these works actually modeled the protocol in its entirety but rather abstracted away many of the details. In this work, we extend a previous formal model of OSPF that was described in [5]. In this extension, the fight-back mechanism and routing table calculation are modeled in more detail. Such extended modeling allows us to better search for attacks that may exploit intricate details of the fight-back mechanism.

6.1 Modeling OSPF

The model is composed of a fixed network topology that contains legitimate OSPF routers and a single malicious router. The modeled functionality includes the LSA message structure, the LSA flooding procedure, the fight-back mechanism, and the routing table calculation. In the following we include a pseudo-code that gives a general overview of the model we used. For the sake of brevity and clarity, the pseudo-code does not encompass all of the model's details. The code of our full model can be found here [2]. The next sections explain the pseudo-code.

6.1.1 *The main function*

The main function of the model is a loop, where in each loop iteration any of the routers (including the attacker) can run their procedure once. Each loop iteration is considered a cycle. The flooding timers of each router are decremented in every cycle (see Section 6.1.3 for an explanation of the flooding timers).

We *mark* LSAs which are deemed to affect the routing table. We count the number of cycles each LSA resides in an LSA database. The last portion of the main loop is the assertion of the model's specification (Section 6.1.5).

6.1.2 *The attacker model*

The attacker generates an LSA with arbitrary content. This models an attacker with complete control over the false LSAs it sends out. Note that the identity of the victim router on whose behalf the false LSA is sent is not predefined. It can be any one of the legitimate routers. The model checker will cover every possible false LSA during its search for an attack. Note that the attack may be composed of a sequence of false LSAs sent by the attacker.

6.1.3 *The router model*

In the initial state, each router R has in its LSA database LSAs originated by R, and for each link described in these LSAs, there is also an LSA in the database with a link back to R. Namely, there is a link from one of R's neighbors back to R. We need these links in our model because, as per the OSPF standard, a link can be considered in the routing table calculation only if there are links in the reverse direction.

Listing 1: The model's main loop

```
main()
{
  loop {
    for every router R
    {
      Router(R); //execute R's procedure

      if (R.flooding_timer > 0)
        R.flooding_timer--;

      for every LSA in R.LSA_DB
      {
        LSA.counter++;

        //violation of this assertion
        //indicates a successful attack
        assert(
          LSA.mark == false ||
          LSA.originated_by_attacker ==
              false ||
          LSA.counter < MIN_COUNTER
          );
      }
    }

    Attacker(); //execute attacker's
        procedure

    num_cycles++;
  }
}
```

Listing 2: The attacker's procedure

```
Attacker()
{
  LSA = generate_arbitrary_LSA();
  flood(LSA);
}
```

Each router has a flooding timer that determines how many cycles it should wait before originating an LSA. When the timer is set to 0 the router can originate its own LSA without any delay. After the router floods its own LSA, its timer is set to MinLSInterval, a predefined constant that determines by how many cycles to delay the next LSA origination. The timer is decreased by 1 in each cycle for which its value is greater than 0.

When the router R receives an LSA, it first checks that it is valid (i.e., properly formatted and received from a valid neighbor). If so, it checks whether such an LSA exists in its LSA database. If it exists and is considered newer than the database copy or if it does not exist in the database, then R floods this LSA. Afterward it checks whether the LSA is self-originated. A self-originated LSA is a one whose Advertising Router field equals R's ID. If this conditions holds, a fight-back is triggered, in which case the LSA will not be installed in the database. The fight-back itself is delayed due to the flooding of the false LSA prior to discovering that it was self-originated.

Otherwise, R adds this LSA to its LSA database and

Listing 3: A valid router procedure

```
Router(R)
{
  //fetch the LSA at the top of the
  //router's incoming queue
  LSA = R.in_queue.dequeue()

  if (!R.valid(LSA))
    continue;

  if(LSA is newer than one in R.LSA_DB)
  {
    R.flood(LSA);

    if (R.is_self_originated(LSA))
    {
      R.flooding_timer = minLSInterval;
      FB_LSA = R.generate_fight_back();

      //replace existing queued fight-back
      //with the new one.
      if (R.flooding_queue.size() > 0)
        R.flooding_queue.dequeue()
      R.flooding_queue.enqueue(FB_LSA);
    }

    else
    {
      R.LSA_DB.update(LSA) ;
      R.calcRT_flag = True ;
    }
  }

  if(R.flooding_timer == 0)
  {
    if (R.flooding_queue.size() > 0)
    {
      LSA = R.flooding_queue.dequeue()
      R.flood(LSA);
    }
  }

  R.RoutingTableCalculation();
}
```

triggers the routing table calculation. The calculation logic must be modeled since not every LSA installed in the LSA database of a router eventually affects that router's routing table.

6.1.4 The routing table calculation

To keep the model compact, we do not model the entire routing table calculation logic. We are not interested in modeling the calculated routing table, but only the functionality that determines which LSAs will ultimately affect it. We start by considering R as reachable (it is the root of its shortest path's tree). Then, a lookup for R's LSA in its database is performed using the Link State ID field. Note that the fetched LSA may be either a valid LSA of R or a malicious LSA crafted by the attacker in a previous stage. For each advertised link in the fetched LSA, let W denote the neighbor on the other side of that link. If an LSA for W is found and contains a link back to R, then R's LSA is

Listing 4: The routing table calculation procedure

```
RoutingTableCalculation()
{
  if (!calcRT_flag)
    continue;

  calcRT_flag = False;
  //R is the current router
  R.reachable = True;

  for every V where V.reachable == True
  {
    //lookup in an arbitrary order within
        LSA database for V's LSA
    V_LSA = DB_lookup(V) ;

    for every link L in V_LSA
    {
      W = L.neighbor;
      W_LSA = DB_lookup(W);
      if (W_LSA has link back to V)
      {
        //V_LSA affects the routing table
        V_LSA.mark = True;
        W.reachable = True;
      }
    }
  }

  for (every Summary–LSA in LSA_DB)
  {
    if (LSA.AdvertisingRouter.reachable ==
        True)
      //LSA affects the routing table
      LSA.mark = True;
  }
}
```

marked and W is considered reachable in the shortest path tree. This loop is executed for every reachable router.

According to the OSPF standard, a Summary-LSA installed in the database affects the routing table if and only if its Advertising Router field contains an ID of a reachable router. Therefore, our model marks such LSAs only if this condition holds.

6.1.5 Specification

We say that an attack is successful if there is an LSA installed in a router for which all of the following three conditions hold:

1. The LSA was originated by the attacker.
2. The LSA is marked, i.e., it affects the routing table calculation.
3. The LSA's counter exceeds MIN_COUNTER (where MIN_COUNTER is some large value).

The last condition ensures that the attack is persistent, i.e., it is not reverted by the fight-back mechanism. The MIN_COUNTER is a predefined constant that determines the persistency of the attack. In our model, when an LSA that was originated by the attacker replaces an older instance of that LSA which was also originated by the attacker, then the counter value of the older LSA is copied to that of the newer one. This allows our model to find attacks in which the false LSA's instances are constantly changed by the attacker. These are also considered persistent attacks since the attacker has an uninterrupted effect on the routing table.

This specification is coded as an assertion in the model's main loop. If the model checker finds a state of the model which violates this assertion, then it must be the case that the attacker managed to persistently affect the routing table calculation while evading the fight-back mechanism.

6.2 Results

We implemented the above model in C and used the CBMC tool [7] to perform the model checking. The CBMC tool is a bounded model checker that receives a C program as input. It can check whether all possible execution paths (loops are checked up to some bound) of the C program satisfy an assertion by translating the C program into a SAT formula and by checking whether it is satisfiable using a SAT solver. It is also possible to add assumptions within the program in order to check only a subset of the execution paths that satisfy the assumptions.

6.2.1 Attacks found in our model

By running the model checker with our OSPF model implementation, we recreated the following attacks:

1. The disguised LSA attack was found, where the attacker sent two LSAs – a trigger LSA and a disguised LSA. This attack was also found on the original model from [5] without the extensions added here.
2. The periodic injection attack – where the attacker periodically sent LSAs on behalf of some router at a higher rate than MinLSInterval. Each LSA sent by the attacker had a sequence number greater than the previous one. Therefore, the delayed fight-back was also replaced and delayed for each new instance. Consequently, a sequence of malicious LSAs were present in the victim's LSA database more than MIN_COUNTER cycles, and the fight-back was never sent due to reset of the flooding timer with every received false LSA.
3. An attack that exploits the new vulnerability was found in our manual analysis stage. The output contained an execution path that described the above attack, where the LSA sent by the attacker was a Router-LSA having a Link State ID equal to the victim's ID but having a different value for the Advertising Router.

6.2.2 Correcting the vulnerabilities in the model

The CBMC model checker halts after it finds the first counterexample for the assertion in our model. Therefore, in order to find new vulnerabilities, we had to correct the ones found above.

1. In order to avoid receiving the disguised LSA attack, we changed the condition that triggers the fight-back mechanism, such that not only newer self-originated LSA instances will trigger it, but also instances that are considered identical to the LSA in the database, and were actually originated by the attacker. In that case the disguised LSA will also trigger a fight-back.

2. In order to avoid receiving the periodic injection attack, we set the MinLSInterval to 0. In that case the fight-back is not delayed thus the attack cannot be persistent.

3. In order to avoid receiving the new attack, we added an assumption that during LSA generation by the attacker, the Link State ID field must equal the Advertising Router field. As noted above, this condition was actually used in the patches released by most vendors who were found vulnerable to the attack.

We reran CBMC on the corrected model and no new attacks were found. This indicates that, once the above conditions are enforced, an attacker will not be able to craft a malicious LSA such that it will not trigger the fight-back mechanism while still affecting the routing table calculation.

Nonetheless, we note that model checking is only as good as the model itself. The above result cannot be considered "proof" that no further weaknesses exist in OSPF in general since we omitted details of OSPF operation that are not relevant to the fight-back mechanism. Moreover, since we used a bounded model checker, namely one that does not explore all the states of the model, the above result cannot be considered "proof" that no other weaknesses in the fight-back mechanism exist. Nonetheless, we believe this result provides a strong indication that there are no additional weaknesses in this mechanism.

7. CONCLUSIONS

We present a systematic security analysis of the OSPF protocol against fight-back evasion attacks. The analysis was conducted both manually and by formal verification. We found a persistent new attack on OSPF that exploits vulnerabilities in the OSPF standard itself. The attack is the first to allow complete poisoning of all routers within the AS. The attack is successful against OSPF implementations of some of the most popular router vendors on the Internet. Then, we proposed three simple mitigation measures to defend against the attack. Using model checking, we sought other persistent attacks that aim to evade the fight-back mechanism. The results gives an indication that once the known weaknesses are mitigated no other weakness in the fight-back mechanism exists.

The general lesson one may draw from this security analysis is that a defense mechanism must always be fully aligned with the functionality that it protects. The slightest misalignment or the tiniest functionality that is left uncovered jeopardizes the entire functionality.

8. REFERENCES

[1] Cisco IOS emualtor. http://dynagen.org/.

[2] Full OSPF Model Implementation. http://pastebin.com/1stZJRhy.

[3] Graphical network emulator. http://www.gns3.net.

[4] Mikrotik Routers and Wireless. http://www.mikrotik.com/software.html.

[5] Orna Grumberg Adi Sosnovich and Gabi Nakibly. Finding security vulnerabilities in a network protocol using parameterized systems. In *Proceedings of CAV*, 2013.

[6] Satish Chandra, Patrice Godefroid, and Christopher Palm. Software model checking in practice: An industrial case study. In *Proceedings of the 24th International Conference on Software Engineering*, ICSE '02, pages 431–441, 2002.

[7] Edmund Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ANSI-C programs. In Kurt Jensen and Andreas Podelski, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 2988, pages 168–176. 2004.

[8] R. Coltun, D. Ferguson, J. Moy, and A. Lindem. OSPF for IPv6. IETF RFC 5340, July 2008.

[9] Edsger. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.

[10] B. Fortz. On the evaluation of the reliability of OSPF routing in IP networks. Technical report, Institut d'administration et de gestion, 2001.

[11] E. Jones and O. Le Moigne. OSPF Security Vulnerabilities Analysis. Internet-Draft draft-ietf-rpsec-ospf-vuln-02, IETF, June 2006.

[12] Felix Lindner. Cisco IOS router exploitation. In Black Hat USA, 2009.

[13] S. U. R. Malik, S. K. Srinivasan, S. U. Khan, and L. Wang. A methodology for OSPF routing protocol verification. In *Proceedings of the 12th International Conference on Scalable Computing and Communications (ScalCom)*, 2012.

[14] P. Matousek, J. Ráb, O. Rysavy, and M. Svéda. A formal model for network-wide security analysis. In *Proceedings of ECBS 2008.*, pages 171–181, 2008.

[15] J. Moy. OSPF version 2. IETF RFC 2328, April 1998.

[16] Gabi Nakibly, Alex Kirshon, Dima Gonikman, and Dan Boneh. Persistent OSPF attacks. In *Proceedings of NDSS*, 2012.

[17] Corina Pasareanu, Matthew Dwyer, and Michael Huth. Assume-guarantee model checking of software: A comparative case study. In *Theoretical and Practical Aspects of SPIN Model Checking*, volume 1680, pages 168–183. 1999.

[18] Feiyi Wang, Brian Vetter, and Shyhtsun Felix Wu. Secure Routing Protocols: Theory and Practice. Technical report, North Carolina State U., May 1997.

[19] S. F. Wu, H. C. Chang, F. Jou, F. Wang, F. Gong, C. Sargor, D. Qu, and R. Cleaveland. Jinao: Design and implementation of a scalable intrusion detection system for the OSPF routing protocol. *ACM Transactions on Computer Systems*, 2:251–273.

Less is More: Cipher-Suite Negotiation for DNSSEC

Amir Herzberg*
Department of Computer Science
Bar Ilan University
Ramat Gan, Israel
amir.herzberg@gmail.com

Haya Shulman
Fachbereich Informatik
Technische Universität Darmstadt
Darmstadt, Germany
haya.shulman@gmail.com

Bruno Crispo
Department of Information Engineering and Computer Science (DISI)
University of Trento, Italy
crispo@disi.unitn.it

ABSTRACT

We propose a transport layer cipher-suite negotiation mechanism for DNSSEC standard, allowing name-servers to send responses containing only the keys and signatures that correspond to the cipher-suite option negotiated with the resolver, rather than sending *all* the signatures and keys (as is done currently).

As we show, a lack of cipher-suite negotiation, is one of the factors impeding deployment of DNSSEC, and also results in adoption of weak ciphers. Indeed, the vast majority of domains rely on RSA 1024-bit cryptography, which is already considered insecure. Furthermore, domains, that want better security, have to support a *number* of cryptographic ciphers. As a result, the DNSSEC responses are large and often fragmented, harming the DNS functionality, and causing inefficiency and vulnerabilities.

A cipher-suite negotiation mechanism reduces responses' sizes, and hence solves the interoperability problems with DNSSEC-signed responses, and prevents reflection and cache poisoning attacks.

Keywords

Cipher suite negotiation, DNSSEC, DNS security, DNS interoperability.

1. INTRODUCTION

A *cipher-suite* is an ordered set of (one or more) cryptographic algorithms, each implementing a corresponding function among the functions used by a cryptographic protocol. For example, the `RSA_WITH_RC4_128_MD5` cipher suite uses RSA for key exchange, RC4 with a 128-bit key for bulk encryption, and MD5 for message authentication. *Cipher-suite negotiation* refers to the process of selecting the cipher-suite to be used in a protocol between two (or more) parties, among multiple cipher-suites supported by each of the participants. Many standard cryptographic protocols, e.g., IKE, SSH, SSL and TLS, [RFC2409, RFC4253, RFC6101, RFC5246] use cipher-suite negotiation to ensure that the parties select the 'best' cipher-suite among those they jointly support, in order to avoid broken algorithms and to facilitate migration to better (more secure, more efficient) algorithms.

Currently, DNSSEC is an exception: it allows the use of multiple signature algorithms and hash functions, e.g., RSA and elliptic curves (see [10] for a complete list of cryptographic algorithms), however, no mechanism was designed to enable parties to select the strongest option that they support. As a result, during a DNS transaction between a resolver and a name server, *all* the keys and signatures, supported by the target zone, have to be exchanged. Namely, the name server always returns all these values to the resolver, even if some of these algorithms are unsupported or unvalidated by the resolver.

The lack of cipher-suite negotiation in DNSSEC implies larger DNSSEC-signed DNS responses, due to inclusion of multiple keys and signatures; Figure 1 plots the responses' sizes from Top Level Domains (TLDs) and top 1 million domains according to Alexa `www.alexa.com`. The measurements show that the overhead of signed DNS responses is significant in comparison to plain DNS responses. For instance, non-existent domain (NXD) is a very common response, which often occurs due to a typo in a DNS query: the size of NXD responses without DNSSEC is less than 400 bytes while with DNSSEC 70% the responses exceed 1000 bytes and 10% are even larger than link Maximal Transmission Unit (MTU) (which also holds for more than 30% of DNSKEY responses). Signed responses for `ANY` query type can reach even 5000 bytes and more, while plain `ANY` type responses are less than 1000 bytes.

This huge overhead is twofold and has implications both on computation and communication complexity.

Communication overhead. Large responses cause noticeable overhead: (1) on the name server that is required to transmit them,(2) on the recepient that needs to receieve and process them (allocate buffers to store and reassemble) and (3) on the network, and intermediate devices, e.g., routers and proxies, causing load spikes and increasing processing. Indeed, recent measurements, e.g., [14], show that a significant increase in the DNS infrastructure, both in resolvers and name servers platforms, is required in order

*Part of this research was conducted while the first author was visiting Technische Universtät Darmstadt and CISPA, Saarland University.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ACSAC '14 Dec. 8-12, 2014, New Orleans, LA, USA
Copyright 2014 ACM 978-1-4503-3005-3/14/12 ...$15.00.
http://dx.doi.org/10.1145/2664243.2664283.

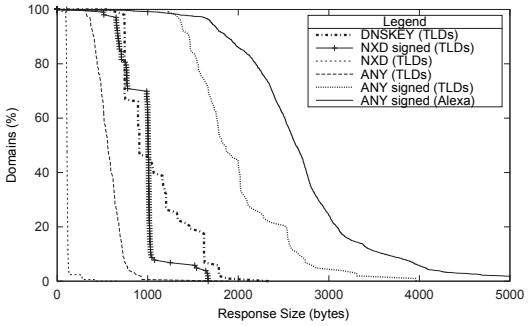


Figure 1: Length of responses for signed and non-signed Alexa and TLDs, for `ANY`, `DNSKEY` and `A` RRs.

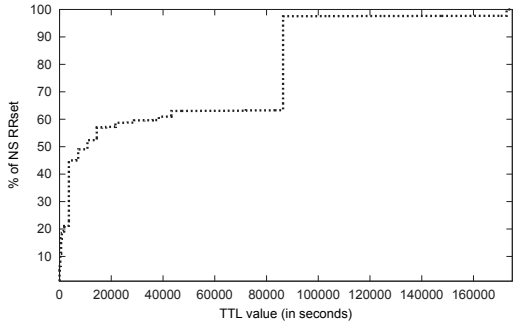


Figure 2: Distribution of the time-to-live (TTL) values in `DNSKEY` records in TLDs and Alexa domains.

to migrate to DNSSEC. Not only much more data, than needed, is sent: the name server sends all the keys and signatures that correspond to the cryptographic options that the zone supports, but also such large responses often result in fragmentation or cause resolvers to use TCP for the DNS transaction. The communication overhead also depends on the frequency at which the resolvers request the records from signed zones; the frequency is dictated by the time-to-live (TTL) value, set by the zone operators, and the resolvers cache the records until the TTL value expires. In Figure 2 illustrates the TTL values of name servers records from TLDs and Alexa domains: 60% of the TTL values are less than a day (86400 seconds) and almost 50% are less than one hour. Furthermore, these large responses, when causing fragmentation, may even expose to DoS and DNS-poisoning attacks, see [4]; more details in Section 2.

Computation overhead. Signatures generation increases the computational overhead on name servers and is significant for dynamic zones, e.g., the common content distribution networks. Signatures verification imposes overhead on busy resolvers and is prohibitive for mobile devices, making support of DNSSEC on end-hosts impractical.

We believe that these problems with large responses, and the lack of cipher-suite negotiation mechanism, motivate administrators to use only a limited number of cipher-suites. In particular, without cipher-suite negotiation mechanism administrators are likely to avoid algorithms with longer keys/signatures, e.g., 2048 or 4096 bit RSA; these may offer better security, but surely will increase response length significantly and as a result also exacerbate the interoperability problems and exposure to attacks. Paradoxically, for a different motivation, administrators are also likely to avoid offering alternative algorithms which may offer good security even with shorter keys/signatures, e.g., elliptic curves. Without ciphersuite negotiation, there is no motivateion to offer such shorter alternatives, since when sent in addition to the existing keys/signatures, e.g., 1024-bit RSA (which is mandatory to support), the resulting length of the response is even *larger*.

With the growing awareness of the existence of powerful, nation-state adversaries, who may be able to crack such key length, we find it alarming that 1024-bit RSA implementations are still dominant. Furthermore, 1024-bit RSA is *entrenched*, by being the default algorithm for DNSSEC, as mandated by NIST and IETF standards [RFC4033-4035], and further by the lack of cipher-suite negotiation in DNSSEC.

Indeed, as Figure 3 shows, the current adoption of DNSSEC algorithms by signed domains appears to support our conclusions: the 1024-bit RSA algorithm, already considered not-sufficiently-secure, is, by far, the most popular, while the elliptic curve (EC) variant, in spite of its shorter keys and signatures, and although it was standardised [RFC6605], is still not deployed.

This lack of adoption of non-default algorithms by name servers, also demotivates *resolvers* from supporting such algorithms, e.g., elliptic curve. This creates a vicious cycle, as zones are further de-motivated from using non-default signature algorithms, which are likely to be compatible only with few resolvers. Elliptic curve signatures (e.g., EC-DSA) also have some significant drawbacks, including long verification time and concerns for potential trapdoors; further research may identify secure signature mechanisms that are more suitable for this challenge.

We conclude that cipher-suite negotiation for DNSSEC is essential to security, interoperability and efficiency of DNS and DNSSEC. Furthermore, it may allow end-to-end cryptographic protection of DNS, in particular, in mobile devices.

Downgrade attacks and existing ciphersuite negotiation mechanisms. In a downgrade attack, a MitM attacker tries to trick a client and server into using a weak cryptographic mechanism, although both the client and the server support and prefer a stronger cryptographic mechanism. For example, such attack is known against version 2 of SSL, which supports cipher-suite negotiation, but insecurely, see, e.g., [16, 11]. Existing standardised cipher-suite negotiation mechanisms, were all designed for authenticated key-exchange protocols, e.g., IKE and TLS. These mechanisms are *interactive*, i.e., operate in two rounds. In the first round, the client sends its list of cipher-suites, typically in order of preferences, and the server responds by identifying the chosen cipher-suite (from the client's list). The second round uses the authenticated key exchanged by the protocol, to validate that, earlier, the server received the correct cipher-suite (i.e., the one sent by the client). Authentication is done by exchanging the Message-Authentication-Code (MAC) computed over the cipher-suites sent by the client (and other exchanged values, including random nonces to prevent replay). However, such a two-round, interactive mechanism, would add too much overhead to a DNS transaction and is not suitable for DNS, which is a pure request-response protocol. In contrast, our proposed secure cipher-suite negotiation protocol is *non-interactive*; it only adds few, short fields, to the messages exchanged during a

DNS transaction. This is important, since adding another round would cause significant extra delay, as well as make deployment much harder.

Note that our single-round, 'piggybacked' mechanism is possible since in DNSSEC, the *entire* set of supported public keys and algorithms, is signed ('certified') *together*; this holds for both *Key Signing Keys* and *Zone Signing Keys*. This is in contrast to typical PKI certificates, as used by both IKE and TLS, where the same entity may have multiple keys, algorithms and certificates, and has no signature over its entire set of keys, algorithms and certificates.

Our proposed DNSSEC cipher-suite negotiation mechanism also has a security advantage, cf. to the negotiation mechanisms previously specified, e.g., for TLS. Namely, known negotiation mechanisms are vulnerable, if the attacker can expose the cryptographic keys *in real time*, allowing the attacker to compute the MAC and thereby trick the parties into believing that cipher-suite negotiation was done correctly. Such an attack was presented against TLS [13].

In this paper, we focus on *host-by-host* (transport layer) cipher-suite negotiation for DNSSEC. An alternative, which we present in [7], is an end-to-end approach, whereby the signaling is in application layer. In the design in [7] the signaling is performed by concatenating a list of DNSSEC algorithms as subdomains, to the query; the preferences for ciphers are expressed via the order of concatenation after the query. The obvious disadvantage of the mechanism in [7] is, therefore, that any possible difference in the order of preferences, results in request to different record. On the other hand, the mechanism in [7] supports well the case of a 'forwarding resolver' which provides resolution services to several 'proxy resolvers', which may each have different cipher-suite policy (order of preference). We compare between the approaches in [9].

Contributions

In this work we carry out a study of DNSSEC deployment status and challenges and design a transport layer cipher-suite negotiation for DNSSEC. We implemented our proposed design as user-space modules, for resolver and name server, and evaluated it with standard Bind-9.9.3 DNS software.

The deployment of cipher-suite for DNSSEC would alleviate deployment obstacles, and would speed up adoption of DNSSEC both by resolvers and zones.

Organisation

In Section 2 we discuss deployment status of DNSSEC, review factors impeding deployment and argue that the main ostacle towards deployment is large DNSSEC-enabled responses. We also show that signed domains currently mainly support weak ciphers. We then provide a design of cipher-suite negotiation for DNSSEC in Section 3, and describe its implementation and evaluation. In Section 4, we provide an overview of DNS. We conclude this work and discuss future research in Section 5.

2. INTEROPERABILITY CHALLENGES AND VULNERABILITIES

Zones that adopt DNSSEC typically support a number of cryptographic keys, which results in large DNSSEC enabled responses; however, as we next show, mostly weak cryptographic options are supported. Signed DNS responses may often cause problems to resolvers, name servers and Internet clients and services. In particular, due to their increased size, signed responses are often exploited for attacks, e.g., cache poisoning and DDoS, or may disrupt DNS functionality and availability. As a result, zone operators are often hesitant about signing their zones, and resolvers' operators typically do not enforce validation. The outcome is impeded deployment of DNSSEC. Indeed, most DNS resolvers do not support DNSSEC validation, and many zones are not signed with DNSSEC. The fact that there is currently limited deployment of DNSSEC further reduces a motivation for early adopters, since protection of DNSSEC only 'kicks in' when all the entities, involved in a resolution of some domain, support DNSSEC.

DNSSEC Validation at Resolvers. A significant fraction of the resolvers currently signal DNSSEC support; however, *less than 3%* actually *enforce DNSSEC validation* [12]. Obviously, for such non-validating resolvers, DNSSEC does not provide added security.

In fact, non-validating resolvers are open to cache-poisoning attacks, in particular attacks exploiting the fact that such resolvers receive large DNSSEC responses (but do not validate them) [4].

DNSSEC Deployment at Zones. To make DNSSEC validation effective in resolvers the zones have to adopt DNSSEC. However, most do not. Recently, the root and some important top-level domains, such as `com, net` and `org`, were signed; through experimental study we found that currently 62% of the TLDs are signed, however, less than one percent (0.46%) of top million Alexa domains are signed. We summarise the deployment status in Table 1. The first column, Table 1, lists the domains in the forward and reverse DNS trees that we tested. The second column contains the number of registered domains that were tested. This is mainly relevant in the reverse DNS tree, where a large fraction of the domains (that correspond to IPv4 address blocks) are not registered; in those cases our queries were answered with *non-existing domain* responses.

The third column contains the number of name servers in each domain space (rounded to thousands, i.e., K $\equiv$ 1000). Then, in subsequent columns, we report on our measurement of the number of signed domains. In forward DNS, we found that 62% of the Top-Level Domains (TLDs) are signed and less than one percent (0.46%) of top million Alexa domains are signed. Even worse results were found by NIST, [15], who checked DNSSEC adoption among 1070 large industrial and financial US domains; out of those, they found only 1.4% to have adopted DNSSEC.

In the reverse DNS tree, we found even lower percents: only 0.07% of the zones are signed. We found this surprising, since the reverse DNS is used by several common security mechanisms. Note that DNSSEC validation requires a 'chain of trust', hence, adoption is meaningful only when all the zones in the path from the root are properly signed.

In what follows we outline the impact of large responses on stability and availability of DNS and Internet networks and services. As we argue, concerns for attacks and interoperability problems are among the main obstacles impeding wide adoption of DNSSEC.

domain	registered	name servers	signed	failure w/DNSSEC
Forward DNS Alexa	$50K$	32.5K	0.46%	23%
Forward DNS TLDs	568	3.2K	62%	2%
rDNS `x.in-addr.arpa.`	229	1.5K	84%	2%
rDNS `y.x.in-addr.arpa.`	$28K$	97K	0.4%	32%
rDNS `z.y.x.in-addr.arpa.`	$27,67K$	9,687K	0.06%	34.5%

Table 1: DNSSEC deployment and failures among domains in forward and reverse DNS trees.

2.1 Interoperability Problems with DNSSEC

DNSSEC enabled DNS responses are significantly larger than the corresponding 'legacy' DNS responses since they contain, in addition to the 'regular' DNS records, also DNSSEC-related records, mainly for signatures and public verification keys. Such responses often exceed the 512B maximal DNS size specified in [RFC1035], requiring use of the EDNS [RFC6891] extension mechanism; EDNS is also used to signal support of DNSSEC.

Some firewalls drop these DNS packets, due to the response length. Indeed, DNSSEC responses often exceed the maximal transmission unit (MTU) on the path, and thus may get dropped or fragmented. Fragments are also blocked by some firewalls, mainly for security concerns.

Due to concerns for the interoperability issues with large signed-DNS responses, many resolvers that support DNSSEC, accept and cache unvalidated responses, thereby exposing themselves to a *downgrade attack*, [RFC4035]; for instance, to avoid interoperability problems, `unbound` resolver supports a 'permissive' mode, accepting responses with missing or incorrect signatures.

In a downgrade attack an attacker sends fake responses that appear similar to responses passing through non interoperable devices. If DNS passes the forged responses to the application, then the attacker successfully performed a *downgrade attack* on the system, and poisoned the cache of the resolver, causing it to accept a forged response. For instance, consider an attacker that poisons the NS and glue A RRs, and provides a forged delegation NS RR to redirect the clients to a host of attacker's choosing. The resolver will simply revert to plain DNS without the DNSSEC protection - and will accept the spoofed records.

2.2 Deployment of Weak Cryptography

In our recent study, [7], where measured the DNSSEC algorithms, as well as the key sizes, supported by signed zones (among Alexa top million domains and TLDs), and found that most support different variations of RSA (with SHA1 or SHA256 hash functions) with weak keys, 1024 bits. The results are plotted in Figure 3. The categories are grouped according to key sizes[1] ranges and according to different variations of the RSA cipher; since all the domains were found to support different versions of RSA algorithm, we do not mention 'RSA' in the figure. The measurements are consistent with the recommendations by NIST and IETF [RFC4641] for a mandatory support of RSA and also to avoid using large keys (specifying a range of 512-2048 bits for (ZSK) key size and recommending a default value of 1024 bits), in order to avoid fragmentation, communication and computation overhead and other problems with large keys and signatures. There is hardly any support of cryptographic algorithms that produce short signatures, such as ECC, since the motivation to add more overhead to the transmitted data is low. This supports our argument that cipher-suite negotiation mechanism is essential not only for deployment of stronger cryptography and adoption of new cryptographic ciphers, which currently the zone operators are hesitant to support, but also to mitigate the interoperability and overhead problems associated with DNSSEC.

[1]The transmitted keys and signatures are encoded in Base64 thus contain an extra byte of redundancy for every three bytes of actual data.

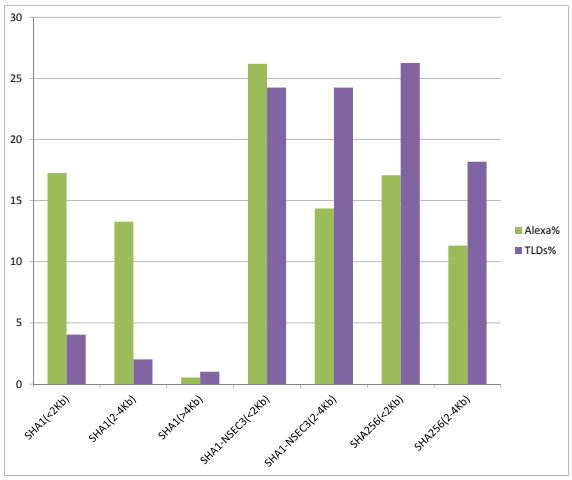


Figure 3: Distribution of the ciphers and public key sizes supported by TLDs and Alexa top million domains.

2.3 Attacks on (Oversized) DNSSEC Responses

We discuss two potential abuses of 'oversized', large signed-DNS responses: for amplification Denial-of-Service (DoS) attacks, and to manipulate responses, in particular, for cache-poisoning. We argue that part of the problem is partially due to the design of DNSSEC, which results in transmission of potentially unnecessary keys and signatures, especially if domains decide to support multiple keys and algorithms (as is highly desirable), and we propose possible (long-term) solutions.

We note that oversized DNSSEC responses can be exploited mainly when served over UDP, hence one simple countermeasure is to only use TCP for long responses. Indeed, this is done by some large servers, most notably of the `com` and `net` top-level domains. Notice, however, that

the use of TCP increases overhead on both network and server, for all responses - which can be significant. Furthermore, [12, 8] recently showed that clients that retrieve DNS records over TCP have high failure rates. We hope future research will result in UDP-based solutions, avoiding this overhead.

Amplification Attacks. The size of DNSSEC responses significantly exceeds the size of traditional DNS responses. Such responses are often abused by attackers to launch *amplification DoS attacks* [17] to clog victim networks and hosts.

In a DNS amplification DoS attack, the attacker sends to one or more DNS servers many requests, with spoofed (fake) source IP address of the victim. Name servers respond to such requests by sending (much larger) responses to the IP address that originated the DNS request. The *amplification factor* is the ratio between the number of response bytes sent by the *amplifying* (benign) DNS server, to the number of request bytes sent by hosts controlled by the attacker, in the corresponding requests. With DNSSEC, the ratio can be as high as a hundred. Indeed, while DNSSEC deployment is still very limited, it has already been abused in the largest DoS attacks in the recent years, with reported bandwidths of 100Gbps in 2010, 60Gbps in 2011 and 2012 (Arbor Networks Report) and 300Gbps (launched in 2013 against Spamhaus and Cloudflare).

Sometimes, the victim of the DNS amplification attack is the amplifying DNS server itself, which wastes computation and network resources on sending the (amplified) response packets; however, more often, the victim is a network or host, which receives the responses. DNS servers use various mechanisms, to prevent their abuse as vectors in amplification attacks, in particular, to avoid wasting their own resources due to such attacks. In fact, the concern for amplification attacks, may be one of the considerations against adoption of DNSSEC by zones.

Defences for name servers against their abuse as amplifiers include:

- Imposing a maximal *quota* of queries from the same source IP address. This defence may help against attacks against a remote host or network, using the same (or related) IP addresses, but not against attacks on the amplifying server itself, where the source IP address is not important. Furthermore, this defense fails if there are many spoofed IP addresses, connected via the same victim network, as in the Coremelt attack [20]. Finally, DNS request quotas are prone to false-positives, i.e., identification of a benign sending resolver as an attacker.

- Challenge-response mechanisms, mostly based on redirection of the request to another domain name, where the new domain name includes a random challenge ('nonce') used to ensure that the request did not contain a spoofed IP address. However, existing designs, e.g., [2, 21, 6], either do not support DNSSEC validation or are not widely deployed.

- As mentioned above, some servers, most notably those serving the large top-level domains `com` and `net`, defend against this threat by sending long responses over TCP rather than over UDP.

Attacks on Fragmented Responses. DNSSEC responses are often large enough, that they can exceed the MTU of networks in the path from the name server to the resolver (see size measurements in Figure 1) and hence, get *fragmented* by the name server or by a router along the path.

Ironically, an attacker can exploit fragmented DNS responses for IP defragmentation cache-poisoning: by sending spoofed fragments, the attacker can trick the defragmentation mechanism into reassembling them together with the fragments sent by the legitimate source (name server).

IP defragmentation cache-poisoning can be applied for DoS attacks, disrupting resolver to name server communication, for name server pinning, or even for DNS cache-poisoning, [4, 19]. (Note that other DNS poisoning techniques are known, e.g., [3, 5].) Furthermore, due to the currently-common case of permissive validation, the resolvers will accept and cache the altered (reassembled) DNS responses.

3. HOP-BY-HOP CIPHER-SUITE NEGOTIATION FOR DNSSEC

In this section we present the design for a cipher-suite negotiation mechanism for DNSSEC. According to our design, resolvers signal to name servers the list of ciphers that they support and the name servers select the strongest cipher from the list and return only the keys and signatures that correspond to the selected option. Recently, [RFC6975] standardised new options in EDNS0 enabling clients to signal the supported ciphers to the name servers. The signaling of deployed ciphers allows zone operators to determine whether they can adopt new cryptographic algorithms. However, the zones would still have to support the other algorithms, to serve resolvers which have not adopted the new cipher.

We design cipher-suite negotiation mechanism, that utilises the signaling of the supported ciphers to produce a full fledged cipher-suite negotiation mechanism. Our design does not require changes to the DNS software or protocol, and can be easily integrated into the existing DNS infrastructure. We implemented our cipher-suite negotiation mechanism as two separate user space modules, *cipher-suite client* and *cipher-suite server* (Section 3.3). The modules, along with a DNS transaction supporting cipher-suite negotiation, are illustrated in Figure 4. The modules are configured to capture all DNS requests and responses exchanged between the resolver and the name server, and to perform cipher-suite negotiation between the resolver and the name server. The outcome is that the server only returns the cryptographic material that corresponds to the strongest cipher that was negotiated.

Furthermore, our design uses the DNS packets themselves for signaling of the supported cryptographic options and thus does not add an additional round trip of communication. According to our design, the resolvers signal the cryptographic options via an EDNS0 OPT resource record, [RFC6891], Section 3.1. EDNS0 is already supported by the resolvers and name servers and is a prerequisite requirement for adoption of DNSSEC. EDNS0 was designed to allow adoption of new mechanisms, such as DNSSEC, by utilising existing EDNS fields; see Section 4 for background.

Our mechanism requires a modification of the current signature procedure `dnssec-keygen` which currently signs each DNS record in a zone file which *all* the supported keys. We show how to extend it to enable cipher-suite negotiation, in

Section 3.2.

To test our implementation, we extended the `dig` (domain information groper) DNS lookup utility[2], with support for signaling of a list of ciphers.

To integrate support of cipher suite also into the process of signing the zone file - instead of signing the same zone file with multiple ciphers, we create separate copies of the zone file, such that each copy corresponds to a single cipher.

We next describe the cipher-suite mechanism, the implementation of the prototype and its evaluation.

3.1 Extending EDNS0 with New Cipher-Suite Options

EDNS0, [RFC6891], was designed to provide extra space for new flags and codes. We propose a new EDNS0 Extension Mechanism to negotiate a DNSSEC cipher option for a transaction between a resolver and a name server. Since EDNS0 is compatible with legacy servers that do not support new options, our cipher-suite negotiation mechanism is also interoperable with existing resolvers and name servers.

To signal support of cipher-suite negotiation, we introduce a new (cipher-suite ok) `CO` bit in the extended flags of the EDNS0 OPT record, see Figure 5. Resolver requests to perform cipher-suite negotiation by setting the `CO` bit in the EDNS0 OPT record, in the DNS request. We also reserve two new record types: `ciphers` and `cipher`. When signaling the `CO` bit, the resolver also places in the EDNS0 a list of cryptographic ciphers, that it supports, in the `ciphers` field, to convey the ciphers to the name server. A name server that supports cipher-suite negotiation mechanism responds with the `CO` bit set, and also places the selected cipher, from the ciphers list, into the `cipher` field in the EDNS0 record of the DNS response. The name server also places in `ciphers` field a list of cryptographic algorithms which the zone (for which it is authoritative) supports, along with a digital signature, stored in `RRSIG` record, over the server's ciphers list, to prevent downgrade attacks.

The algorithms conveyed in records `ciphers` and `cipher` are encoded as numbers. Each algorithm in DNSSEC contains a code number that corresponds to a DNSSEC public key (stored in DNSKEY record). The Delegation Signer (DS) and Hashed Authenticated Denial of Existence (NSEC3) use digests (hashed values as part of their resource record data). These hash algorithms also have code numbers. These codes signal to validating resolvers which cryptographic algorithm was used to generate the digital signature and the hash digest.

Next to each cipher number, we also add a priority number in ascending order (highest priority is 1). In order to ensure a deterministic outcome from the ciphers negotiation process, the cipher-suite client must assign different priorities to each ciphers that it supports (and cannot assign the same priority to two distinct ciphers); e.g., in case a number of ciphers are considered equally secure by the resolver or by the target zone.

3.2 Splitting the Zone File

When supporting a number of cryptographic options, zone operators sign, typically using `dnssec-keygen` procedure, the same zone file with a number of algorithms. The signing process of DNSSEC creates a signature per each RR set, i.e., set of resource records of the same type; for instance, all the name servers (NS) records (of a specific domain) are signed as a single unit, and for each supported cryptographic option a signature is attached to NS RR set. When a number of cryptographic options are supported (as is typically the case) a number of signatures are created and attached to the RR set. The same applies to other DNS records, such as DNSKEY, i.e., all the DNSKEY RR set is signed with all the supported algorithmic options. As a result, DNS responses typically contain a number of signatures (that correspond to all supported cryptographic ciphers) for each RR set that they return.

Similarly to [7], to encorporate support for cipher-suite negotiation, we extended the `dnssec-keygen` procedure, to generate a separate zone file, per each cryptographic cipher that the zone supports. Our `ednssec-keygen` (extended `dnssec-keygen`) performs the same number of signing operations as would have been performed by `dnssec-keygen`. However, instead of signing the same zone file with different

```
ednssec-keygen(ciphers, zone_file){
  signed_zone_file1 = dnssec-keygen(ciphers[1], zone_file)
  signed_zone_file2 = dnssec-keygen(ciphers[2], zone_file)
  signed_zone_file3 = dnssec-keygen(ciphers[3], zone_file)
  return (for i to ciphers: return signed_zone_file_i)
  }
```

keys (which results in a zone file where every RR set is coupled with a set of signatures per each supported cipher), it creates a separate copy of the zone file for each supported cryptographic cipher. Each resulting RR set in a signed zone file contains the signature that corresponds only to the cipher which was used to sign that copy of the zone file.

3.3 Cipher-Suite Negotiation Protocol

The steps of the protocol are illustrated in Figure 4. In step 2, cipher-suite client receives a DNS request (sent by the resolver in step 1). Cipher-suite client adds a list of supported ciphers to the `ciphers` field, of the EDNS0 record (in the `additional` section of a DNS request). Upon receipt of a DNS request, in step 3, with the `CO` bit set, cipher-suite server performs cipher-suite negotiation according to the preferences specified by the resolver and those supported by the zone, and forwards the request to a zone file that corresponds to the selected cipher. The name server retrieves the DNS records from the correct zone file and sends a response, step 5. Upon receipt of a response from the name server, cipher-suite server attaches the list of `ciphers` which the name server supports, along with a signature over that list and specifies the selected cipher in `cipher` field, then returns the request to the resolver, step 6. When cipher-suite client receives a response, it checks that the selected cipher is correct (according to the options supported by the resolver and the zone), validates the signature over the `ciphers` field, and forwards the response to the resolver, step 7. If one of the validation steps fails, the response is discarded.

Cipher-Suite Client. Cipher-suite client adds signaling of cipher-suite negotiation along with a list of supported ciphers, to DNS requests, and validates that the DNS records in the responses are signed with the correctly negotiated cipher.

Handling DNS Requests. Cipher-suite client adds to EDNS0 record the `ciphers` field with a list of ciphers that it supports and sets the `CO` bit.

Handling DNS Responses. Upon arrival of a DNS re-

[2] `dig` is a tool for performing DNS lookup by interrogating DNS name servers, and displaying the responses.

sponse, cipher-suite client extracts from the EDNS0 record the content of `ciphers`, and checks that the value in `cipher` record contains an intersection between the ciphers supported by the server and the client. If so, it proceeds to validating the signature over the `ciphers` field transmitted in `RRSIG` record. For signature validation the cipher-suite client uses the key signing key (KSK) of the target zone; the KSK of the target zone can be obtained from the parent zone or by alternative means, e.g., via email or public key repository such as DLV, [RFC5074]. If the signature is valid, the cipher suite negotiation is successful. The resolver can then proceed to DNSSEC validation of the received DNS records with the negotiated cipher.

Cipher-Suite Server. Cipher-suite server runs on the name server side. It obtains a list of all the ciphers that the zone supports, signed with the KSK of the zone; signing with a KSK is essential to prevent downgrade attacks (details within). This list of ciphers is placed into the `ciphers` field in EDNS0 record of the DNS responses sent to resolvers that signal support of cipher-suite negotiation in their requests.

Handling DNS Requests. Upon receipt of a DNS request with the `CO` bit set, cipher-suite server extracts the ciphers list from `ciphers` record, signaled by the cipher-suite client. It then selects a cipher based on the ciphers list signaled by cipher-suite client and the list of ciphers supported by the target zone, while also applying the respective priorities (specified by the cipher-suite client and the zone). Once the negotiated cipher is established, cipher-suite server identifies the zone file on the name server that corresponds to the selected cipher, from which the DNS records are retrieved.

Handling DNS Responses. Cipher-suite server then adds to the `ciphers` field in EDNS0 record of the response from the name server a signed list of ciphers supported by the zone, and adds the selected cipher to the `cipher` field.

The cipher-suite client uses the signed list of ciphers from the name server to validate that the cipher-suite negotiation was correct and that the ciphers list, that it sent in the DNS request, was not altered by an attacker, e.g., to cause a downgrade attack.

3.4 Resilience to Downgrade Attacks

Our cipher-suite negotiation is resilient to downgrade attacks. In particular, any attempt by the attacker to cause the communicating parties to use a weaker cipher version than the optimal one supported by them both, will be detected. In order to downgrade the defence to a weaker cipher the attacker can tamper with one of the following exchanges:

Request to Target Zone. Attacker can tamper with the request of the resolver, and remove the strong ciphers from the `ciphers` field. The name server will not detect the attack since the request is not signed, and will send the response from the zone file that corresponds to the cipher selected by the attacker. However, the DNS response contains a signed list of ciphers supported by the zone, which allows the resolver to calculate the cipher which should have been selected by the name server. Since the outcome of the cipher selection process is deterministic, computation of the optimal cipher by the cipher-suite client, at the resolver, should yield the same outcome as the computation performed by the cipher-suite server, running on the name server.

Response from Target Zone. If the attacker tampers with the response, sent by the name server, and removes ciphers from the `ciphers` list of the name server, signature verification by cipher-suite client will fail, and tampering will be detected.

Downgrade of Chain-of-Trust. In order to authenticate the public verification (ZSK) keys of the target zone, DNSSEC validating resolver establishes a chain-of-trust from the root to the target zone. Attacker can tamper with the referral response from the parent zone, by removing all the DS records that correspond to strong ciphers supported by the target (child) zone. However, notice that the signature of the parent is over *all* the DS records, hashed public verification (ZSK) keys, of the child zone. Furthermore, the ZSK key is subsequently used to sign the `ciphers` list, which allows the resolver to authenticate the keys and to detect tampering during computation of optimal cipher.

3.5 Implementation and Evaluation

In this section we describe the implementation of cipher-suite client and cipher-suite server modules. We configured an `iptables` firewall rule to capture all packets to/from port 53 (DNS requests/responses) and to pass them to our cipher-suite client (on the resolver side) and cipher-suite server (on name server side) user space `C` modules for processing. In our implementation of the cipher-suite client and server we used the `libipq` module to retrieve the packets from the kernel queue for user space processing.

Cipher-suite client (resp. cipher-suite server) module can be placed on the same physical host as the DNS resolver (resp. DNS name server) or can be configured on a separate host, with a suitable firewall rule to redirect the DNS traffic via the cipher-suite client (resp. cipher-suite server).

Cipher-suite server can be configured to support online and offline DNSSEC signing. According to the original design, DNSSEC required *offline signing*, i.e., the zone file is signed on a separate host, and then it is distributed to all name servers via a zone update. However, this setting was not suitable for many domains, e.g., especially dynamic domains, where the zone file is updated frequently. To accomodate for those scenarios an *online signing* was standardised, [RFC4470, RFC4471]. Online signing is more flexible and is already integrated into a number of DNS proxies, both proprietary and free, e.g., Secure64, PowerDNS, and an increasing number of signed zones adopt an online signing procedure.

Our cipher-suite server can be integrated into an online signing proxy, without requiring any changes to the DNS software or to zone files. The configuration only requires to apply the processing of cipher-suite server to requests, prior to passing them to the online signing proxy, and then on responses from the proxy.

To be able to identify which cipher corresponds to the response returned by the name server, the proxy should maintain state, between the arrival of the request, and until a response is served by the name server. The state is composed of query, TXID, source IP address, and the selected cipher, from the list sent by the client.

In an offline signing, the name server has to identify the correct zone file that corresponds to the optimal negotiated cipher.

Extended DNS Lookup Utility. To evaluate cipher-suite negotiation, in addition to implementation in resolvers and name servers, we also extended `dig` DNS lookup utility, to support cipher-suite negotiation mechanism, and to set the required cipher-suite negotiation options in EDNS0; see

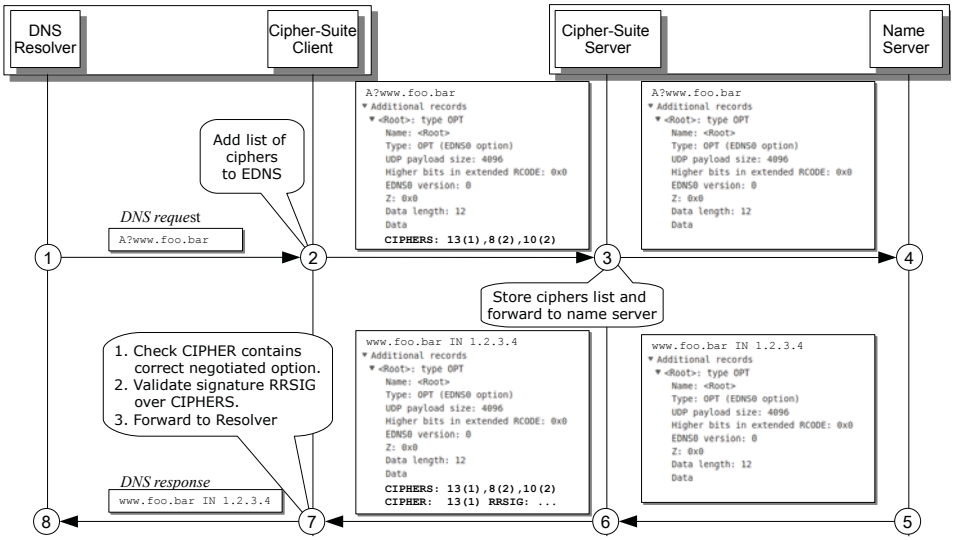


Figure 4: Cipher-suite negotiation for DNSSEC.

screen capture in Figure 5.

Evaluation. We evaluated the communication overhead for signed DNS responses with transport layer cipher suite negotiation, and without. The evaluation compared the amount of bytes transmitted, Figure 6.

For our evaluation we configured cipher-suite client and cipher-suite server as modules on separate hosts, such that cipher-suite client was set up on the resolver's network and cipher-suite server on name server's network. We added firewall rules to redirect the DNS traffic via the modules.

We wrote a script that was stripping all the keys and signatures from responses of the name servers and was adding keys and signatures according to a cipher that we selected - simulating the cipher-suite selection process. Due to short keys and signatures size, we used EC cipher, and the keys and signatures were computed with an EC secret signing key that we generated for this experiment; the resolver was configured with a corresponding trust anchor, i.e., public verification key, which enabled it to establish a chain of trust and to successfully verify signatures on responses. The DNS response with the new keys and signatures was then passed to cipher-suite server, which added the signed ciphers list, set the `CO` bit on, and specified the selected cipher (EC) in `cipher` field.

Cipher-suite client module captured the requests and responses from the resolver, added cipher-suite negotiation options to EDNS0, and then validated cipher-suite negotiation in responses (using the trust anchor that we configured in it especially for this experiment). We performed two series of evaluations of requests' latency over a list of Alexa top sites and TLDs: (1) we wrote a python script that receives in an input a list of TLDs and Alexa top domains, invokes the `dig` utility on each domain, and measures the time between transmission of request and receipt of the response as well as the number of bytes received in the response; we used Bind-9.9.3 software to set up a recursive DNS resolver. (2) Then, to measure the impact of cipher-suite negotiation, we wrote a python script, which used the extended `dig` tool with our cipher-suite negotiation options for sending DNS requests.

```
dig A .www.foo.bar +dnssec +ciphers=RSASHA512(1), ECDSAP256SHA256(1), RSASHA1(2), RSAMD5(3)

; <<>> DiG 9.7.3 <<>> A www.foo.bar +dnssec +ciphers=RSASHA512(1), ECDSAP256SHA256(1),
                                                   RSASHA1(2), RSAMD5(3)

;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 35927
;; flags: qr rd ra; QUERY: 1, ANSWER: 3, AUTHORITY: 7, ADDITIONAL: 1

;; OPT PSEUDOSECTION:
; EDNS: version: 0, flags: do co; udp: 4096
; CIPHER:  RSASHA512
; RRSIG: RRSIG      CIPHERS 8 1 21600 20140120183409 20140113183409 26790 foo.bar.
T5ifp0XpaOcjMS96iCSfsAfwyzBohgEz9cTsAcRBuEbdn87b+PU5yRvkQPrm8yCNXIZqgTn+92tGuv/s
ldoq11/Y70MlU40Wn0TBpB4XHgFM+rTxYLFtkXTlTFgQDKl5nCH+0gXIYm2hpiYSYRqFofcXrxlB
3LZUu7n qUgqUnA=
; CIPHERS: RSASHA512(1), RSAMD5(2)
```

Figure 5: Execution of `dig` enabled with our cipher-suite negotiation mechanism; the new options are underlined.

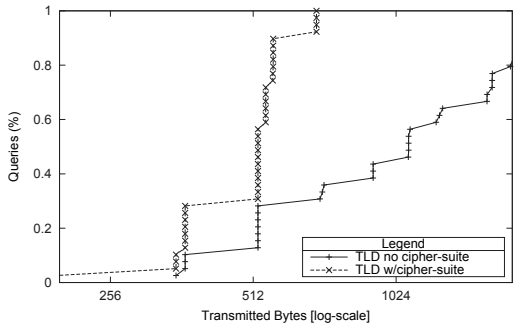


Figure 6: Amount of bytes transmitted in DNSKEY responses from TLDs, with and without support of cipher-suite negotiation.

We measured the impact of cipher-suite negotiation on the amount of the transmitted bytes with signed TLDs, using DNSKEY responses. The graph, plotted in Figure 6, with cipher-suite negotiation shows three steps. Intuitively,

one may expect to see an almost straight line, however, the reason for the steps is that some domains are in a process of key rollover for ZSK and thus use two ZSK (in addition to a KSK). The improvement in traffic is the difference between the two graphs: the upper (with cipher-suite negotiation) and the lower (without cipher-suite negotiation).

4. OVERVIEW: DNS AND DNSSEC

The domain name system (DNS), [RFC1034, RFC1035], is a distributed data base of Internet mappings (also called *resource records (RRs)*), from *domain names* to different values. The most popular and widely used mappings, [1], are for IP addresses, represented by `A` type RRs, that map a domain name to its IPv4 address, and name servers, represented by `NS` type RRs, that map a name server to domain name; see [RFC1035] for a list of standard DNS mappings.

The client side of the DNS infrastructure is composed of resolvers, which lookup records in zones by sending DNS requests to corresponding name servers. The resolvers communicate to the name servers using a simple request-response protocol (typically over UDP); for instance, (abstracting out subtleties) to translate `www.foo.bar` resolvers locate the name server `ns.foo.bar`, authoritative for `foo.bar`, and obtain the IP address of the machine hosting the web server of the website `www.foo.bar`, see Figure 7. Resolvers store DNS records, returned in responses, in their caches for the duration indicated in the Time To Live (TTL) field of each record set.

The resource records in DNS correspond to the different services run by the organisations and networks, e.g., hosts, servers, network blocks.

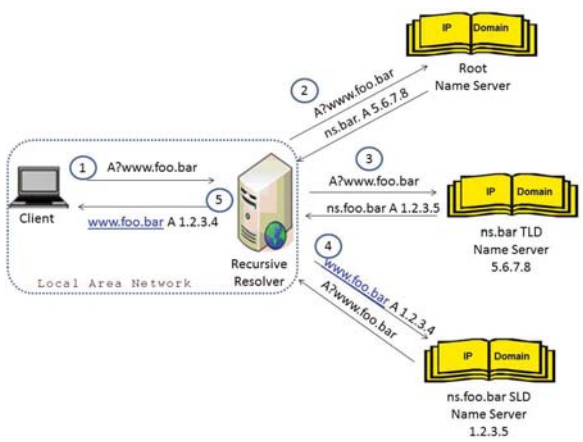


Figure 7: DNS resolution process for `www.foo.bar` and the involved DNS servers.

The zones are structured hierarchically, with the root zone at the first level, Top Level Domains (TLDs) at the second level, and millions of Second Level Domains (SLDs) at the third level. The IP addresses of the 13 root servers are provided via the *hints* file, or compiled into DNS resolvers software and when a resolver's cache is empty, every resolution process starts at the root. According to the query in the DNS request, the root name server redirects the resolver, via a `referral` response type, to a corresponding TLD, under which the requested resource is located. There are a number of TLDs types, most notably: *country code TLD* (ccTLD), which domains are (typically) assigned to countries, e.g., `us, il, de`, and *generic TLD* (gTLD), whose domains are used by organisations, e.g., `com, org`, and also US government and military, e.g., `gov, mil`. Domains in SLDs can also be used to further delegate subdomains to other entities, or can be directly managed by the organisations, e.g., as in the case of `ibm.com, google.com`.

A DNS *domain* is divided into zones, and includes all the nodes of the subtree rooted at the zone. A DNS *zone* constitutes a portion of a domain name space. A zone can be divided into subdomains, with its own DNS name servers. For instance, when querying the root zone for `foo.bar.`, the resolver will be redirected to `bar.` domain, via a `referral` to the authoritative servers for `bar.` zone. When querying the name servers of `bar.`, the resolver receives another `referral` for `foo.bar.` zone. Notice that `bar.` zone does not include subdomains, e.g., like `foo.bar.`, but those are delegated from `bar` to their name servers.

Domains and their mappings are also administered hierarchically; the mappings of each domain `foo.bar` are provided by a *name server*, managed by the owner of the domain.

5. CONCLUSIONS AND FUTURE RESEARCH

DNSSEC deployment has progressed slowly and although standardised more than 15 years ago, it is still not widely adopted. One of the problems is that early adopters do not obtain a security benefit, worse, they are exposed to interoperability problems and attacks. Currently, almost *all* (99.9% !) of DNSSEC signatures use RSA, with 1024 bit keys, which is already considered insecure, and the use of 2048-bit keys results in even larger DNS responses.

Signed zones do not deploy stronger ciphers due to concerns for compatibility with clients. Of course, domains could use both elliptic curve (or some other form of 'short-key-signatures') together with the existing RSA signatures. However, unfortunately, DNSSEC does not contain a 'cipher-suite negotiation' mechanism in contrast to most other IETF cryptographic standards, e.g., TLS [RFC2246]. Hence, when a domain uses multiple algorithms (or keys), it will send *all* the keys and all the signatures to the client - increasing, rather then decreasing, the length of the responses. The same problem may arise, when domains consider using larger RSA keys or a more secure algorithm, for improved security, or to change keys periodically. Hence, zones are likely to continue using the old keys and cipher, for a long time or forever, due to the concern of interoperability with resolvers.

All this implies obstacles to deployment of more appropriate cryptography, resulting in large DNSSEC responses, interoperability problems, abuse in reflection amplification DoS attack and weak security.

In this work, we reviewed the central security and deployment problems with the current DNSSEC design. Several of these stem from the same problem essentially: DNSSEC requires transmission of non-standard DNS packets, in particular, significantly larger than standard DNS packets (limited to 512 bytes). We outline the problems and propose a design for a transport layer cipher-suite negotiation mechanism, to alleviate them. Our cipher-suite negotiation allows domains to use multiple keys and algorithms while sending only these needed by the resolvers. We performed a study showing that our design would significantly reduce the overhead in communication to TLDs and top Alexa domains

by reducing the (unnecessary) DNS traffic, and by avoiding fragmentation and transition to TCP for large responses.

This work focused on negotiation of DNSSEC signature algorithms. There are several proposals for additional cryptographic defenses to DNS, which will provide confidentiality and privacy in addition to authenticity; see criticism and discussion in [18]. We believe that our approach can be extended to support also negotiation of the relevant confidentiality algorithms, e.g., encryption.

Acknowledgements

This research was supported by grant 1354/11 from the Israeli Science Foundation (ISF), by the Ministry of Science and Technology, Israel, by the German Federal Ministry of Education and Research (BMBF) within EC SPRIDE, by the Hessian LOEWE excellence initiative within CASED, and was partially supported by the TENACE PRIN Project (n. 20103P34XC) funded by the Italian Ministry of Education, University and Research.

6. REFERENCES

[1] H. Gao, V. Yegneswaran, Y. Chen, P. Porras, S. Ghosh, J. Jiang, and H. Duan. An empirical reexamination of global dns behavior. In *Proceedings of the ACM SIGCOMM 2013 conference on SIGCOMM*, pages 267–278. ACM, 2013.

[2] F. Guo, J. Chen, and T. cker Chiueh. Spoof Detection for Preventing DoS Attacks against DNS Servers. In *ICDCS*, pages 37–37. IEEE Computer Society, 2006.

[3] A. Herzberg and H. Shulman. Security of patched DNS. In *Computer Security - ESORICS 2012 - 17th European Symposium on Research in Computer Security, Pisa, Italy, September 10-12, 2012. Proceedings*, pages 271–288, 2012.

[4] A. Herzberg and H. Shulman. Fragmentation Considered Poisonous: or one-domain-to-rule-them-all.org. In *IEEE CNS 2013. The Conference on Communications and Network Security, Washington, D.C., U.S.* IEEE, 2013.

[5] A. Herzberg and H. Shulman. Vulnerable delegation of DNS resolution. In *Computer Security - ESORICS 2013 - 18th European Symposium on Research in Computer Security, Egham, UK, September 9-13, 2013. Proceedings*, pages 219–236, 2013.

[6] A. Herzberg and H. Shulman. DNS Authentication as a Service: Preventing Amplification Attacks. In *ACM Annual Computer Security Applications Conference (ACM ACSAC)*, December 2014.

[7] A. Herzberg and H. Shulman. Negotiating DNSSEC Algorithms Over Legacy Proxies. In *Cryptology and Network Security, 13th International Conference, (CANS) 2014, Heraklion, Crete, Greece, October 22-24, 2014. Proceedings*, 2014.

[8] A. Herzberg and H. Shulman. Retrofitting Security into Network Protocols: The Case of DNSSEC. *Internet Computing, IEEE*, 18(1):66–71, 2014.

[9] A. Herzberg and H. Shulman. Cipher-Suite Negotiation for DNSSEC: Hop-by-Hop or End-to-End? *(to appear) Internet Computing, IEEE*, 2015.

[10] Internet Assigned Numbers Authority (IANA). Domain Name System Security (DNSSEC) Algorithm Numbers. `http://www.iana.org/assignments/dns-sec-alg-numbers/dns-sec-alg-numbers.xhtml`, March 2014.

[11] T. Jager, K. G. Paterson, and J. Somorovsky. One bad apple: Backwards compatibility attacks on state-of-the-art cryptography. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2013.

[12] W. Lian, E. Rescorla, H. Shacham, and S. Savage. Measuring the Practical Impact of DNSSEC Deployment. In *Proceedings of USENIX Security*, 2013.

[13] N. Mavrogiannopoulos, F. Vercauteren, V. Velichkov, and B. Preneel. A cross-protocol attack on the TLS protocol. In T. Yu, G. Danezis, and V. D. Gligor, editors, *ACM Conference on Computer and Communications Security*, pages 62–72. ACM, 2012.

[14] D. Migault, C. Girard, and M. Laurent. A performance view on dnssec migration. In *Network and Service Management (CNSM), 2010 International Conference on*, pages 469–474. IEEE, 2010.

[15] National Institute of Standards and A. N. T. D. Technology. Estimating Industry IPv6 and DNSSEC External Service Deployment Status. http://fedv6-deployment.antd.nist.gov/cgi-bin/generate-com.

[16] E. Rescorla. *SSL and TLS: designing and building secure systems*, volume 1. Addison-Wesley Reading, 2001.

[17] C. Rossow. Amplification Hell: Revisiting Network Protocols for DDoS Abuse. In *Proceedings of the Network and Distributed System Security (NDSS) Symposium*, February 2014.

[18] H. Shulman. Pretty Bad Privacy: Pitfalls of DNS Encryption. In *Proceedings of the 13th annual ACM workshop on Privacy in the electronic society*, WPES '14, 2014.

[19] H. Shulman and M. Waidner. Fragmentation Considered Leaking: Port Inference for DNS Poisoning. In *Applied Cryptography and Network Security (ACNS), Lausanne, Switzerland*. Springer, 2014.

[20] A. Studer and A. Perrig. The Coremelt Attack. In M. Backes and P. Ning, editors, *ESORICS*, volume 5789 of *Lecture Notes in Computer Science*, pages 37–52. Springer, 2009.

[21] R. Tzakikario, D. Touitou, G. Pazi, et al. Dns anti-spoofing using udp, Nov. 17 2009. US Patent 7,620,733.

DNS Authentication as a Service: Preventing Amplification Attacks

Amir Herzberg*
Department of Computer Science
Bar-Ilan University
Ramat-Gan, Israel
amir.herzberg@gmail.com

Haya Shulman†
Fachbereich Informatik
Technische Universität Darmstadt
Darmstadt, Germany
haya.shulman@gmail.com

ABSTRACT

We present the first defence against DNS-amplification DoS attacks, which is compatible with the common DNS servers configurations and with the (important standard) DNSSEC. We show that the proposed DNS-authentication system is efficient, and effectively prevents DNS-based amplification DoS attacks abusing DNS name servers. We present a game-theoretic model and analysis, predicting a wide-spread adoption of our design, sufficient to reduce the threat of DNS amplification DoS attacks. To further reduce costs and provide additional defences for DNS servers, we show how to deploy our design as a cloud based service.

Keywords

Denial of service attacks, DNS Amplification, DNS Reflection, Source Authentication, DNS Authentication

1. INTRODUCTION

Amplification Denial of Service (DoS) attacks pose a major threat to the stability of the Internet, to organisations and to critical infrastructures. In the last decade the Internet has experienced a wave of Denial of Service (DoS) attacks threatening its welfare and stability. For example, consider the recent DoS attacks against Spamhaus (see [12]); Spamhaus contracted the Cloudflare services to resist the attack, to which the attackers responded with even more traffic, clogging the Internet Exchange Points (IXPs) serving Cloudflare (and many other customers).

The most prevalent type of such attacks is reflection via different application protocols, particularly those that provide amplification. Domain Name System (DNS) has a long history of abuse in amplification Denial of Service (DoS) attacks, to flood the victim networks and services, and it is one of the most popular amplifiers. A relatively small DNS request, e.g., up to 100 bytes, results in a much larger DNS response. We performed a study of typical DNS response sizes of Top Level Domains (TLDs) and top-million Alexa domains, `www.alexa.com`, signed and non-signed, in Figure 1; as can be seen, DNS responses signed with DNSSEC result in much larger packets, than the traditional (unsigned) responses, and even the 'non-existing domain (NXD)' responses that are signed, often exceed the maximal transmission unit (MTU). In the course of a reflection attack, the attacker sends multiple DNS requests (possibly from spoofed source IP addresses) to DNS servers, either via recursive resolvers or directly to the name servers, which incites much larger DNS answers in response. As a result, the victim is flooded with multiple (large) responses. These attacks have a detrimental impact on the bandwidth and performance of the DNS name servers and recursive resolvers, that are abused as a vector in amplification DoS attack, due to the excess processing of the numerous (malicious) requests and generation (or forwarding) of the significantly larger responses.

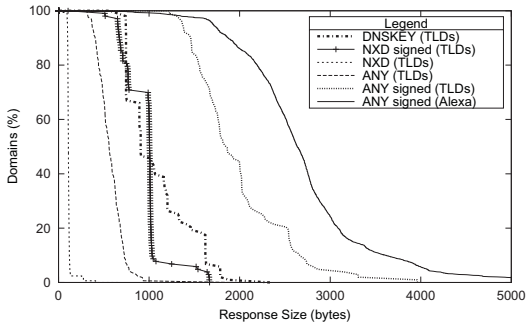


Figure 1: Length of responses for signed and non-signed Alexa and TLDs, for `ANY`, `DNSKEY` and `A` resource records; `A` records were sent for random subdomains of tested domains, and resulted in NXD responses.

In this work we focus on the defence mechanisms that the DNS servers[1] can deploy to foil attempts of the attackers to abuse them as a vector in amplification DoS attacks. Such

*Part of this research was conducted while the first author was visiting Technische Universtät Darmstadt and CISPA, Saarland University.
`sites.google.com/site/amirherzberg/`

†`sites.google.com/site/hayashulman/`

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ACSAC '14 Dec. 8-12, 2014, New Orleans, LA, USA
Copyright 2014 ACM 978-1-4503-3005-3/14/12 ...$15.00.
http://dx.doi.org/10.1145/2664243.2664281.

[1]In this work, we use the 'DNS server' to refer both to the

a defence would not only improve performance of the DNS servers and reduce the latency for their clients, but would also (indirectly) provide benefit to multiple Internet networks and services, by making it more difficult for attackers to find suitable amplifiers to flood the victims.

Rate limiting is a common defence used to protect the DNS servers from being abused in amplification DoS attacks, by limiting the number of requests that a name server or an open recursive resolver (such as Google Public DNS) would respond to; the additional requests, that exceed the imposed threshold, are discarded. However, rate limiting is not an effective defence and attackers typically circumvent it by distributing the attack among multiple DNS servers, sending a lower volume of requests to each, thus still achieving the required amplification against a victim network. The attackers also often distribute the attack between multiple compromised hosts that they control, and typically employ IP addresses' spoofing, to evade detection as well as to disguise the source of the attack making it difficult to block it.

Indeed, amplification DoS attacks are frequently launched by exploiting name servers and (typically open) resolvers.

Related Work

The challenge-response schemes, proposed in prior art, [4, 16], are designed to enable the name servers to filter the requests sent from spoofed IP addresses. Identification of spoofed requests typically relies upon a standard DNS functionality of the resolvers, and requires support for such filtering on the name server side. Upon a DNS request, the challenge-response authentication schemes on the name servers, [4, 16], send referral responses and encode a cookie via a `CNAME` or `NS` type DNS records. The cookie is the source IP address that appeared in the request. The resolver is then expected to echo the challenge in the subsequent DNS request (asking for the corresponding `CNAME` or `NS` records). The protection mechanism verifies that the source IP address in the second request is the same as the one appearing in the first request, and if so, sends another referral response, redirecting the resolver to the real name server (which the resolver eventually asks for the records that it originally needed to obtain).

However, none of the proposed schemes is deployed in practice and currently the name servers are patched to support rate limiting, i.e., reducing the number of requests which are answered, and ignoring sources which send too many requests. The attackers easily overcome that defence, by distributing the requests among multiple hosts, such that each host sends only a few requests.

We show that the challenge-response authentication via a random cookie does not constitute an effective defence against reflection amplification DoS attacks. In addition, these defences may often even 'disrupt' the DNS functionality for legitimate clients.

We next discuss the problems and limitations in the defences proposed in prior art, and argue that these pose a significant obstacle towards deployment thereof and make them ineffective for protection against attacks. In Section 2 we report on our measurements' results which show that the defences proposed in prior art are not suitable for common DNS servers configurations.

name servers and to recursive resolvers, since both are frequently abused as a vector in amplification DoS attacks.

Signed DNS Zones

The challenge-response mechanisms for source authentication, [4, 16], are incompatible with DNSSEC signed zones: when receiving a DNS request, the defence mechanism sends a freshly generated cookie (containing the source IP address of the requester) in the response. This requires signing the DNS record on-the-fly. However, most singed zones support offline signing, where the secret signing keys are not kept on the name server, hence it will not be able to produce a valid signature. As a result, DNSSEC-enabled resolvers would not be able to validate the DNS responses from signed zones. Unfortunately, the signed zones are typically those that are exploited for DoS attacks since they generate a very large amplification factor (due to cryptographic keys and signatures).

The signed zones were already abused in the largest DNS amplification DoS attacks, although, most zones do not support DNSSEC, and less than 1% of top-million Alexa are signed, [14]. In addition, DNSSEC still faces deployment challenges, and is suffering from impeded adoption, [8], this should change in the future, when the obstacles are resolved.

Identifying Standard Resolvers

Typical networks support a number of common variations for configuring recursive DNS resolvers, such that a number of different IP addresses participate in a resolution chain of a single DNS request. Since the challenge-response mechanisms perform authentication of the *source IP address* that sent the request, and implicitly assume that the chain of the requests will be served by the same resolver throughout the resolution chain – the authentication of benign resolvers would fail. In Section 2, we provide evidence, based on our measurements study, showing that an increasing number of requests, in a single resolution chain, are served by multiple resolvers.

Limited to a Single Name Server

The mechanisms are limited to protection of a single name server (since they generate a dedicated zone for each name server). In addition, the defences are not suitable for protection of recursive resolvers (which are also often exploited for amplification DoS attacks).

Ineffective Against Advanced Attackers

The mechanisms provide protection only against (naive) attackers, whereas a sophisticated attacker can circumvent the defences. For instance, attackers that can do source IP spoofing, would request the cookie once, and then distribute it among all the attacking hosts, which would send requests from a spoofed source IP address, that matches the one in the cookie.

Contributions

We propose a DNS authentication system for detecting and preventing amplification DoS attacks. The system effectively and efficiently detects and thwart amplification DoS attacks, which exploit DNS resolvers or name servers, whether the attacker employs IP spoofing or not. In contrast to other proposals, it is compatible with DNSSEC and with clusters-of-resolvers; it detects attacks which exploit IP spoofing, as well as attacks which employ 'zombie' hosts that send traffic from real IP addresses.

The idea behind our design is to construct a graph of legitimate resolvers. The connected components in that graph would correspond to legitimate resolvers, while isolated nodes, to (potentially) attacking hosts. Our design also takes into account hosts that may become compromised over time, or new resolvers that may be installed – we accomplish this by using a challenge-response authentication based on the *destination IP address*, to which the cookie is echoed. In particular, our defence system is allocated an address block, and each request would need to be echoed to the correct IP within that block.

We believe that this service could be provided by cloud operators and propose a concept of *DNS Authentication As A Service*, and a design for deployment of our system on cloud platforms.

Adoption is a significant challenge to any new technology, and especially for a system whose main benefit is to the entire Internet (and not merely to group of specific early adopters). We present a game-theoretic model and analysis of the clogging defences and attacks, which predicts rapid adoption and significant reduction in DNS amplification attack traffic. Our model and analysis may be of value in other scenarios and for other systems.

2. ILLUSION OF CHALLENGE-RESPONSE AUTHENTICATION

During the recent decade the DNS infrastructure has evolved into complex platforms with multiple resolving hosts. For instance, resolvers that serve multiple clients are organised in a *resolvers farm* configuration. This enables the operators to balance the load of DNS requests among multiple physical machines, which often use a shared cache, e.g., similar to Google Public DNS.

We present a number of common configurations and then show why source authentication fails in those settings.

2.1 Complex DNS Platforms

In this section we review common configurations of the resolvers and name servers.

2.1.1 Resolver-behind-Upstream

Multiple resolvers use upstream resolvers for their DNS resolution services, e.g., see [9, 7]. This scenario typically occurs when, e.g., Internet Service Providers (ISPs) apply load-balancing to DNS requests among multiple upstream resolvers. Using an upstream forwarder is also considered to provide additional benefits, such as security and availability to resolvers, and is thus an increasingly growing configuration.

2.1.2 Resolver-behind-NAT

Resolvers can sometimes be located behind a many-to-many Network Address Translation (NAT) devices. In this case, different DNS requests may be sent by the NAT device with different source IP addresses, from the pool of addresses available to the NAT device. Recent studies, [3, 7, 1], observed that a large number of recursive DNS resolvers is located behind NAT devices, and often multiple resolvers reside behind the same NAT device. Furthermore, [11], found that 90% out of 20,000 DSL lines (from a major European ISP) were connected to the Internet via the NAT devices.

2.1.3 Common Configurations of Name Servers

Recent works, [14, 15], studied different aspects of the DNS name servers infrastructure and found new common configuration of *recursive authoritative name server* (RANS), whereby a recursive resolver is registered as an authoritative name server (in the parent domain and in the zone file of the child domain). That (server-side) recursive resolver receives DNS requests from the client-side resolvers, to the target domain and then forwards the requests to the name server hosting the zone file. The resolver then forwards the requests to the name server hosting the zone file for the target domain. Upon responses from the name server, the server-side resolver caches them and subsequently returns them to the requesting client-side resolver. The setting is illustrated in Figure 2. In addition, [15], found evidence that a large

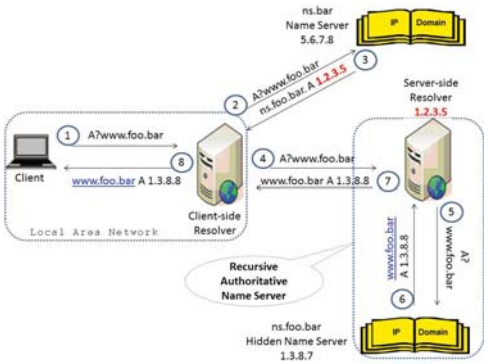


Figure 2: DNS resolution process for a domain supporting a RANS configuration.

fraction of the resolvers in a RANS configuration support *open recursion* (hence called ORANSes), i.e., respond to *any* query from any client. Furthermore, [15], found that 42% of the ORANSes use a *chain* of recursive resolvers, consisting of at least two intermediate resolvers.

2.2 Failure of Source Authentication

The challenge-response mechanisms, that we reviewed in Section 1, perform authentication of the *source IP address* that sent the request, and implicitly assume that a requests' chain, initiated by some resolver, will be served by the same resolver throughout. However, this assumption is incorrect, and prior research, [13, 15], shows that an increasing number of requests in a single resolution chain are served by multiple resolvers. Therefore, mechanisms relying on source authentication are not only ineffective, but would also disrupt the DNS availability for benign clients.

2.2.1 Protecting Name Servers vs Protecting Zones

The proposed source authentication defences, [4, 16], apply their processing to the traffic destined to a name server. The processing increases the latency for DNS requests, whether they are legitimate, and sent from real source IP addresses, or malicious and sent from spoofed source IP address part of a reflection DoS attacks on victim networks and services. The latency of all proposed defences in [4, 16] requires at least an additional round trip time (RTT), and typically two RTTs; a typical RTT is between 50 and 150 ms, but can be also much larger.

As we showed above it is common for multiple zones to be hosted on a single name server. However, not all zones may be equally abused in reflection attacks, and thus not all may be willing to sacrifice latency for the offered protection against reflection attacks. For instance, typically zones signed with DNSSEC have larger responses (see Figure 1) and are thus more frequently abused in reflection DoS attacks. Such zones would benefit from the source authentication offered by the proposed mechanisms. However, other zones, e.g., serving plain (unsigned and thus much shorter) DNS records, would not be willing to bear the extra latency.

Notice that another relevant issue is that the defences per name server may not be suitable for the business model of many zones. A name server deploying such a defence may want to increase the charge of the zones that it hosts. But, unsigned zones may not be willing to share the costs since they may not be a prey of abuse for reflection attacks.

We believe that this motivates inspection of defences on a *per zone* rather than per name server, basis.

2.2.2 Circumventing Source Authentication

Our findings show that 8% of the top 25K Alexa domains and 3.9% TLDs, out of those that we measured, use ORANSes (illustrated in Figure 2). In this case, the requests are sent to an open resolver (whose IP address is registered as the name server for the tagret domain), which is configured to forward the requests to the name server hosting the zone file. As a result, when the source authentication mechanism is hosted on the name server network, it can be trivially circumvented: the request is sent to the open resolver, which then forwards it to the name server from its *real* IP address - thus source authentication would succeed.

The problem is further exacerbated when the open recursive authoritative name server is based on a complex infrastructure, whereby a set of resolvers are responsible for receiving requests, and another set is responsible for querying the name servers, such as the infrastructure supported by Google DNS.

3. DNS AUTHENTICATION

In this section we present a design of our DNS authentication system. We first introduce the *Request-Authentication* (Section 3.1), which filters requests sent from spoofed IP addresses, and detects amplification DoS attacks. We then design the *Resolver-Authentication* (Section 3.2) that identifies standard compliant resolvers and maintains a list of potentially compromised hosts.

The *Resolver-Authentication* mechanism identifies *resolving clusters* Standard-conforming ('good') resolving clusters will be connected components in a graph, such that there will be edges between any two nodes in a component. In contrast, requests sent from spoofed IP addresses, will be isolated nodes in a graph, without any incoming edge; and requests from compromised (malicious) hosts, will only have incoming edges from itself or from other compromised hosts.

3.1 Request Authentication

The *Request-Authentication* uses a challenge-response based authentication, Figure 3, which standard resolvers can conclude correctly. In contrast, the attackers cannot successfully conclude authentication protocol since they either lack a complete (standard) DNS software and thus cannot respond to the challenge correctly or send requests from spoofed IP addresses (of the victims) and thus cannot obtain the challenge returned by the challenger in step 1. The *Request-Authentication* mechanism, Figure 3, consists of the following modules: a CHALLENGER, a VERIFIER and a CONTROLLER.

CHALLENGER. Upon receipt a DNS request, the challenger returns a (referral type) DNS response, along with a challenge, which is an IP address, randomly selected from the address block allocated to the verifier; step 2, Figure 3. The response from the challenger causes a standard-conforming DNS resolver to issue a subsequent DNS request (step 3 in Figure 3) to the name server that resides on the challenge IP address.

This response causes standard compliant resolvers to issue a subsequent request (step 3 in Figure 3), to the name server indicated by the IP address in the additional section of a DNS response.

VERIFIER. The verifier is allocated a range of IP addresses a.b.c.d/x (in CIDR notation [RFC4632]). We propose a technique allowing to artificially increase the range of IP addresses that can be allocated to the verifier, in Section 3.3.1. Upon a DNS request (step 4, Figure 3), the verifier confirms that the challenger received a matching request earlier, and that the current request arrives on the correct (challenge) IP address, selected by the challenger. The verifier also checks that the request originates from a standard resolver or a resolving cluster and was not sent from a spoofed source IP address or from a compromised host.

CONTROLLER. The challenger and the verifier communicate to the controller. Upon a request, the challenger and verifier send the request to the controller, which validates it and stores it in a database, ad checks for attacks. The challenger, in response to valid requests, receives from the controller a challenge, and verifier receives an ACK/NAK (indicating whether to forward the request to the protected DNS server, or if to discard it. When the controller receives a request from the challenger it adds an entry, corresponding to a DNS request, to table T, which maps between the query, in a DNS request, the source IP address from which the request was sent, and the challenge, i.e., the IP address to which the subsequent request should be issued, and the time at which the query arrived.

When the controller receives a request from the verifier, it first confirms that there is a corresponding entry in table T, and that the request arrived on the correct *challenge* IP address (that was provided by the challenger in step 2), and finally that the request is originated by a standard resolver and not sent from a spoofed IP address (see details in Section 3.2) and if so, forwards the request to the DNS server. When the DNS server returns a response, the verifier forwards it to the requester (after changing the source IP address to its own and destination to the address of the requester).

The controller uses a timer based mechanism which evicts old entries from table T, τ seconds after arrival; τ is the maximal delay between step 2 and 4 (Figure 3). The controller also triggers alerts on queries for which attacks are detected; details in Section 3.4. Queries with alerts can be, e.g., served over TCP.

3.2 Resolver Authentication

Resolving platforms, whereby more than a single IP address is involved in resolution of a query, are becoming dominant in current Internet. In these cases, the request sent in

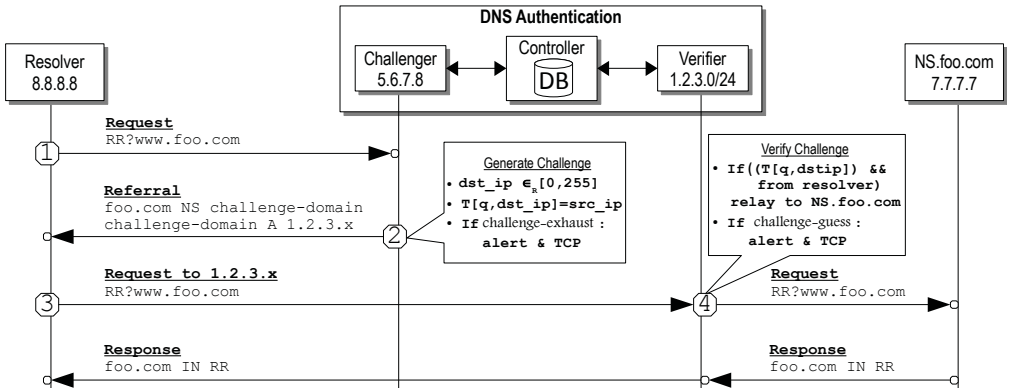


Figure 3: DNS authentication mechanism, consisting of challenger, verifier and the controller.

step 3 (Figure 3), will originate from a source IP address different than the one used in request in step 1. We develop a methodology that allows us to discover IP addresses that belong to the same cluster of DNS resolvers. Furthemore, our methodology allows to identify potentially compromised IP addresses.

Our methodology proceeds in two phases: first it constructs a graph of resolvers then parses the graph (periodically) to identify the IP addresses that exhibit a behaviour of standard-compliant resolvers, and classifies which of those IP addresses are really resolvers, and which of them are potentially compromised hosts that emulate the behaviour of standard resolvers.

The first phase is continually running and is applied on DNS requests arriving at the challenger and the verifier (in steps 1 and 2). We apply our own challenge-response mechanism (Figure 3) to construct resolvers graph. Other options are also possible for construction of the clusters. For instance, one could use TCP for classifying new resolvers, and IP addresses that successfully conclude establishment of a TCP connection would be classified as legitimate. Unfortunately, many resolvers and clients lack support for TCP and further DNS resolution over TCP incurs significantly more failures than resolution over UDP, [6, 10]; therefore, to avoid these interoperability problems we use our own mechanism (which is run over UDP). Another option would be to assume a 'quiet' initialisation phase, without attacks, for construction of the resolvers' graph.

3.2.1 Constructing Resolvers Graph

We construct a graph $G = (V, E)$ of resolvers, as follows: when a request arrives a from source IP ip_1, in step 1, we add a node to the graph $V = V \cup ip_1$. When a request in step 3 arrives from source IP ip_3, and a corresponding (matching) query was received in step 1, we add a node to the graph $V = V \cup ip_3$, and we add a directed edge $E = E \cup \{ip_1, ip_3\}$ to the graph G; if $ip_1 = ip_3$, then we add a self edge, i.e., an edge from a node to itself. The nodes and edges are added permanently only if both steps 1 and 3 were completed correctly, namely, sent a query in step 1, and followed with a correct challenge in step 3. Otherwise, a node is not added to the graph.

We make the following observation: standard-conforming individual resolvers (with a single IP) will be mapped to nodes with only self edges, and standard-conforming resolving clusters will correspond to connected components in a graph. The distribution of DNS requests, among resolvers belonging to the same cluster, is uniform in the long run. Thus resolvers belonging to the same cluster will all participate at some point of time in sending requests in step 1 and in sending requests in step 3 and therefore there will be edges between any two nodes in a component; we elaborate more on this in Section 3.2.2. In contrast, requests sent from spoofed IP addresses, will correspond to isolated nodes in a graph, or nodes without any incoming edges from legitimate resolvers, and will have only outgoing edges or incoming edges from compromised hosts.

For instance, assume a resolver platform is composed of two IP addresses `1.2.3.4` and `7.7.7.7`; see Figure 4. First request, sent in step 1 from `1.2.3.4` for query `x`, is responded with a referral. Subsequent request is sent to the challenge IP that appeared in the referral response. The authentication is successful since a request from `1.2.3.4` was correctly followed by a request from `7.7.7.7`, and both nodes are permanantly added to graph along with an edge from `1.2.3.4` to `7.7.7.7`. If at a later time, a second request is sent from `7.7.7.7` to `1.2.3.4`, and the protocol is concluded correctly, an edge is added from `7.7.7.7` to `1.2.3.4`. This results in a connected component consisting of two nodes, with edges from each one to another.

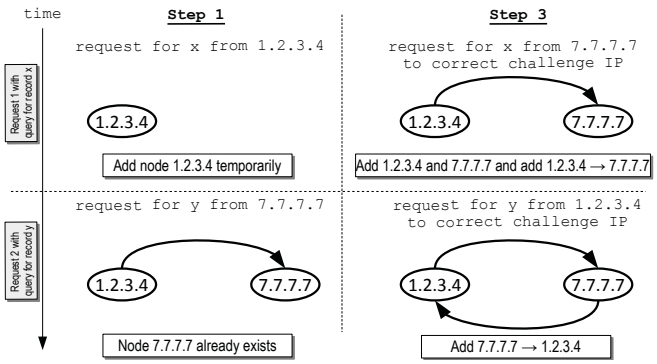


Figure 4: Construction of a cluster consisting of a connected component with two nodes, `1.2.3.4` and `7.7.7.7`.

Once a graph is constructed, a number of known algorithms, e.g., Dijkstra, can be applied to identify connected components; the vertices of a connected component are then added to form a cluster. The output is a list of clusters, $\{C_i\}_{i=1}^{\infty}$, such that each cluster C_i is a group of DNS resolvers, i.e., a set of IP addresses of resolvers.

3.2.2 Uniform Order of IP Addresses in Requests

The scenarios we discussed in Section 1 show that there are two central settings, whereby resolvers may use different IPs in consecutive requests. (1) Resolvers apply server selection algorithms when they need to decide which upstream DNS server to forward the request to, e.g., in case of forwarders. (2) Resolvers can be assigned IP addresses, e.g., if they are behind a NAT device. We show that, in the long run[2], resolvers' settings exhibit uniform order and distribution of IP addresses of the resolvers in DNS requests.

Resolvers Behind NAT Devices.

Resolvers behind a many to many NAT device will be assigned different available IP addresses each time they send a DNS request, and thus an observation period of a sufficiently large time interval will contain requests in steps 1 and 3 from different IP addresses. Due to load balancing algorithms, the same logic applies to resolvers which use multiple network interface cards, or to resolvers pools.

DNS Server Selection.

DNS resolvers apply different logic when selecting the DNS server to which they send the request. Some resolvers select a DNS server at random, e.g., WindowsDNS, others assign higher preference to least latent servers, e.g., Bind, (using the Round Trip Time (RTT) as a metric to compare the latency), but the goal of all server selection mechanisms is to distribute the load among the name servers. Prior art studied server selection behaviour with respect to unpredictability as a measure against cache poisoning attacks, e.g., [5, 17], and showed that even DNS resolvers that apply RTT-based (non-random) server selection, still periodically, e.g., every other minute, probe all the servers, i.e., including highly latent or non-responsive. Server selection logic is applied when sending requests to name servers directly, or when forwarding requests via the upstream resolvers.

3.3 Configuration and Deployment

The challenger can be configured to protect a DNS server in two ways: (1) updating the delegation NS records of the DNS server in the parent zone to point at the IP address of the challenger[3] or (2) configuring the firewall (behind which the protected DNS server resides) to forward all UDP packets, destined to port 53, to the challenger's IP address.

The later option is easier to configure but, depending on the location of the challenger, may add slightly more latecy to DNS responses.

For security, the verifier should communicate to the protected DNS server over a secure channel protocol, e.g., IPsec [RFC6071] or SSL/TLS.

[2]The time period depends on the number of DNS resolvers (or alternately IP addresses assigned to the resolvers) and on the rate of requests from the resolvers to our mechanism).

[3]Notice that this only applies to name servers and if the protected DNS server is a recursive resolver a firewall configuration should be supported.

The firewall of the network, that deploys the protection mechanism, should be configured to allow only the verifier to communicate to the DNS server. The communication between the firewall and the challenger and verifier should be over a secure channel protocol.

3.3.1 Allocating IP addresses

The verifier should be allocated a set of IP addresses, e.g., by configuring multiple network cards or via virtual IP mapping (allocating multiple IP addresses to the same network card). However, IP addresses are a scarce resource and allocating multiple addresses to the verifier may not be possible for many networks, where spare addresses is a deficiency.

We show that allocating new addresses is not essential and existing addresses can be reused for this purpose. The design is simple: *all* the IP addresses, allocated to the hosts, can be reused by the verifier, only for packets destined to port 53. Specifically, the firewall of the network should be configured to with a suitable rule, to forward all packets, destined to port 53 and with destination IP address within the address block of the network, to the verifier. We sampled the IP address blocks allocated to networks among the top million Alexa sites, `www.alexa.com`, and found that the median network block size is 2^{11}; see Figure 5.

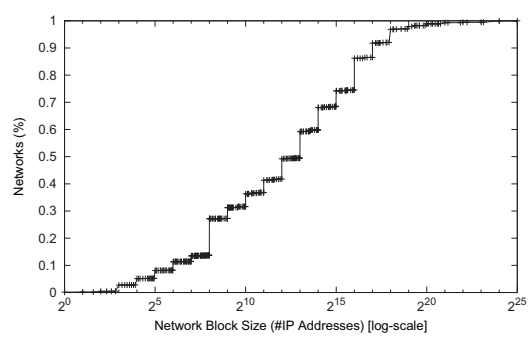


Figure 5: IP address' range sizes.

3.3.2 DNS Authentication as a Cloud Service (DAaaS)

Both the challenger and the verifier can be placed on cloud's network[4].

In our evaluations we set up the verifier hosts, the challenger and the controller on virtual machines on Amazon EC2. The database can either be placed on EC2, or, if it grows significantly, it can be placed on Amazon Simple Storage Service (S3). Amazon hosts are not expensive and in our evaluations we used a single EC2 account; a single EC2 account provides up to 25 IP addresses. We used micro virtual instances, that are offered free of charge. The advantage of deploying our design in the cloud is also that the cloud periodically changes the IP addresses, allocated to the virtual hosts, unless elastic IP address service is used (in which case a fixed and constant IP address is assigned to the client's host). This provides an artificial inflation of the address block allocated to the verifier hosts. Another source of randomisation is that the IP addresses on cloud are not allocated from a consecutive addresses' block, which

[4]The challenger and the verifier can be configured to reside on the same network as the protected name server.

makes it more difficult for the attacker to anticipate the address range among which addresses are assigned to the verifier hosts; notice that in our design the IP addresses do not need to be from a consecutive 'addresses block'.

We evaluated the overhead added by our mechanism on the resolution of DNS requests, the results are plotted in Figure 6. The evaluation consists of measurements of latencies for resolution of records in TLDs, without our mechanism, with our implementation on Amazon EC2 (US East, Northern Virginia), and with our implementation configured on our university network.

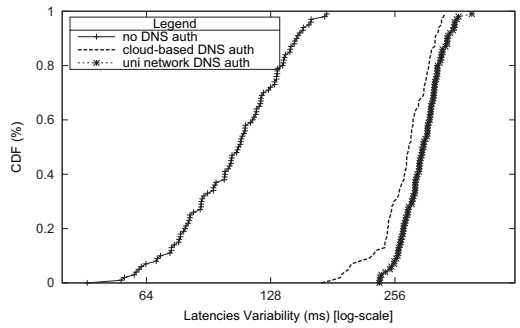


Figure 6: Latency of DNS resolution for records in TLDs. No DNS authentication, DNS authentication in cloud (Amazon EC2) and DNS authentication on university network.

3.4 Security Analysis and Evaluation

The analysis of our DNS authentication mechanism follows the steps illustrated in Figure 7. When sending the first request, in step 1 (Figure 3), the attacker can use either a spoofed source IP address or a real one. Typically to avoid exposure and losing control of compromised hosts the attacker would prefer not to use an IP address of a host under its control. Indeed many of the reflection attacks are orchestrated from spoofed source IP addresses.

When using a spoofed source IP address in step 1 (Figure 7), the attacker will not be able to obtain the challenge. As a result, it will have to guess the correct destination IP address in step 3 via *challenge exhaust/guess attack*. In order to obtain the challenge the attacker may resort to more sophisiticated strategies, such as sending packets from real (not spoofed) IP address, or launching a *challenge-intercept attack*.

3.4.1 Challenge-Exhaust/Guess Attacks

In a *challenge-exhaust attack* the attacker triggers multiple DNS requests (step (1) Figure 3), all containing the *same* query, (from spoofed source address(es)) within a valid time interval τ. This results in an allocation of the corresponding number of challenge IP addresses. As a result, the attacker can trivially hit the correct challenge destination IP address (in step (3)) by sending a DNS request with the correct query, to *any* destination IP address, from a spoofed source IP address of the victim, thus foiling the defence mechanism.

In a *challenge-guess attack* the attacker may try to guess the challenge, and following a request (in step (1)) from a spoofed IP address of a victim, the attacker would send requests to random (multiple) destination addresses (in step

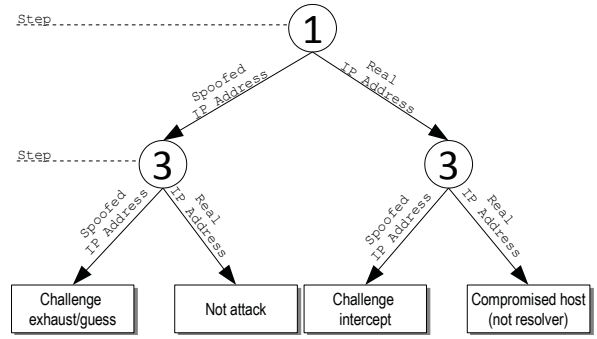


Figure 7: Attacks' flows.

(3)) hoping to guess the challenge. The concept of this attack is similar to *challenge-exhaust* attack except in this case, the attacker attempts to hit multiple destination IP addresses in the IP address block.

To counter these attacks our design selects the challenge IP address out of a set of IP addresses. If the addresses range is sufficiently large, the attacker should not be able to circumvent the DNS authentication mechanism and cause the DNS server send a large response to the victim. Let N be the address block size, then the probability that the attacker hits the correct destination IP address in step 3 (Figure 3) is $\frac{1}{N}$. To increase its success the attacker can try to occupy multiple addresses, following challenge-exhaust attack, in step 1. Let n be a number of concurrent DNS requests from the same IP address for the same query (in step 1) during interval τ seconds. Then, the success probability for guessing the correct IP address in step 3 is $\frac{n}{N}$. If $n = N$ the attacker's success probability of causing amplification is 1. Thus we also impose a limit ω on the number of requests for the same query and from the same IP (sending from different IPs decreases the success) that the attacker can send (trying to reduce the challenge range).

Assuming a typical network size of $N = 2^{11}$ (Figure 5), the threshold ω can be sufficiently large while still allowing detection of attacks.

We next calculate a threshold on the number of requests for the same resource record from the same IP address (or a set thereof) in a time interval. The query rate to the protected DNS server depends on the Time to Live (TTL) *ttl* in the requested resource record. Any subsequent requests, in time interval $[0, ttl]$ are served from the cache. The threshold in this case is $1/ttl$, namely, a request per time to live. However, during our evaluations we identified legitimate situations in which a query will be requested by the same IP address with frequency higher than TTL. This phenomenon is frequent when a number of resolvers are located behind a NAT device or behind a non-caching forwarder. In this case, the same IP, of the NAT or the forwarder, may forward the request multiple times within the TTL period.

We assume that the requests rate for a DNS record q follow a Poisson distribution and arrive at a rate λ requests per second. Let *ttl* be a Time To Live (TTL) of record q. The average interval between two DNS requests from a specific IP address $a.b.c.d$ to a specific resource record q is $\frac{1}{\lambda}$. Thus requests for a query q will be sent every interval $\lambda \cdot ttl + 1$ at rate $\frac{1}{(\lambda \cdot ttl + 1)}$. If n resolvers are located behind a single

IP address, via NAT or forwarder, then the requests rate is $\frac{n}{(\lambda \cdot ttl+1)}$.

Another parameter impacting the query rate is popularity of domains, [2], namely, the popularity of a domain is proportional to a request rate for resource records within that domain.

3.4.2 Challenge-Intercept/Compromised Host

We use our graph based approach, introduced in Section 3, for clustering and identification of standard conforming recursive DNS resolvers. This allows us to detect compromised hosts that launched the amplification attack, whether the address used to send the (subsequent) request in step 3, is spoofed or real.

If the attacker sends a request with a spoofed source IP address of the victim in step 1 (Figure 3), then it will not be able to correctly echo the challenge in step 3; since the response is sent to IP address of the victim which the attacker does not control, it will not receive the response. Thus, neither the IP of the victim nor of the attacker are added to the graph.

The attacker can lanuch a *challenge-intercept attack*. To obtain the challenge, in step 1 (Figure 3), the attacker uses an IP address which it controls. Then, the subsequent request, sent in step 3, can be issued from a spoofed source IP address of a victim but to a correct challenge IP (which was obtained in earlier step). For instance, attacker sends a first request, in step 1, from its own IP address, and subsequent request, in step 3, from IP address of the victim. In this case, two nodes, corresponding to the IP addresses of attacker and victim are added to the graph, Figure 8, with an edge from attacker's IP to victim. However, the attacker should not be able to add an opposite edge, from an IP address of the victim to an IP that it controls. This is due to the fact that it would need to cause a query from victim, to obtain the challenge value, and then to send a request from attacker's IP to the verifier in step 3 with the correct challenge.

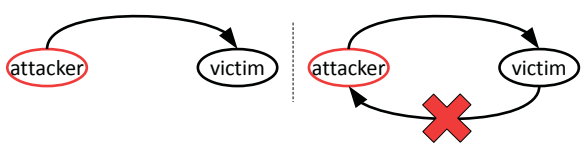


Figure 8: Disconnected components generated by malicious hosts.

To detect this attack and prevent it, we use the clustering mechanism (in Section 3.2) which enables us to identify whether the request belongs to a legitimate resolver or if exploited by an attacker.

4. THE CLOGGING GAME

In this section, we present and analyse the *clogging game*, a simplified game-theoretic model of clogging[5], considering the attacker and each of the servers (reflectors) as rational entities with perfect information. We focus on the use of DNS servers as amplifiers, and refer to the amplifiers (reflectors) as 'servers'; we denote the set of servers by S.

The clogging game is a simple, two-move, complete-information sequential game. In the first more, each of the servers $s \in S$ decides whether to adopt an anti-amplification defence or not; we denote the decision by $\alpha(s) \in \{0,1\}$. We focus on a simple binary decision, to adopt ($\alpha(s) = 1$) or not, and assume that adoption completely prevents amplification for spoofed requests - as, indeed, is the situation with the request-authentication anti-amplification mechanism. In the second move, the attacker determines whether to attack at all, and, if she attacks, what is the optimal rate of attack traffic $r^*(s)$ to send to each server $s \in S$. Note that while the first step is specific to amplification attacks, and even contains few aspects specific to DNS reflection/amplification attacks, the second step is general and relevant to any clogging attack.

We first analyse, in subsection 4.1, the second step of the clogging game; then we analyse the first move (of the reflectors/servers), in subsection 4.2.

While the model is a simplification of reality, we hope that its analysis still provides useful insights for clogging (bandwidth DoS) attacks, and specifically for the cost/benefit trade-offs for adoption of the proposed DNS request authentication mechanisms, and the implications on the expected adoption of the mechanism and on its impact on DoS-victims and attackers. The most interesting aspect is the ability of such mechanisms, to reduce the amount of DNS-amplified DoS traffic in the Internet. This is not trivial, since the defence has to be deployed by DNS servers, which are often not the victims of the clogging attack. However, even if the attack is not directed at the DNS servers, they may still have a motivation to defend against it, since the defence may reduce the amount of undesirable traffic they handle (and hence reduce expenses and waste).

Indeed, our analysis shows that this is the case, and that the defence will be adopted by all DNS servers with 'large' amplification factors. Intuitively, this holds since (1) the attacker prefers name servers with maximal amplification factor, and (2) servers which amplify high amounts of traffic, will adopt the defence to reduce costs. Hence, all attack traffic will be limited to one single (amplifying) DNS server. Furthermore, we show that typically it becomes more efficient to adopt the defence, when the attack traffic exceeds twice the benign traffic of the DNS server. Such low attack rate, implies that the defence essentially eliminates the threat of DNS amplification DDoS attacks.

4.1 Analysis of Clogging-Attacker Decisions

Attackers usually control a large set of corrupted machines ('bots'), which they control, and use to send different spoofed DNS requests to different name servers. We assume that the attacker has some 'attack cost' $C_A(r)$ associated with sending requests at total rate r (of spoofed DNS *requests per second (rps)*). Our assumption is that the cost is only a function of the total traffic rate sent by the attacker to all servers and victims. The sending cost may reflect actual costs paid by an attacker to the owner of the botnet (when the attacker leases the bots), or may reflect the loss of potential alternate profit from the bots (when the attacker 'owns' the bots directly). We adopt the typical economic assumption of increasing costs of resources, i.e., assume that C_A is positive, monotonic-increasing and

[5]Clogging attacks are bandwidth DoS attacks, based on flooding and/or the use of amplifying-reflectors, in particular, DNS servers.

convex, i.e., $C_A(r),\ C'_A(r),\ C''_A(r) > 0$ for every r.

The attacker has income from generating attack-traffic to victim hosts and networks. The attacker income from the attack-traffic may be due to direct interests of the attacker, e.g., to disable competitors or to harm an 'opponent' entity; or, the attacker may be paid for the DoS attack by some 'attack-customer'.

For simplicity, we assume that the victims are *not* in the set S (i.e., not servers), and that the attacker is only sending attack traffic (requests) to the servers (and not directly to victims). Let $\vec{r} : S \to \Re^+$ denote the allocation of attack traffic to the different servers, i.e., $r = \sum_{s \in S} \vec{r}(s)$.

When sending to a server (reflector), the attacker specifies the victim's IP address as the sender; the server is sending the clogging-traffic to the victim, as responses sent to the victim's IP address. DNS servers are often abused in this way, since the length of the reflected responses can be significantly longer than that of the spoofed requests sent by the attacker (*amplification*).

We assume a known amplification factor $\mu(s)$ for each server s, i.e., the largest amplification factor for (maliciously-chosen, spoofed) request arriving at server s. The amplification factor is only valid if the server does not adopt the defence, i.e., if $\alpha(s) = 0$. Notice that $\mu(s)$ refers to amplification of *spoofed traffic*; it is (significantly) larger than 1 for a server that do not adopt the anti-spoofing defence ($\alpha(s) = 0$). However, we assume that there is no amplification if s deploys defences, i.e., if $\alpha(s) = 1$; in fact, since in this case attacker can send directly to victim, we assume that for every server s s.t. $\alpha(s) = 1$, the attacker sends no traffic, i.e., $r^*(s) = 0$.

Each server s also has some a limit $r_A(s)$ on the maximal amount of incoming attack (spoofed, clogging) traffic; this limit is due to the input and output capabilities of the server, after reducing the capacities consumed by the benign traffic. The amount of attack-traffic $a[\vec{r}, \alpha](s)$ (reflected, and potentially amplified) sent by server $s \in S$ s.t. $\alpha(s) = 0$, for given attack rate $\vec{r}(s)$, is therefore given by $a[\vec{r}](s) = \mu(s) \cdot \min\{r_A(s), \vec{r}(s)\}$

We model the attacker's income I_A as a function of the total amount $a[\vec{r}, \alpha] \equiv \sum_{s \in S | \alpha(s)=0} a[\vec{r}](s)$ of clogging (attack) traffic she generates. We adopt the typical economic assumption of *diminishing returns*, i.e., assume that I_A is positive, monotonic increasing but *concave*, i.e., for every a holds $I_A(a) > 0,\ I'_A(a) > 0$ but $I''_A(a) < 0$.

The attacker goal is to select the optimal *attack-rate allocation* r^* to maximize the attacker utility. The utility is the difference between attacker's *income* and attacker's *costs*, i.e., $U_A[\vec{r}, \alpha] = I_A(a[\vec{r}, \alpha]) - C_A(r)$, where $r \equiv \sum_{s \in S} \vec{r}(s)$. Namely, $r^* \leftarrow \arg\max_{\vec{r}} I_A(a[\vec{r}, \alpha]) - C_A\left(\sum_{s \in S} \vec{r}(s)\right)$.

Since $\mu(s) > 1$ for every $s \in S$, it follows that the optimal allocation sends the maximal amount of traffic to the non-adopting servers with the highest amplification factors. Assume that all amplification factors differ, i.e., $(\forall s, s' \in S | s \neq s')(\mu(s) \neq \mu(s'))$. We obtain:

LEMMA 4.1. *Assume exists $s^* \in S$ s.t. $\alpha(s^*) = 0$ and $r_A(s^*) > r^*(s^*) > 0$. Then for every $s \neq s^*$ s.t. $\alpha(s) = 0$, (1) if $\mu(s) < \mu(s^*)$ then $r^*(s) = 0$ and (2) if $\mu(s) > \mu(s^*)$ then $r^*(s) = r_A(s)$.*

4.2 Analysis of Servers' Decisions

We next analyse the adoption of anti-amplification defences, specifically, DNS request-authentication, by the servers. If *all* servers adopt request-authentication, the attack traffic and the utility to the attacker will be significantly reduced - with even more significant reduction in the damages to victims and to other network users. It remains to analyse the expected adoption; for that, we must consider also the incentives of the owners of the servers.

In addition to the adversarial amplification factor $\mu(s)$, each server $s \in S$ also has a *benign amplification factors* $\eta(s)$, namely the average ratio between the size of responses and the size of corresponding benign requests. Note that the benign amplification factor $\eta(s)$ refers to legitimate (non-spoofed) requests, and is independent of the adoption of defences. Clearly, for every server $s \in S$ which does not adopt anti-spoofing defence, i.e., $\alpha(s) = 0$, holds: $\mu(s) \geq \eta(s)$ (typically, $\mu(s) >> \eta(s)$).

Server s sends responses to the benign requests at the fixed *benign rate* $r_B(s) \cdot \eta(s) \leq B(s)$. Hence, the attacker can use this server to amplify only up to total bandwidth $B(s)$, i.e., the maximal effective adversarial-input rate to s is:

$$r^*(s) \leq r_A(s) \equiv \frac{B(s) - r_B(s) \cdot \eta(s)}{\mu(s)}$$

Let $C[r^*, \alpha](s)$ denote the total cost for server s, if each server s' makes adoption decision $\alpha(s')$, and the attacker send to s' at rate $r^*(s')$. We do not model any income for name servers, i.e., a rational name server s would seek to minimize $C[r^*, \alpha](s)$, selecting $\alpha(s)$ accordingly.

We assume that the costs of every server s depend only on its own decision $\alpha(s)$ and on the amount of traffic it receives and sends, using a common linear function: $C(s) = F_{\alpha(s)} + \gamma \cdot t(s)$, where γ is a constant (price per traffic unit), F_α represents fixed equipment and operational costs, and $t(s)$ is the total traffic into and out of s. We assume that $F_1 > F_0$, i.e., the fixes costs are higher when request-authentication is deployed (reflecting the fixed costs of deploying the defence).

The traffic $t(s)$ depends on server selection α and adversarial rates r^*). Recall that the attacker will not send spoofed traffic to servers which adopted the defence, i.e. if $\alpha(s) = 1$ then $r^*(s) = 0$. Hence:

$$t(s) = \begin{cases} \alpha(s) = 1 & \Rightarrow \quad (3 + \eta(s)) \cdot r_B(s) \\ \alpha(s) = 0 & \Rightarrow \quad (1 + \mu(s)) \cdot r^*(s) + (1 + \eta(s)) \cdot r_B(s) \end{cases} \tag{1}$$

Notice that the adoption of the defence causes slight increase in overhead of handling benign requests ($3 + \eta$ instead of $1 + \eta$), due to the challenge.

Server s would rationally adopt the defence, i.e., select $\alpha(s) = 1$, if this leads to lower cost $C(s)$. By substituting the values for $t(s)$, and simple algebraic manipulations, this holds when $r^*(s)$, the rate of requests-traffic sent by the attacker to s, exceeds the *profitability threshold rate* $r_\alpha(s)$ defined as:

$$r_\alpha(s) \equiv \frac{\gamma \cdot 2r_B(s) + F_1 - F_0}{\gamma \cdot (1 + \mu(s))} \tag{2}$$

Eq. 2 does not yet allow us to determine which serves will adopt the defence, since the servers make the *first* move in the game, i.e., every server s selects $\alpha(s)$ *before* the attacker selects $r(s)$; and the adversary's choice depends on the adoption-decisions (α values) by the servers. Notice also that in this model, all servers decide simultaneously, i.e., server s cannot observe which other servers has adopted the defence.

Namely, each server s has to decide if to deploy the defence or not, based on its *prediction* of the value of $r(s)$ and of the (relevant) decisions of other servers, based on the assumption that all servers, as well as the attacker, are rational (and have perfect information, e.g., aware of the μ values for all servers).

From Lemma 4.1, the optimal amount of traffic that server s receives, $r^*_\alpha(s)$, depends only on the adoption decisions (α values) of itself and servers s' with *higher* adversarial amplification factors $\mu(s') > \mu(s)$. Since we assume perfect information, this allows each server s to compute the decisions of all servers s' with higher amplification factor, and thereby to know if it should deploy ($\alpha(s) = 1$) or not.

Consider the server s with maximal amplification factor $\mu(s)$, that decides *not* to deploy, i.e., $\alpha(s) = 0$ and $(\forall s' \neq s^*)(\alpha(s') = 1) \vee (\mu(s') < \mu(s^*)$. Since s does not deploy, surely $r^*_\alpha(s) < r_\alpha(s)$.

Assume that s has lower threshold rate than its feasible attack-request rate (otherwise, s will never amplify, and should not be in S); namely, $r_\alpha(s) \leq r_A(s)$. Combining this inequality with $r^*_\alpha(s) < r_\alpha(s)$, we have that $r^*_\alpha(s) < r_A(s)$, i.e., attacker is not fully utilizing the available attack bandwidth of s.

Since s is the non-deploying server with largest adversary amplification factor μ, is follows from Lemma 4.1 that it is the *only* server to receive any attack traffic; i.e., for every $s' \neq s$ holds $r^*(s') = 0$, and $r^*(s) < r_\alpha(s) \leq r_A(s)$. This implies a very limited amount of clogging, bounded by Eq. (2). Indeed, if we ignore the fixed 'setup costs' $F_1 - F_0$, we get that the maximal attack rate is $\frac{2r_B(s)}{\mu(s)+1}$, namely, the amount of clogging traffic is about only $2r_B(s)$ - twice the benign request rate of the server.

5. CONCLUSION

We design an anti-reflection mechanism which nullifies the amplification factor of DNS responses, when abused for DoS attacks. As a result, the attacker is required to expend an amount of resources that is equivalent to the impact on the victim network. Hence, our mechanism makes it not profitable for the attackers to abuse DNS servers, protected with our mechanis, in their reflection attacks. This provides an added value not only for the victims but also for the DNS servers, by saving their bandwith and resources for serving legitimate clients.

As a side effect, our mechanism also produces a list of 'good' DNS resolvers and resolvers' platforms as well as potentially compromised (suspect) hosts. We deployed and tested our mechanism in the cloud, and we believe that *DNS authentication as a service* would benefit a wide range of Internet clients and services.

We also developed a game theoretic model showing the profitability of our mechanism. The model is of independent interest and can be used to analyse defences against different types of DoS attacks.

Acknowledgements

This research was supported by grant 1354/11 from the Israeli Science Foundation (ISF), by the Ministry of Science and Technology, Israel and by the German Federal Ministry of Education and Research (BMBF) within EC SPRIDE, and by the Hessian LOEWE excellence initiative within CASED.

6. REFERENCES

[1] H. A. Alzoubi, M. Rabinovich, and O. Spatscheck. The anatomy of ldns clusters: findings and implications for web content delivery. In *Proceedings of the 22nd international conference on World Wide Web*, pages 83–94. International World Wide Web Conferences Steering Committee, 2013.

[2] R. P. Doyle, J. S. Chase, S. Gadde, and A. M. Vahdat. The trickle-down effect: Web caching and server request distribution. *Computer Communications*, 25(4):345–356, 2002.

[3] O. Gudmundsson and S. D. Crocker. Observing DNSSEC Validation in the Wild. In *SATIN*, March 2011.

[4] F. Guo, J. Chen, and T. cker Chiueh. Spoof Detection for Preventing DoS Attacks against DNS Servers. In *ICDCS*, pages 37–37. IEEE Computer Society, 2006.

[5] A. Herzberg and H. Shulman. Security of Patched DNS. In S. Foresti, M. Yung, and F. Martinelli, editors, *ESORICS*, volume 7459 of *LNCS*, pages 271–288. Springer, 2012.

[6] A. Herzberg and H. Shulman. DNSSEC: Security and Availability Challenges. In *Communications and Network Security (CNS), 2013 IEEE Conference on*, pages 365–366. IEEE, 2013.

[7] A. Herzberg and H. Shulman. Vulnerable Delegation of DNS Resolution. In *European Symposium on Research in Computer Security*, Lecture Notes in Computer Science. Springer, 2013.

[8] A. Herzberg and H. Shulman. Retrofitting security into network protocols: The case of dnssec. *Internet Computing, IEEE*, 18(1):66–71, 2014.

[9] C. Kreibich, N. Weaver, B. Nechaev, and V. Paxson. Netalyzr: illuminating the edge network. In *Proceedings of the 10th ACM SIGCOMM conference on Internet measurement*, pages 246–259. ACM, 2010.

[10] W. Lian, E. Rescorla, H. Shacham, and S. Savage. Measuring the Practical Impact of DNSSEC Deployment. In *Proceedings of USENIX Security*, 2013.

[11] G. Maier, F. Schneider, and A. Feldmann. NAT Usage in Residential Broadband Networks. In *Passive and Active Measurement*, pages 32–41. Springer, 2011.

[12] M. Prince. The DDoS That Almost Broke the Internet. CloudFlare Blog, April 2013.

[13] K. Schomp, T. Callahan, M. Rabinovich, and M. Allman. On measuring the client-side dns infrastructure. In *Proceedings of the 2013 conference on Internet measurement conference*, pages 77–90. ACM, 2013.

[14] H. Shulman. Is DNSSEC Ready for Prime Time: The Challenge of Legacy Infrastructure. Technical Report, September 2014.

[15] H. Shulman. The (in)Security of Outsourced DNS. Technical Report, August 2014.

[16] R. Tzakikario, D. Touitou, G. Pazi, et al. Dns anti-spoofing using udp, Nov. 2009. US Patent 7,620,733.

[17] Y. Yu, D. Wessels, M. Larson, and L. Zhang. Authority Server Selection of DNS Caching Resolvers. *ACM SIGCOMM Computer Communication Reviews*, April 2012.

Relation Extraction for Inferring Access Control Rules from Natural Language Artifacts

John Slankas, Xusheng Xiao, Laurie Williams
North Carolina State University
890 Oval Drive
Raleigh, North Carolina, 27695, USA
{john.slankas, xxiao2, laurie_williams}@ncsu.edu

Tao Xie
University of Illinois, Urbana-Champaign
201 N. Goodwin Ave
Urbana, Illinois, 61801, USA
taoxie@illinois.edu

ABSTRACT

With over forty years of use and refinement, access control, often in the form of access control rules (ACRs), continues to be a significant control mechanism for information security. However, ACRs are typically either buried within existing natural language (NL) artifacts or elicited from subject matter experts. To address the first situation, *our research goal is to aid developers who implement ACRs by inferring ACRs from NL artifacts.* To aid in rule inference, we propose an approach that extracts relations (i.e., the relationship among two or more items) from NL artifacts such as requirements documents. Unlike existing approaches, our approach combines techniques from information extraction and machine learning. We develop an iterative algorithm to discover patterns that represent ACRs in sentences. We seed this algorithm with frequently occurring nouns matching a subject–action–resource pattern throughout a document. The algorithm then searches for additional combinations of those nouns to discover additional patterns. We evaluate our approach on documents from three systems in three domains: conference management, education, and healthcare. Our evaluation results show that ACRs exist in 47% of the sentences, and our approach effectively identifies those ACR sentences with a precision of 81% and recall of 65%; our approach extracts ACRs from those identified ACR sentences with an average precision of 76% and an average recall of 49%.

Categories and Subject Descriptors

K.6.5 [**Management of Computing and Information Systems**]: Security and Protection

General Terms

Documentation, Reliability, Security, Verification.

Keywords

Security, access control, classification, natural language parsing

1. INTRODUCTION

With over forty years of use and refinement, access control, often in the form of access control rules (ACRs), remains a significant and widely-used control mechanism for information security. ACRs regulate who can perform specific actions on specific resources within a software-intensive system and are considered a critical component to ensure both confidentiality and integrity [26]. Despite access control's widespread usage, systems continue to have vulnerabilities of incorrectly implementing ACRs. In the 2011 CWE/SANS Top 25 Most Dangerous Software Errors [1], 30% of the errors directly relate to access control.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.
ACSAC '14, December 08 - 12 2014, New Orleans, LA, USA
Copyright 2014 ACM 978-1-4503-3005-3/14/12 … $15.00
http://dx.doi.org/10.1145/2664243.2664280

To meet security requirements, ACRs need to be complete, consistent, and correct [37]. Analysts must manage ACRs such that they can be evaluated against these three attributes. Existing project artifacts for systems such as use cases, requirements documents, and user manuals often capture the intended ACRs for the systems. However, manually sifting through these existing artifacts to extract the buried ACRs can be a tedious, time-consuming, and error-prone endeavor.

To address this issue, various researchers have proposed approaches for extracting ACRs from natural language (NL) artifacts. These approaches leverage techniques such as controlled natural languages (CNLs) [2, 14, 28, 29], manual analysis [6, 11], and natural language processing (NLP) [30, 35]. Each of these techniques has its own strengths and weaknesses. For example, manual analysis typically produces the most accurate results, but at the cost of requiring more skilled human evaluators and time. Using CNLs produces comprehensive results as CNLs are designed to minimize the inherent ambiguity and complexity of NL [28]. However, using CNLs typically requires specialized authoring tools and conversion of pre-existing documents [29]. NLP techniques often require less manual effort than other techniques and can work on existing documents. However, NLP techniques tend to produce less accurate results than other techniques [15].

Our research goal is to aid developers who implement ACRs by inferring ACRs from NL artifacts.

We propose a novel approach, Access Control Rule Extraction (ACRE), to allow organizations to use existing, unconstrained NL texts such as requirements documents for inferring ACRs. ACRE combines NLP, information extraction (IE), and machine learning (ML) techniques. Internally, ACRE represents NL sentences as a type dependency parse graph [22] that shows words as vertices and relationships between words as edges. Figure 1 shows the graph for the sentence "The user enters a grade for each student".

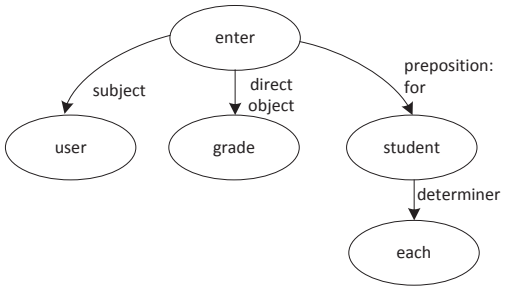


Figure 1. Type Dependency Graph

ACRE then searches these graphs for a basic pattern of *noun (subject)–verb–noun (direct object)*. ACRE creates a frequency table of all of the discovered subjects and direct objects. Any subject or direct object that occurs more times than the median count is assumed to a legitimate subject or resource (direct object) for the system that the requirements documents are for. ACRE then searches all combinations of the discovered subjects and resources within the graphs. For each found combination, the approach extracts the minimal spanning tree (MST) and assumes that any verb within the tree corresponds to the action. ACRE adds the extracted MST to a set of valid ACR patterns. ACRE then expands the pattern set by allowing a subject or resource node within the pattern to match any word of the same part of speech in other sentence graphs (i.e., "wildcarded"). ACRE then searches all of the sentence graphs for instances of the ACR patterns. The found instances are assumed to represent ACRs and the elements of the ACR are then extracted. ACRE iterates these steps, searching for additional patterns until no more patterns can be found in the sentences. To minimize incorrect patterns from being created and applied, ACRE constructs a naïve Bayes classifier to accept or reject patterns based upon domain-independent features found in other documents.

ACRE builds upon various concepts in the Text2Policy approach proposed by Xiao et al. [35]. However, ACRE incorporates IE and ML techniques to automatically learn new ACR patterns, whereas Text2Policy is limited by a fixed set of manually predefined ACR patterns.

This paper makes the following main contributions:

- Bootstrapping mechanism to seed and iteratively discover ACR patterns in NL text.
- Approach and supporting tool (built upon the combination of NLP, IE, and ML techniques) to extract ACRs.
- Labelled data set of identified ACRs and sentences[1].
- Evaluation of ACRE against documents from systems in three domains: conference management, education, and healthcare; Evaluation results showing that ACRE effectively identifies ACRs sentences with a precision of 81% and a recall of 65%, and ACRE extracts ACRs from those identified ACR sentences with an average precision of 76% and an average recall of 49%.

2. Background

This section provides background with regards to NLP, IE, ML, and the challenges faced by these techniques for extracting ACRs from NL documents.

2.1 Natural Language Processing (NLP)

NLP originates from the 1950s with the emergence of artificial intelligence research. Most notably, Chomsky's seminal work [5] demonstrated NLP's difficulty. Over 50 years later, perfect solutions do not exist for the vast majority of NLP tasks. NLP research in the 1960s and 1970s largely followed a rationalist approach with hand-built rules and grammars dominating the field [15]. In the 1980s, researchers returned to an empirical approach for NLP through the emergence of probabilistic techniques and large scale sets of annotated text [15]. These probabilistic techniques form the foundation of modern statistical parsers that represent the current state of the art in NLP [31]. Modern parsers include seven main tasks: tokenization, sentence segmentation, part-of-speech (POS) tagging, lemmatization, named-entity recognition, syntactic parsing, and coreference resolution.

Tokenization involves detecting individual words, punctuation, and other items from the text. In many situations, tokenization is straightforward in the English language due to spaces and punctuation marks; however, abbreviations, contractions, and other structures exist to complicate tokenization. Furthermore, technical documents may have structures (such as Java package names) that generally-trained tokenizers (lexical analyzers) cannot handle.

Sentence segmentation identifies sentence boundaries and then splits the tokens found in tokenization into groups.

POS tagging identifies the part-of-speech tags (e.g., noun, verbs, adjectives) for each token. The Penn Treebank [27] contains 36 different tags. The current state of the art achieves 97.3% accuracy for individual tokens and a "modest" 57% for accuracy for an entire sentence [21].

Lemmatization generates the common root word for a group of related words. For instance, sang, sung, and sings are all forms of a common lemma "sing". This common root differs from a stem, which is the root of a word after a suffix has been stripped [15]. Stems are produced through rule-drive techniques to derive base forms of words without taking into account the word's context or part of speech. In contrast, lemmatization takes into account the word's part of speech and other information sources such as a vocabulary to derive the base form of a word [20]. The state of the art achieves around 99% accuracy for the English language [9].

Named-entity recognition (NER) seeks to classify phrases into entity types (e.g., people, organizations, locations, times, vehicles, events) from text [15]. Real-world NER architectures use a combination of high-precision rules, probabilistic matching, and machine learning techniques [15]. The state of the art for the NER general task (the CoNLL-2003 shared task) has a F_1 score of .89% [18].

Syntactic parsing assigns a parse-tree structure to a sentence [15]. The tree structure provides a basis for other tasks within NLP such as question and answer, IE extraction, and translation. The state of the art parsers have an F_1 score of 90.4% [31].

Coreference resolution identifies whether or not two expressions in a document refer to the same identity. A common subset of this problem occurs within extracting ACRs from NL texts in that pronouns must be resolved to their antecedents (the actual role or resource). The state of the art has a 78% accuracy for the CoNLL-2012 Shared Task [25]. Kennedy and Boguraev [16] introduced an algorithm to resolve pronoun anaphora resolution (match the correct noun to a pronoun) that does not require parsing and achieves a 75% accuracy on their test set.

2.2 Information Extraction (IE)

IE creates structured data from text [15]. Common IE tasks include named-entity recognition, reference resolution, relation extraction (RE) and event extraction. A relation expresses the relationship among two or more items. Common relation types include "is-a" and "part-of". For example, "a doctor is a licensed healthcare practitioner (LHCP)" is represented by *is_a(doctor, LHCP)* and "medical records contain family history" is represented by *contains(medical record, family history)*. Similarly, we can have relation *writes(doctor, prescription)* to indicate that a doctor can write a prescription. The IE field has advanced largely due to the investments by the United States government through challenges sponsored by DARPA [4] and NIST[2]. State-of-the-art systems for RE in the sponsored challenges typically have around 85% precision and 70% recall [24]. In a task similar to ACR extraction, Zhu et al. [36] reported a

[1] https://sites.google.com/site/AccessControlRuleExtraction

[2] http://www.itl.nist.gov/iad/mig//tests/ace/

0.742 F_1 score for extracting medical semantic relations (*problem,tests,treatments*) from clinical texts.

ACR extraction is most similar to the RE task in that the relation (action) between the subject and resource implies much of the ACR within a sentence. However, ACR extraction differs from most RE tasks in that it is not constrained by small, fixed sets of binary relations [15]. Additionally, ACR extraction needs to infer the appropriate permissions based upon the identified relation (action). The permission may be further constrained by other features within the sentences such as negativity or access limitations to just a particular person or role.

Another IE technique is shallow parsing (semantic role labeling). This technique involves identifying the different predicates (phrases) in a sentence along with the appropriate role for each phrase [3, 15]. The labelled roles then constitute the subject, action, and resource for an ACR.

2.3 Machine Learning (ML)

ML allows us to employ an inductive technique for extracting ACRs from NL rather than specifying specific extraction rules. To decide whether or not a sentence contains an ACR, we construct a *k*-nearest neighbor classifier (*k*-NN) [10], which is a supervised learning algorithm. A *k*-NN classifier works by classifying a test item based upon which items previously classified are closest to the current test item. The classifier finds the *k* nearest "neighbors" and returns a majority vote of those neighbors to classify the test item. A distance metric determines the closeness between two items. Euclidean distance often serves as a metric for numerical attributes. For nominal values, the distance is binary – zero if the values are the same or one if the values differ. *k*-NN classifiers may use custom distance functions specific to the current problem. Advantages of *k*-NN classifiers include the ability to incrementally learn as new items are classified, to classify multiple types of data, and to handle a large number of item attributes. The primary drawback of *k*-NN classifiers involves high algorithm runtime cost; if the classifiers have *n* items stored, classification takes $O(n)$ time.

We also construct a naïve Bayes (NB) classifier to decide whether or not a generated pattern is valid. These classifiers work by selecting a class with the highest probability from a set of trained data sets given a set of features [10]. Fundamentally, it assumes that each feature of a class exists independently of other features and the probabilities are derived from the counts of features occurring in each class. Despite the simplification, the technique performs effectively in real-world problems.

We selected both classifiers after evaluating multiple types of classifiers with different features. The classification performance on the test data was used as the selection criteria. As aside, we found that a tree classifier had the highest performance for the training data in classifying patterns, but a lower performance score for the test data as the tree classifier overfits the training data.

2.4 Challenges

Many significant, complex problems exist for NLP, IE, and ML; these problems carry forward into ACR extraction.

Ambiguity issues appear in many contexts, including what meaning words have. For example, the sentence "the bank collapsed yesterday" creates confusion as one does not know whether the "bank" was a financial institution or the side of levee. Systems must disambiguate such terms based upon the context – within the sentence itself or other sentences in the document. Similarly, the same word can create issues for ACR extraction. For example, "the patient *enters* his zip code to find the closest physician" and "the administrator *enters* a new patient" have separate permissions implied for the verb "enter". In the first case, the patient searches for physicians based upon a zip code so a read permission is necessary. In the second case, the administrator effectively creates a new patient in the system and, hence, some form of a create permission is necessary.

Synonyms for the same term are another frequent source of ambiguity in NL texts. For example, "professor" and "instructor" may refer to the same role in sentences S2 and S4 in Figure 2.

Negativity, denying users access to a specific action and/or resource, is a critical problem for access control. In certain cases, users may be granted access to part of a record, but then explicitly denied from reading other parts of that record. Negativity can appear in multiple ways within sentences: negative determiners (e.g., no, zero, neither), adjectives (e.g., unable), nouns (e.g., none, nothing), verbs with negative connotation (e.g., stop, prohibit), and adverbs (e.g., never) [12]. All such possibilities should be considered, and the identified negativity should be tied back to the appropriate subject, action, or resource.

Resolution issues also appear in ACR extraction. One common situation is the appearance of pronouns in place of the actual subjects or resources. A specific case, anaphora resolution, exists when a pronoun or other phrase (e.g., that, there) refers back to a previously occurring noun or entity (the antecedent). A similar resolution situation occurs when generic terms such as "data", "entries", and "records" appear in text. Another situation occurs when the system appears to be the subject (S3 in Figure 1) when the actual actor can be identified from a prior sentence.

S1: The system displays a list of courses taught by the professor.
S2: The professor selects a course.
S3: The system displays a list of enrolled students for the course.
S4: The instructor can enter a grade for each student.

Figure 2. Sample ACP Sentences

3. Access Control Rule Extraction (ACRE)

This section details ACRE – our approach to extracting ACRs from NL text.

3.1 ACR Representation

Internally, ACRE represents sentences with a dependency graph [22] as depicted in Figure 3 for the sentence "a nurse can order a lab procedure for a patient." In Section 3.2.2, we discuss how these graphs are produced. Each vertex represents a word from the sentence along with the word's part of speech. In the figure, "NN" represents a noun, "VB" represents a verb, and "MD" represents a modal verb. Edges represent the grammatical relationship between two words. For instance, "nurse" functions as the nominal subject (nsubj) for "order", and "lab procedure" is the direct object (dobj) to be ordered. The indicators correspond to the subject("S"), action("A"), resource("R") typically defined within an ACR. Dependency graphs can be considered as trees in most situations and are typically rooted by the sentence's main verb. When conjunctions are present, vertices may have multiple parents, and thus the structure needs to be treated as a graph.

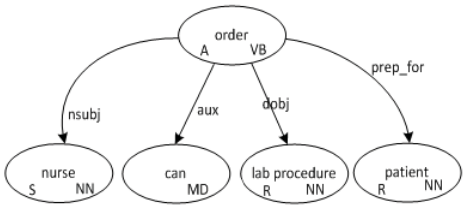


Figure 3. ACRE Sentence Representation

To represent an ACR, we use the pattern presented in Figure 4. A defines the overall ACR. s contains an order set of vertices that comprise the subject of a rule. Similarly, a and r represent the

action and resource, respectively. n contains the vertex representing negativity if required for the rule. If the rule should be limited to a particular subject s, l contains the indicating vertex. c contains additional vertices required to provide context to a given action for a set of permissions. H represents the subgraph of a sentence's dependency graph that contains the vertices and necessary edges to connect all of the vertices listed in s, a, r, n, l, c. p represents the permissions typically associated with an action. We limit permissions to have the values of "create", "retrieve", "update", and "delete" as we are primarily concerned with controlling the ability to view and manipulate data in systems. We do use "execute" for permissions that do not map to one or more of the four preceding permissions. From the example in Figure 3, we define the two rules in Figure 5.

$$A(\{s\}, \{a\}, \{r\}, [n], [l], \{c\}, H, p)$$

Figure 4. ACR Representation

$$A((nurse), (order), (lab\ procedure), (\ ), (\ ), (V: nurse, order, lab\ procedure;$$
$$E: (order, nurse, nsubj); (order, lab\ procedure, dobj)\ ), create)$$
$$A((nurse), (order), (patient), (\ ), (\ ), (V: nurse, order, patient;$$
$$E: (order, nurse, nsubj); (order, patient, prep_for)\ ), read)$$

Figure 5. Extracted ACRs

Situations exist in which not all of the ACR elements may be present within a single sentence. ACRE allows for only subjects to be missing. While users may manually identify ACRs with missing resources, ACRE has no support to find such patterns. Developers would be pointed to the surrounding sentences to finish defining the ACRs outside of the automated process.

3.2 ACRE Approach Details

The ACRE approach consists of five main steps:

1. Preprocess text documents
2. Produce dependency graphs
3. Classify each sentence as access control or not
4. Extract access control elements
5. Validate access control rules

3.2.1 Step 1: Preprocess Text Documents

In the approach, we first read the entire text into the tool built to support the approach. We separate the input into lines by either a carriage return or by periods at the end of sentences. Next, we apply a concise grammar [30] to label each token to a specific type (title, list element, list start, normal). By identifying specific types of sentences, we can increase the classification performance. Specifically, we found that section titles typically do not contain ACRs, and we design the mechanism of scoring sentence similarity to compare titles with only other titles within the document. We also found that list items require more context than just the specific item itself and process each list item combined with the start of the list.

3.2.2 Step 2: Produce Dependency Graphs

In our approach, after identifying the different sentence types, we parse each line (sentence) using the Stanford Natural Language Parser[3] and output a graph in the Stanford Type Dependency Representation (STDR) [22]. While the Stanford Parser has several output formats available, we choose the STDR because it incorporates the sentence's syntactic information in a concise and usable format and captures the grammatical relationship between words. Dependency parse trees have become a critical component for many semantic role labeling systems as the NLP community evolved in the early 2000s from a strict constituent-based (i.e., shallow parsing) systems [33]. From the STDR generated by the parser, we create the sentence representation (SR) since we need to track additional attributes for the sentence and for each word.

3.2.3 Step 3: Classify Each Sentence as Access Control or Not

Next, a k-NN classifier classifies whether a sentence contains an ACR. If the sentence does not express an ACR, we perform no further analysis on it. The k-NN classifier works by taking a majority vote of the existing classifications of the k nearest neighbors to the item under consideration. The classifier uses an adapted version of the Levenshtein distance [19] as the distance metric. Rather than using the resulting number of edits to transform one string into another as the Levenshtein distance does, our adapted distance metric computes based on the number of word transformations to change one sentence into another.

Although other machine learning algorithms can provide similar performance to a k-NN classifier, the k-NN classifier provides easier interpretation of the results for analysts since they can see a ranked list of similar sentences and associated classifications.

Once we determine that the sentence contains an ACR, the user may review the determination and correct it if necessary within the tool. Figure 6 shows a screenshot of the tool's user interface. The top table contains the document with individual columns to display the line number, sentence type, assigned classification, and completion status, assigned cluster (groups of similar sentences, optional functionality), and the sentence themselves. The dialog in the lower left allows users to review and manually enter or correct ACRs (discussed in the next section). The area in the lower right displays the SR.

3.2.4 Step 4: Extract Access Control Elements

Next, we need to extract the subject, action, and resource elements from the SR. We construct a relation extraction algorithm for the identification of ACR elements and subsequent extraction of the ACR. The algorithm follows a well-known bootstrapping technique [7], but has been adapted specifically for ACR extraction. The basic concept is to start with a small, well-known set of ACR patterns and then expand those patterns to find other, closely related patterns.

To initialize the algorithm (presented in Figure 8), we seed a set of ten basic ACR patterns with each pattern consisting of just three vertices as shown in Figure 7. Each pattern is the same, except a different verb[4] is used for a "Specific Action". Wildcards are used to match any noun in sentences containing the pattern. We initially chose the words "create", "retrieve", "update", and "delete" because the words are commonly associated with viewing and manipulating data. We then examined the frequencies of all verbs within the test documentation and chose to add more verbs associated with data and appearing with high frequencies within the document. Based upon the application domain or other documents, users may choose a different set of starting actions. From these patterns, we match all occurrences of the subjects and resources within the document along with their associated frequency counts. From the counts, we compute the median values for the subjects and resources. We then assume any word that occurs more than the median belongs to the application domain. Without a threshold, the potential for misidentified subjects and resources is much greater as any word matching the pattern would be accepted.

The subjects and resources are then stored in a list of known subjects and resources. From this listing, we then search the documents to see whether any subject exists along with any

[3] http://www-nlp.stanford.edu/software/corenlp.shtml

[4] create, retrieve, update, delete, edit, view, modify, enter, select

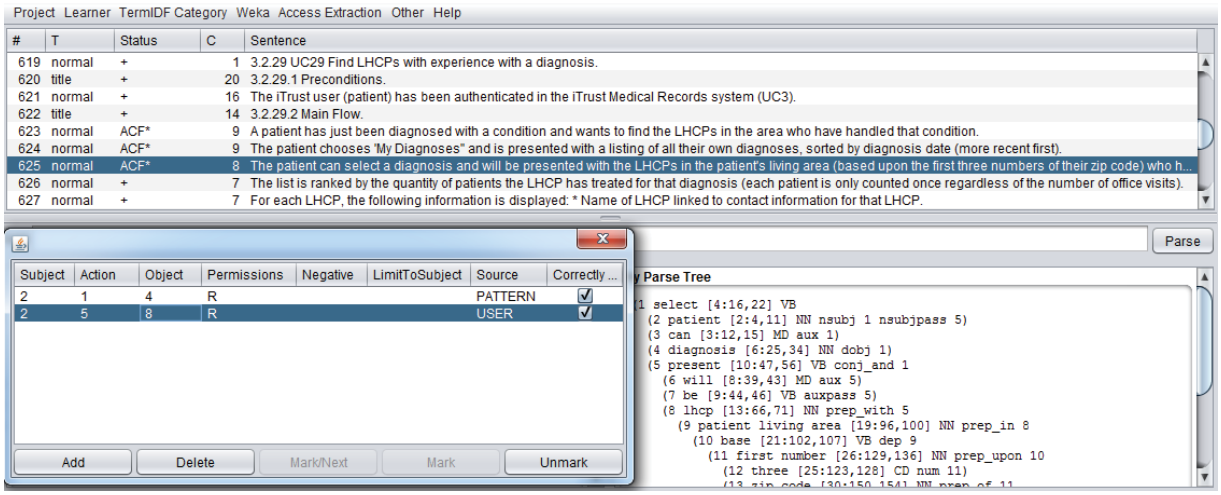


Figure 6. Tool Screenshot of Access Control Rule Extraction (ACRE)

resource. For each sentence that does match the condition, we extract the dependency pattern between subject and resource vertices. We then assume that any verbs existing in that pattern are the actions. If more than one verb exists in the shortest path from the subject to the object, we combine the verbs, but use the last appearing verb when defining permissions. In the sentence, "the administrator chooses to create a new patient", we combine "choose" and "create" to "choose create" for the action. The subject would be "administrator" and the object would be "patient". We derive permissions for each pattern by finding the closest synonym (via WordNet[5]) that has an already defined permission.

Once we extract the pattern, we apply a series of transformations to extract additional patterns that may locate additional ACRs. Specifically, we transform patterns that have an active voice into passive voice and vice versa. We also transform the patterns to assume conjunctions may exist for two or more subjects, two or more actions, and two or more resources. To find additional subjects and resources, we apply wildcards to the identified subject or resource vertices. Only the subject or the resource vertex is wildcarded to minimize semantic drift in bringing in unrelated sentences or patterns to access control.

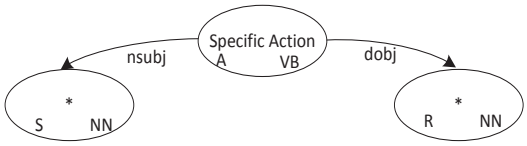


Figure 7. Basic ACR Seed Pattern

Furthermore, to ensure that the patterns are appropriate, we apply a naïve Bayes classifier to judge whether an identified pattern is appropriate. To detect the features used by the classifier, we use a forward, stepwise selection model [10]. From this model, we found the pattern itself (the POS tags and relationships between vertices) to have the biggest influence. After that, the relationships to the resources and subjects had the next largest influence. After that, the identified parts of speech for the resources and subjects improved classification performance slightly. We did not use any further features (size, specific words, use of wildcards) since we found those features began to decrease the classifier's performance.

From the pattern set, we then search the documents for sentences matching one or more patterns. Once we find any match, we check to see whether other patterns match the same sentence. If more than one pattern matches, and one pattern is contained within another pattern, we discard the former as the latter pattern provides a more specific match. Additionally, we check the matched sentences for any children vertices (of the matched pattern) that imply negativity or subject limitation (i.e., we check whether a relevant indicator exists just outside of the matched pattern).

The extracted ACR is then stored in a list for validation and output to the user. Any new subjects or resources are then added to the list of known subjects and resources. If newly discovered subjects or resources exist, then the algorithm iterates until no new items or patterns are discovered. Once the algorithm converges, the user may inject two additional patterns into the process to find more ACRs. Similar to the Basic ACR Seed Pattern in Figure 7, the first injected pattern allows any pronoun to be a subject within an ACR pattern. The second pattern searches for only actions and resources, leaving resolution of the subjects to another algorithm. The injection occurs after the algorithm initially converges to avoid spurious matches that would occur if the patterns were injected at the start of the algorithm. Additionally, the user may manually identify ACR patterns through identifying ACRs in the sentences. The information from these patterns is fed into the algorithm to search for additional extracted elements.

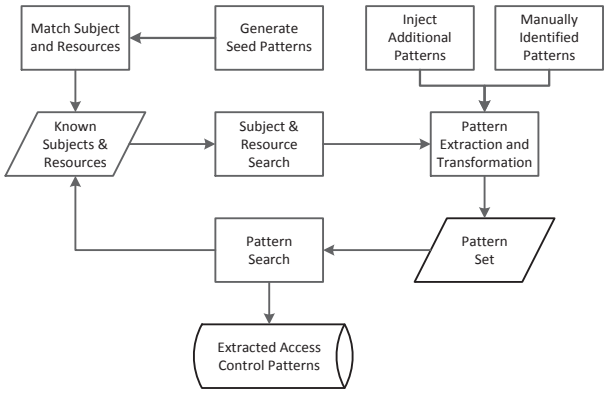


Figure 8. ACR Extraction Overview

[5] http://wordnet.princeton.edu/

Table 1. Study Documents						
Document	**Abbreviation**	**Domain**	**Number of Sentences**	**Number of ACR Sentences**	**Number of ACRs**	**Fleiss' Kappa**
iTrust for ACRE	iTrust_acre	Healthcare	1160	550	2274	0.58
iTrust for Text2Policy	iTrust_t2p	Healthcare	471	418[10]	1070	0.73
IBM Course Management	IBM_cm	Education	401	169	375	0.82
CyberChair	Cyberchair	Conference Mgmt	303	139	386	0.71
Collected ACP Documents	Collected	Multiple	142	114	258	n/a

3.2.5 Step 5: Validate Access Control Rules

In this step, the tool checks for coverage and conflicts within the extracted ACRs. Coverage is reported as the measurement for each subject as to the number of identified resources that it has ACRs identified. As we assume a default of no-access, 100% coverage is not required. However, low coverage values may indicate a need for further ACRs. Conflicts occur within ACRE when a specific subject has been both granted permission to a specific resource and restricted for the same permission on the same resource. Such conflicts may arise due to rule extraction in multiple locations or the use of a limiter to restrict access to a specific subject.

4. Study Methodology

This section presents the methodology for collecting the study documents, creating the study oracle, and running the analysis.

4.1 Study Documents

As access control widely exists in numerous domains for software systems, we chose multiple domains for the evaluation. We selected documents from the electronic healthcare, educational, and conference management domains. Additionally, to compare results to prior work, we included Xiao et al.'s study documents [35]. Table 1 lists the study documents.

For the electronic health care domain, we selected iTrust[6] [23]. The requirements consist of 40 use cases plus additional non-functional requirements, constraints, and a glossary. We used two versions of this document. The first (iTrust for ACRE) was extracted directly from the project's wiki[7] while the second (iTrust for Text2Policy) was taken from the documentation[8] used by Xiao et al. [35]. The first version more closely matches specifications used in industrial settings in that it has separate sections for introduction, glossary, non-functional requirements, and other materials. The second version includes only the use cases themselves and simplifies complex sentences to be more consistent with the rules of their parser [35]. For the educational domain, we took the eight use cases from the IBM Course Registration System [13] used in a prior research study [34]. For the conference management system, we used documents from CyberChair[9] [32], which has been used by over 475 different conferences and workshops. We also included a combined document of 114 sentences with ACRs that Xiao et al. [35] collected.

4.2 Study Oracle

To train the classifiers used within ACRE and to evaluate its performance, we developed a study oracle. To create the oracle, the first author followed the following steps:

1. Convert the document into a "text-only" format.
2. Correct the resulting text file to account for improper line breaks and other formatting issues.
3. Import the document into the ACRE tool.
4. Mark each sentence as to whether or not it is an ACR sentence.

After the classification was complete, we validated the classification through multiple ways. First, we created clusters of related sentences. We then compared the classifications within each cluster and investigated further those sentences that did not have the same classification as other sentences in the group. Additionally, as we classified each sentence, we had access to the neighbors contained within the *k*-NN classifier. This way allowed for more rapid manual classification by suggesting initial classification that we could then verify or correct as deemed necessary. Additionally, any discrepancies in the predicted classification could be easily traced back to the source sentences.

We then had two other researchers manually classify each document. We then computed the Fleiss' kappa [7] for each document (see Table 1). From Landis' and Koch's guidelines [17], scores between 0.41 and 0.60 indicate moderate agreement, while those between 0.61 and 0.80 indicate substantial agreement. 0.81 to 1.00 is considered almost perfect agreement. The three individuals then discussed the differences and collectively choose the appropriate classification for each sentence with disagreements.

The first author then manually identified each ACR within the documents. Once those rules were identified, another researcher then validated a random sampling of 20% of the ACRs from the document "iTrust for ACRE". To ensure that the reviewer diligently examined each set of tuples, we injected five "mistakes". The reviewer made 29 corrections out of the 359 ACRs and found 80% of the injected "mistakes".

To create the initial classifications of the five documents, the first author spent six hours to classify the 2,477 sentences as access control or not. Identifying the specific access tuples took additional 62 hours for the five documents.

4.3 Study Procedure

Once the oracle has been created, we ran the ACRE Tool to produce several reports to pull out details to examine properties of sentences with access control. To evaluate how well we identify sentences with ACRs, we ran the *k*-NN classifier on a combined document of the iTrust ACRE requirements, IBM Course Management, and CyberChair. We also report results on each of the five documents being individually classified. Each document was tested with stratified *n*-fold cross-validation and computed the precision, recall, and F_1 measure. With the *n*-fold cross-validation, data is randomly partitioned into *n* folds based upon each fold of approximately equal size and equal response classification. For each fold, the classifier is trained on the remaining folds and then

[6] http://agile.csc.ncsu.edu/iTrust/
[7] http://agile.csc.ncsu.edu/iTrust/wiki/doku.php?id=requirements
[8] https://sites.google.com/site/text2policy/
[9] http://www.borbala.com/cyberchair/

[10] Xiao et al. [35] reported 448 sentences with 117 containing access control for the same document evaluated from their web site (https://sites.google.com/site/text2policy/). In this work, we marked a sentence as containing an access control rule(s) when an actor performed an action with regards to some resource within the sentence. In the prior work [35], sentences with access control had to follow one of four patterns (Table 6).

Table 3. Metrics by Document					
	iTrust_acre	iTrust_t2p	IBM CM	CyberChair	Collected
Text2Policy Pattern – Modal Verb	210	130	46	71	93
Text2Policy Pattern – Passive voice w/ to Infinitive	66	21	10	39	9
Text2Policy Pattern – Access Expression	32	7	5	1	18
Text2Policy Pattern – Ability Expression	45	21	14	11	3
Number of sentences with multiple types of ACRs	383	146	77	105	36
Number of patterns appearing once or twice	680	173	162	184	97
ACRs with ambiguous subjects (e.g. "system", "user", etc.)	193	119	139	1	13
ACRs with blank subjects	557	206	29	187	5
ACRs with pronouns as subjects	109	28	5	11	11
ACRs with ambiguous objects (e.g., entry, list, name,etc.)	422	228	45	47	34

the contents of the fold are used to test the classifier. The *n* results are then averaged to produce a single result. We follow Han et al.'s recommendation [10] and use 10 as the value for *n* as this value helps produce relatively low bias and variance. The cross-validation ensures that all sentences are used for training and that each sentence is tested just once. We also report the results of using the individual documents as folds within the combined document. As it is not necessarily feasible to have a trained classifier readily available, we also evaluate the amount of work necessary to classify a document from scratch in terms of the performance when classifying the rest of the document.

In the final phase of the study, we created the naïve Bayes classifier using a document-fold way (i.e., training the classifier with all of the documents being evaluated except for one and then repeating the experiment until all documents have been tested). We then ran the ACRE process to extract the ACRs. The extracted ACRs were compared against the study oracle to determine the accuracy of the results.

4.4 Evaluation Criteria

To evaluate results, we use recall, precision, and the F_1 measure. Recall measures how many of the ACR sentences we identified from all of the sentences. Precision measures how well we identified the ACR sentences in looking at classification mistakes. To compute these values, we categorize the classifier's predictions into four categories. True positives (TP) are correct predictions. True negatives (TN) are cases where we correctly predicted that a sentence was not an ACR sentence. False positives (FP) are cases where we mistakenly identify a sentence as an ACR sentence when it is not. False negatives (FN) occur when we fail to correctly predict an actual ACR sentence. From these values, we define precision (P) as the proportion of correctly predicted classifications against all predictions against the classification under test: $P = TP/(TP + FP)$. We define recall as the proportion of classifications found for the current classification under test: R = TP/(TP+FN). The F_1measure is the harmonic mean of precision and recall, giving an equal weight to both elements: $F_1 = 2 \times \frac{P \times R}{P+R}$. From an access control perspective, high values for both precision and recall are desired. Lower precision implies that the approach could more likely identify non-ACR sentences as ACR sentences. Lower recall implies that the approach could more likely miss ACR sentences.

5. Evaluation

We present and answer our research questions in this section.

5.1 Access Control Sentence Analysis

RQ1: What patterns exist among ACR sentences?

For this research question, we explore different patterns that ACR sentences have. Xiao et al. reported four common sentence patterns for sentences with ACRs (see Table 6). In Table 2, we present the number of times we found these sentence patterns within our study documents. As we did not have access to their tool, we identified how often ACR sentences met one of their four patterns based upon when certain conditions were met. For their "Modal Verb in Main Verb Group" pattern, we checked whether a modal verb existed in an ACR sentence. For the "Passive Voice" pattern, we checked whether the sentence was in passive voice with "allow" and "to" appearing in the sentence. For the "access expression" pattern, we checked whether an ACR sentence had some form of "access" within it. Finally, for the "Ability Expression" pattern, we whether there existed some form of "access" within the sentence. Xiao et al. found that 85% of the ACRs followed these patterns in the document of iTrust for Text2Policy. The updated results derived by us showed that these patterns cover only 57% of the ACR sentences that we identified.

We also examined how frequently extracted ACRs pattern appear. Table 2 shows the top ACR patterns for all documents. The most common pattern (equivalent to ACRE's basic seed pattern) occurs approximately 14% for all ACRs. We also found a high occurrence of ACR patterns with a preposition in them.

Table 2. Top ACR Patterns	
Pattern	**Num. of Occurrences**
`(VB root(NN nsubj)(NN dobj))`	465 (14.1%)
`(VB root(NN nsubjpass))`	122 (3.7%)
`(VB root(NN nsubj)(NN prep))`	116 (3.5%)
`(VB root(NN dobj))`	72 (2.2%)
`(VB root(NN prep_%))`	63 (1.9%)

We then examined how frequently multiple types of ACRs could occur in one sentence. For example, S3 in Figure 2 has multiple forms of ACRs to be extracted. We found multiple ACRs in approximately 33% of all ACR sentences. Multiple forms of access being tied to the same action can also cause issues with permissions since different permissions may be necessary.

RQ2: How frequently do different forms of ambiguity occur in ACR sentences?

We also examined how frequently different forms of ambiguity (i.e., when a subject or object is not clearly defined in an extracted ACR), occur within the documents. We found that pronouns occurred as subjects in only 3.2% of the identified ACRs. "System" and "user" occurred as subjects in 11% of all ACRs. We also found that a subject could not be explicitly identified as a subject in 17.3% of the ACRs. We found very few instances of pronouns as objects. However, we did find that ambiguous terms (e.g., "list", "name", "record") occurred 21.5% in all of the ACRs. From these observations, a strong need exists to track actors from one sentence to another sentence to complete blank subjects or replace "system" and "user" occurrences. Similarly, an effective approach needs to be able to resolve ambiguity issues with objects. In most cases, this ambiguity resolution can be done by examining prepositions that relate directly to the ambiguous word. We also found that missing objects within ACRs only occurred in four times in all of the documents.

5.2 Identification of ACR Sentences

RQ3: How effectively does ACRE detect ACR sentences in terms of precision and recall?

Table 4 presents the results of running the classifier against each document individually with a ten-fold cross validation. We then evaluate the documents of iTrust for ACRE, IBM Course Management, CyberChair, and Collected ACP using both a 10-fold cross validation and a document-fold validation. For general use, the document-fold validation would most represent the performance as the user would use an existing classifier against a new document. For this result, ACRE had a precision of 81% and a recall of 65%. The scores for the documents of iTrust for Text2Policy and Collected ACP were abnormally high due to the high concentration of ACR sentences.

We also evaluated how ACRE would perform classifying sentences in which the user did not have a pre-trained classifier available. Figure 9 shows the classification performance for the documents based upon what percentage of the sentences in a document has been correctly classified by a user. For the document of iTrust for ACRE, the F_1 score is already above 80% after just 25% of the sentences in the document have been classified as having ACRs or not. For the sentences in the document of IBM Course Management, the performance shows two irregularities where performance decreases (at 10% and 35% completion). Based on examining the document, the first situation contains the glossary. The sentences contained here tend to have high similarity to other sentences later in the document, but do not have any corresponding user-based actions. As such, they do contain ACRs. They then cause false negatives as later sentences are parsed. In the second situation, the User Login Use Case was evaluated. As these sentences deal primarily with authentication, they do not contain ACRs. The documents for iTrust for Text2Policy and Collected ACP were not placed into the graph since the overall percentages of ACR sentences are so high that there is no practical change to the classification performance.

Table 4. Identification of ACR Sentences

Document	Precision	Recall	F_1
iTrust for Text2Policy	96%	99%	98%
iTrust for ACRE	90%	86%	88%
IBM Course Management	83%	92%	87%
CyberChair	63%	64%	64%
Collected ACP	83%	96%	89%
10-fold validation	81%	84%	83%
Document-fold validation	81%	65%	72%

5.3 Access Control Rule (ACR) Extraction

RQ4: How effectively can the subject, action, and resources elements of ACRs be extracted from ACR sentences?

Table 5 presents the results of the bootstrapping algorithm on each of the documents. We train the naïve Bayes classifier by using the identified patterns in the other documents. The document of iTrust for Text2Policy is not used for training. Also, neither of the iTrust documents are used in the training set when evaluating one of those documents.

Table 5. ACR Extraction

	Precision	Recall	F_1
iTrust for Text2Policy	80%	75%	77%
iTrust for ACRE	75%	60%	67%
IBM Course Management	81%	62%	70%
CyberChair	75%	30%	43%
Collected ACP	68%	18%	29%

ACRE performed best on document of iTrust for Text2Policy. This document contained a number of subjects and resources repeated throughout the document. As complex sentences were split into multiple, simpler sentences, there were less complex patterns to discover. ACRE performed worst on the document of Collected ACP. This document contains ACR sentences extracted from 19 sources. As little repetition exists in sentence structure, subjects, and resources, ACRE performs poorly in finding the initial set of known subjects and resources as well as in expanding the patterns. For the document of Collected ACP, if we start the algorithm with a known list of resources and subjects (which can be easily obtained from a glossary or similar section), ACRE has a precision of 85%, a recall of 70%, and a F_1 of 77%. The document of CyberChair also demonstrated a very low recall. As this document was a combination of an introductory page as well as a conference paper, little repetition exists in the sentence structure. Bootstrapping from a known list of subjects and resources only slightly improves the recall to 39% as 117 of the missing 240 ACRs were covered by patterns with more than three nodes containing missing subjects.

When examining which features to use for the naïve Bayes classifier, we found that the patterns themselves produced 80% of the possible performance. The relationships to the subjects and the objects then made the next most noteworthy performance improvements. After these three features, adding more features into the classifier reduced the classifier's performance.

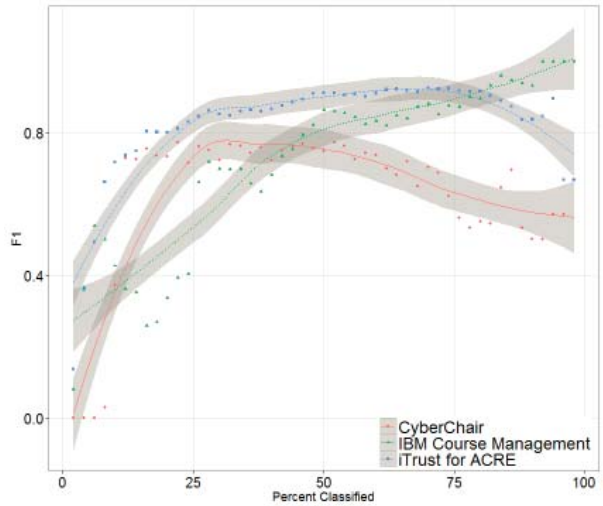


Figure 9. Classification Performance (F_1) by Completion %

6. Discussion and Future Work

Since our ACRE approach uses NLP techniques, ACRE and its supporting tool cannot extract information contained in images. With regards to ACRs, the bootstrapping mechanism does not take into account the presence of contextual information or conditions that may affect the generated ACRs. But the user can manually enter such information. The ACRE approach also requires that subjects and resources be identified as nouns and actions as verbs unless the user manually enters a rule. The approach also assumes that all necessary information for an ACR is contained within the same sentence. But it is feasible for elements of an ACR to exist in surrounding sentences of the corresponding ACR sentence.

Our approach currently does not handle resolution issues. These issues occur when a pronoun or generic term such as "system" or "data" is used in place of a descriptive term. In future work, we plan to incorporate resolution techniques to address such issues. We also plan to investigate how to search for larger ACR patterns in which the subject is missing.

The threat to external validity is mainly due to the representativeness of the subjects. To reduce the threat, we

evaluate our approach on documents from three domains. However, to further reduce the threat, additional evaluation needs to occur across multiple domains and applications. We surmise that the approach can work for other narrative-based texts, but "task/step-oriented" documents such as test scripts and user manuals would be less effective as the subject is often assumed throughout a series of steps. For such documents, we would need to study the use of "action–resource" pairs to generate patterns.

ACRE does require manual effort to setup the classifiers. We were able to identify whether or not a sentence contained an ACR at an average rate around one sentence per 9 seconds. However, this identification process is just making a simple yes or no decision and we had optimized an interface such that the user only had to press a single button per sentence. Considerably more effort is required to identify each ACR within a sentence. We identified the ACRs at an average rate of one ACR per 50 seconds. However, we found that the naïve-Bayes classifier for patterns could be used effectively across our documents and domains. In future work, we plan to study ways to reduce the workload in human annotation for NL classification tasks.

7. Related Work

This section presents related work in regards to controlled NLs and extraction of ACRs from NL artifacts.

7.1 Controlled Natural Language (CNL)

Approaches of controlled natural language (CNL) were proposed to convert NL to and from ACRs. Schwitter [28] defined a CNL as "an engineered subset of a natural language whose grammar and vocabulary have been restricted in a systematic way in order to reduce both ambiguity and complexity of full natural language." While a CNL provides consistent, semantic interpretations, a CNL limits authors to the defined grammar and typically requirse language-specific tools to stay within the language constraints. Previously created project documents cannot be used as inputs without pre-processing the documents into CNL-specific tools. Rules authored outside of the tools must conform to strictly limited grammars to be automatically parsed. Brodie et al. [2] used such approach in the SPARCLE Policy Workbench. By using their own NL parser and a controlled grammar, they effectively translated from NL into the formal rules. Inglesant et al. [14] demonstrated similar success with their tool, PERMIS, which used a role-based authorization model. However, they reported issues with users not comprehending the predefined "building blocks" imposed by using a CNL. Recently, Shi and Chadwick [29] presented their approach to authoring ACRs using a CNL. While they showed the improved usability of CNL interface, their approach was limited in the complexity of the rules that could be created since their supporting tool did not support conditions such as previous actions that must be taken before a user could access data. ACRE removes the CNL constraints, working against original, unconstrained texts.

7.2 Natural Language and Access Control

NL sources have been analyzed to infer and generate ACRs. Fernandez and Hawkins [6] presented a basic overview of extracting RBAC from use cases in 1997. Fontaine [8] proposed an approach based upon goal-based requirements engineering to extract authorization and obligation rules from NL texts into a policy language. He and Antón [11] proposed an approach to generate ACRs from NL based upon available project documents, database design, and existing rules. Using a series of heuristics, developers manually analyze the documents to find ACRs whereas our approach seeks to automatically extract ACRs.

7.3 Text2Policy

Xiao et al. proposed Text2Policy [35], for automated extraction of ACRs. Text2Policy accepts use cases in NL text as input and outputs the extracted ACRs in the eXtensible Access Control Markup Language (XACML) format. Text2Policy first uses shallow parsing techniques with finite state transducers to annotate sentences with "phrases, clauses, and grammatical functions of phrases such as subject, main verb, and object." From those annotations, Text2Policy matches a sentence into one of four possible access control patterns (Table 6). If such a match can be made, Text2Policy classifies the sentence as an ACR.

Once Text2Policy classifies sentences as ACRs, Text2Policy uses the annotated portions of the sentences to extract the subject, action, and object from the sentence. Text2Policy uses a pre-defined domain dictionary to associate the action with specific semantic classes such as "UPDATE" and "DELETE" to determine appropriate permissions for the ACR.

While ACRE and Text2Policy both target the problem of ACR extraction from NL, they differ in their basic approaches. Fundamentally, the differences between ACRE and Text2Policy can be summarized by an inductive versus a deductive approach. ACRE applies inductive reasoning to find and extract ACRs. Text2Policy applies deductive reasoning based upon existing rules (sentence patterns) to find and extract ACRs.

ACRE identifies sentences containing ACRs through a supervised learning approach. As such, the approach requires a labelled dataset similar in structure and content to the document being analyzed. Text2Policy identifies sentences containing ACRs based upon whether or not the shallow parser can parse the sentence into one of its four required patterns. In this regard, Text2Policy has an advantage since Text2Policy does not require a labelled data set to train a classifier. However, Text2Policy can miss ACRs that do not follow one of its four patterns. In our analysis of the documents, we found that only 34.4% of the identified ACR sentences followed one of Text2Policy's patterns. Additionally, Text2Policy's NL parser required splitting longer sentences as the parser could not handle complicated sentence structures.

Text2Policy can extract only one ACR per sentence. For example, ACRE would extract two rules from S4 in Figure 2 (*instructor, enter*$_{create}$*, grade*) and (*instructor, enter*$_{read}$*, student*). Text2Policy would find only the former ACR. From our evaluation in Section 5.3, we found that sentences containing multiple ACRs account for 33% of the examined sentences.

Table 6. Text2Policy - Semantic Role Patterns in Access Control Sentences [35]

Semantic Pattern	**Examples**
Model Verb in Main Verb Group	An LHCP$_{[subject]}$ **can view**$_{[action]}$ the patient's account$_{[resource]}$. An admin$_{[subject]}$ **should not update**$_{[action]}$ the patient's account$_{[resource]}$.
Passive Voice followed by To-infinitive Phrase	An LHCP$_{[subject]}$ **is disallowed to update**$_{[action]}$ the patient's account$_{[resource]}$. An LHCP$_{[subject]}$ **is allowed to view**$_{[action]}$ the patient's account$_{[resource]}$.
Access Expression	An LHCP$_{[subject]}$ **has read**$_{[action]}$ access to the patient's account$_{[resource]}$. A patient's account$_{[resource]}$ **is accessible**$_{[action]}$ to an LHCP$_{[subject]}$.
Ability Expression	An LHCP$_{[subject]}$ **is able to read**$_{[action]}$ patient's account$_{[resource]}$. An LHCP$_{[subject]}$ **has the ability to read**$_{[action]}$ access to the patient's account$_{[resource]}$.

8. Conclusion

In this paper, we have presented an approach, ACRE, to assist developers in automatically extracting ACRs from NL documents. ACRE provides a way for developers to quickly generate an initial set of ACRs with traceability back to the originating sentences. Developers can apply the approach to detect conflicts in generated rules as well as evaluating the coverage of the generated rules to the identified subjects and resources. We demonstrated how effectively a bootstrapping algorithm can extract rules from a very small initial set of patterns.

We found that ACRs exist in 47% of the examined sentences. Our ACRE approach correctly identifies ACR sentences with a precision of 81% and recall of 65%. The approach extracts ACRs from those identified ACR sentences with an average precision of 76% and an average recall of 49%. Due to the bootstrapping mechanism, the approach works better with a longer document with similar sentences structures, subjects, and resources repeated throughout the document.

9. ACKNOWLEDGMENTS

This work is supported by the USA National Security Agency (NSA) Science of Security Lablet. Any opinions expressed in this material are those of the author(s) and do not necessarily reflect the views of the NSA. We would like to thank the NCSU Realsearch group for their helpful comments on the paper.

10. REFERENCES

[1] 2011 CWE/SANS Top 25 Most Dangerous Software Errors: 2011. *http://cwe.mitre.org/top25/*. Accessed: 2011-11-14.

[2] Brodie, C. a. et al. 2006. An Empirical Study of Natural Language Parsing of Privacy Policy Rules Using the SPARCLE Policy Workbench. *In Proc. SOUPS*. (2006), 8–19.

[3] Carreras, X. and Màrquez, L. 2005. Introduction to the CoNLL-2005 Shared Task: Semantic Role Labeling. *In Proc. CoNLL* (2005), 152–164.

[4] Chinchor, N. and Sundheim, B. 1996. Message Understanding Conference - 6: A Brief History. *In Proc. Coling* (1996), 466–471.

[5] Chomsky, N. 1956. Three models for the description of language. *Information Theory, IRE Transactions on*. 2, 3 (1956), 113–124.

[6] Fernandez, E.B. et al. 1997. Determining Role Rights from Use Cases. *In Proc. ACM Workshop on RBAC* (1997), 121 – 125.

[7] Fleiss, J.L. 1971. Measuring nominal scale agreement among many raters. *Psychological Bulletin*. 76, 5 (1971), 378–382.

[8] Fontaine, P.J. 2001. *Goal-Oriented Elaboration of Security Requirements*. Université catholique de Louvain.

[9] Gesmundo, A. and Samardžić, T. 2012. Lemmatisation as a Tagging Task. *In Proce. ACL*. (2012), 368–372.

[10] Han, J. et al. 2011. *Data Mining: Concepts and Techniques*. Morgan Kaufmann.

[11] He, Q. and Antón, A.I. 2009. Requirements-based Access Control Analysis and Policy Specification (ReCAPS). *Information and Software Technology*. 51, 6 (Jun. 2009), 993–1009.

[12] Huddleston, R. and Pullman, G. 2002. *The Cambridge Grammar of the English Language*. Cambridge University Press.

[13] IBM 2004. *Course Registration Requirements*.

[14] Inglesant, P. et al. 2008. Expressions of Expertness: The Virtuous Circle of Natural Language for Access Control Policy Specification. *In Proc. SOUPS* (2008), 77–88.

[15] Jurafsky, D. and Martin, J. 2009. *Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition*. Pearson.

[16] Kennedy, C. and Boguraev, B. 1996. Anaphora for everyone: pronominal anaphora resoluation without a parser. *In Proc. Coling* (1996), 113–118.

[17] Landis, J.R. and Koch, G.G. 1977. The Measurement of Observer Agreement for Categorical Data Data for Categorical of Observer Agreement The Measurement. *Biometrics*. 33, 1 (1977), 159–174.

[18] Language-Independent Named Entity Recognition: 2003. *http://www.cnts.ua.ac.be/conll2003/ner/*.

[19] Levenshtein, V.I. 1966. Binary Codes Capable of Correcting Deletions, Insertions, and Reversals. *Soviet Physics Doklady*. 10, 8 (1966), 707–710.

[20] Manning, C. et al. 2008. *Introduction to Information Retrieval*. Cambridge University Press.

[21] Manning, C. 2011. Part-of-speech tagging from 97% to 100%: is it time for some linguistics? *In Proc. CICLing* (2011), 171–189.

[22] De Marneffe, M.-C. et al. 2006. Generating Typed Dependency Parses from Phrase Structure Parses. *In Proc. LREC* (2006), 449–454.

[23] Meneely, A. et al. 2011. iTrust Electronic Health Care System: A Case Study. *Software System Traceability*.

[24] Piskorski, J. and Yangarber, R. 2013. Information Extraction: Past, Present, and Future. *Multi-source, Multilingual Information Extraction and Summarization*. T. Poibeau et al., eds. Springer Berlin Heidelberg. 23–50.

[25] Potts, C. and Recasens, M. 2012. The Life and Death of Discourse Entities : Identifying Singleton Mentions. 0, Table 1 (2012).

[26] Samarati, P. and Vimercati, S. de 2001. Access control: Policies, models, and mechanisms. *Foundations of Security Analysis and Design*. (2001), 137–196.

[27] Santorini, B. 1995. Part-of-Speech Tagging Guidelines for the Penn Treebank Project (3rd Revision, 2nd printing). June 1990 (1995).

[28] Schwitter, R. 2010. Controlled Natural Languages for Knowledge Representation. *In Proc. CICLing* (2010), 1113–1121.

[29] Shi, L. and Chadwick, D. 2011. A Controlled Natural Language Interface for Authoring Access Control Policies. *In Proc. SAC* (2011), 1524–1530.

[30] Slankas, J. and Williams, L. 2013. Access Control Policy Extraction from Unconstrained Natural Language Text. *In Proc. PASSAT* (2013), 435–440.

[31] Socher, R. et al. 2013. Parsing with Compositional Vector Grammars. *In Proc. ACL*. (2013).

[32] Stadt, R. Van De 2012. Cyberchair: A web-based groupware application to facilitate the paper reviewing process. *arXiv arXiv:1206.1833*. (2012).

[33] Surdeanu, M. and Johansson, R. 2008. The CoNLL-2008 Shared Task on Joint Parsing of Syntactic and Semantic Dependencies. *In Proc. CoNLL* (2008), 159–177.

[34] Vidya Sagar, V.B.R. and Abirami, S. 2014. Conceptual modeling of natural language functional requirements. *Journal of Systems and Software*. 88, (Feb. 2014), 25–41.

[35] Xiao, X. et al. 2012. Automated Extraction of Security Policies from Natural-Language Software Documents. *In Proc. FSE* (2012).

[36] Zhu, X. et al. 2013. Detecting concept relations in clinical text: insights from a state-of-the-art model. *Journal of biomedical informatics*. 46, 2 (May 2013), 275–85.

[37] Zowghi, D. and Gervasi, V. 2003. On the interplay between consistency, completeness, and correctness in requirements evolution. *Information and Software Technology*. 45, 14 (Nov. 2003), 993–1009.

Centrality Metrics of Importance in Access Behaviors and Malware Detections

Weixuan Mao[†] Zhongmin Cai[†*] Xiaohong Guan[†] Don Towsley[§]
[†]MOE KLINNS Lab
Xi'an Jiaotong University
Xi'an, Shaanxi, China
{wxmao, zmcai, xhguan}@sei.xjtu.edu.cn
[§]School of Computer Science
University of Massachusetts
Amherst, MA, USA
towsley@cs.umass.edu

ABSTRACT

System objects play different roles in a computer system and exhibit different degrees of importance with respect to system security. Identifying importance metrics can help us to develop more effective and efficient security protection methods. However, there is little previous work on evaluating the importance of objects from the perspective of security. In this paper, we propose a novel approach to evaluate the importance of various system objects based on a bipartite dependency network representation of access behaviors observed in a computer system. We introduce centrality metrics from network science to quantitatively measure the relative importance of system objects and reveal their inherent connections to security properties such as integrity and confidentiality. Furthermore, we propose importance-metric based models to characterize process behaviors and identify abnormal access patterns with respect to confidentiality and integrity. Extensive experimental results on one real-world dataset demonstrate that our model is capable of detecting 7,257 malware samples from 27,840 benign processes at 93.94% TPR under 0.1% FPR. Moreover, a selective protection scheme based on a partial behavioral model of important objects achieves comparable or even better results in malware detection when compared with complete behavior models. This demonstrates the feasibility of the devised importance metrics and presents a promising new approach to malware detection.

Keywords

Importance Metrics, Access Behaviors, Centrality, Malware Detection

1. INTRODUCTION

Protection of confidentiality, integrity and availability in a real computer system is a challenging task. The complex interactions between programs and various system objects, such as files, registries and memory blocks, make fine grained and accurate protection time consuming and error prone. In fact, not every object or operation is of equal importance to the security of the system. Some objects, e.g., ntdll.dll in Windows systems, are essential for the secure running of the system, while some other objects have much less impact on the security. If we can correctly identify those important objects or operations in terms of security, we will be able to protect the system more effectively and efficiently. Moreover, the important system objects are usually the targets of various attacks. Protection of the important system objects is critical to defend against computer attacks.

Prioritizing by importance is a natural and powerful strategy to solve real-world complex problems when resources, capabilities, or available time are limited. Importance estimation or measurement plays an essential role in the success of this strategy. Importance measurement is problem specific. For example, the important parts of a building according to aesthetics may be completely different from the important parts identified by structural strength. In computer science, importance measurement has been studied in social networks, co-authorship networks [21], and system call graphs from the perspectives of system stability [27], reliability [6], and connectivity [24]. However, to the best our knowledge, there is little previous work focusing on the systematic evaluation of object importance in a computer system from the perspective of security.

In this paper, we propose to measure the importance of system objects with respect to security by encoding them into networked structures from a system-wide perspective. We construct bipartite dependency networks to characterize the complex interactions between system objects according to observed access behaviors. The bipartite dependency network is a directed graph where each vertex represents a system object and each directed edge represents the dependency relationship between objects. In network science, the relative importance of vertices has been evaluated according to a number of "centrality" metrics [19]. We analyze the connections between centrality metrics in dependency networks and security properties of confidentiality and integrity, and leverage centrality metrics to quantitatively measure the importance of system objects.

Furthermore, we propose a set of importance-metric based feature vectors to characterize the process behaviors, and leverage statistical learning methods to discriminate benign and malicious processes. We evaluate the model on a data set consisting of 27,840 processes from 534 different benign programs and 7,257 malware samples. The experimental results, 93.94% true positive rate at 0.1% false positive rate, show that importance metrics can be used as an effective behavioral specification to discriminate malware from benign processes.

Importance metrics may also provide a basis for selective modeling, which can lead to an effective and efficient prioritized protection scheme. Experimental results show that selective modeling on identified important objects only can achieve comparative or even

*Corresponding author

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org
ACSAC '14, December 08 – 12, 2014, New Orleans, LA, USA
Copyright 2014 ACM 978-1-4503-3005-3/14/12 ...$15.00
http://dx.doi.org/10.1145/2664243.2664286.

better results in malware detection when compared with modeling of all access behaviors on all system objects.

The contributions of this paper are summarized as follows:

- Bipartite dependency networks on system objects. We introduce a bipartite dependency network to encode access behaviors into the dependency relationships, which offers insights into the importance of system objects from a system-wide perspective.
- The importance metric of system objects with respect to their roles in security. We leverage centrality metrics from network science [19] to identify the relative importance of objects in the dependency network, and reveal their inherent connections to security properties, i.e., confidentiality and integrity.
- Importance-metric based malware detections. We propose importance-metric based models to characterize process behaviors and identify abnormal access patterns with respect to confidentiality and integrity. Experimental results show that our model is capable of detecting 7,257 malware samples from 27,840 benign processes at 93.94% TPR under 0.1% FPR.
- Importance-metric based selective protections. We propose a selective protection scheme of using partial behavioral models on important objects to detect malware. Experimental results show that the partial behavioral model is capable of achieving comparable or even better results when compared with more complete behavior models.

The remainder of this paper is organized as follows: Section 2 presents definitions relating to the dependency network. Section 3 presents the centrality based importance metrics with their security meanings. Then we propose our importance metrics based models for malware detection in Section 4 and Section 5 demonstrates the experimental results. Related work is presented in Section 6. Finally, Section 7 concludes our work and discusses future work.

2. DEPENDENCY NETWORK OF SYSTEM OBJECTS

Access behaviors between processes, files and registries generate data flows among involved objects and provide evidence to evaluate the importance of security. Before we propose our measures on the importance of objects, we define a dependency network describing observed access behaviors.

2.1 Dependency Relationship

We refer to system calls related to the access of system resources such as files or registries as *access events*. We define the access behavior of a process as the sequence of access events generated during the execution of the process. Access events lead to data flows between the process, files, registries and other system objects. In this paper, we utilize the directions of these data flows to construct a directed graph characterizing the access behaviors between processes, files and registries. The contents involved in the data flows are ignored. The directions of the data flows are defined as follows:

- when a subject a reads an object b, we say the data of b (source) flows to a (sink).
- when a subject a writes an object b, we say the data of a (source) flows to b (sink).

The data flow introduces the dependency between objects. Then we have the following definition of a dependency relationship:

Definition 2.1. *Dependency relationship:* When data flows from an object a to an object b, we say, b depends on a, denoted by $b \Rightarrow a$.

There are two parts in a dependency relationship: a source object and a sink object. An arrow indicates that the sink object depends on the source object, denoted as sink $\Rightarrow$ source.

Based on this definition, we derive the dependencies between registries and processes, and between files and processes.

2.1.1 Dependencies Between Registries and Processes

As a critical resource in Windows™ system, the registry contains settings for low-level operating system components and for applications running on the platform. There are various kinds of access events on registries, e.g., *RegQueryValue*, *RegEnumValue*, *RegCreateValue*, *RegDeleteValue*, *RegSetValue*, etc. We only consider the direction of flows, and divide all access events on registries into dependency relationships of two directions.

For access events, such as *RegQueryValue*, *RegEnumValue*, the data flows from registry objects to process objects. The dependencies are: process objects $\Rightarrow$ registry objects.

For access events, such as *RegCreateValue*, *RegDeleteValue*, *RegSetValue*, the data flows from process objects to registry objects. The corresponding dependencies are: registry objects $\Rightarrow$ process objects.

2.1.2 Dependencies Between Files and Processes

Files are also critical resources for storing information in computing systems. The access behaviors between files and processes are more complex than those between registries and processes, since a process is an instance of a program file that is being executed. The dependency relationships between them are formulated as follows.

Firstly, for every process, there is a dependency between it and its image file, the image file is the source object and the process is the sink object.

We divide the access events into two categories. One is *files-processes*, which includes events like *CreateFile*, *QueryInformation*, *ReadFile*, *WriteFile*. The other is *processes-processes*, which includes *ProcessCreate*.

For the *files-processes* events, similar to registries, we divide them into two types. For an access event like *QueryInformation*, *ReadFile*, etc., the direction of the flow is from file objects to process objects. That is, they correspond to dependencies: process objects $\Rightarrow$ file objects. On the other hand, for access events like *CreateFile*, *WriteFile*, etc, the direction of data flow is from the process objects to the file objects. They correspond to the dependencies process objects $\Rightarrow$ file objects.

With respect to *ProcessCreate*, the process usually forks a child process with some arguments, or default arguments. We assume there is a data flow from the parent process to the child process. More precisely, the parent process will pass the arguments to the image file of the child process. Meanwhile, the parent process achieves some functionalities by executing the child process. Thus, the data exchanges between the parent process and the child process exhibit three dependencies as follows: 1) the image file of the child process $\Rightarrow$ the parent process ; 2) the image file of the parent process $\Rightarrow$ the child process; 3) the child process $\Rightarrow$ the image file of the child process.

2.2 Dependency Network

We encode the objects into the vertices of a graph, and dependency relationships, in Definition 2.1, into directed edges of the graph. We refer to this graph as the dependency network, which

depicts a picture of all of the data flow from a system-wide perspective. This directed graph can be easily constructed from observed access behaviors:

- Every distinct registry, file or process corresponds to a vertex in the dependency network.
- Every distinct dependency relationship corresponds to an edge between related vertices and the direction of the edge is specified according to dependencies.

A bipartite network is a networked structure whose vertices can be divided into two disjoint sets U and V, such that every edge connects a vertex in U to one in V. Then, we have formal definitions of two dependency networks describing dependencies between the system objects.

Definition 2.2 (Registry-Process Dependency Network)**.** *A directed graph $G(V,E)$:*

- V is a set of process objects distinguished by process name.
- U is a set of registry objects distinguished by path.
- $E = \{e(n_1,n_2)|n_1 \in V, n_2 \in U, \text{or}, n_1 \in U, n_2 \in V\}$, $e(n_1,n_2)$ denotes a dependency relationship $n_1 \Rightarrow n_2$, the sink n_1 is dependent on the source n_2.

Definition 2.3 (File-Process Dependency Network)**.** *By substituting the set of file objects into U in Definition 2.2, we obtain the definition of File-Process Dependency Network.*

3. OBJECT IMPORTANCE METRICS ON DEPENDENCY NETWORKS

The dependency network encodes the importance of involved objects with respect to the structural information. We leverage centrality, a measure of the relative importance of vertices in network science, to evaluate the importance of objects in the dependency network [19].

3.1 Importance Metrics in Security

In security, confidentiality refers to limiting information access to authorized users and preventing access by unauthorized ones. Integrity refers to the trustworthiness of information resources. The BLP [2] and Biba [5] models, two well-known access control models, are concerned with two sets of rules on confidentiality and integrity respectively.

Rule 3.1. *Rules w.r.t. Integrity*

- A subject with a given level of integrity must not read an object at a lower integrity level (no read down).
- A subject with a given level of integrity must not write to any object at a higher level of integrity (no write up).

Rule 3.2. *Rule w.r.t. Confidentiality*

- A subject at a given security level may not read an object at a higher confidentiality level (no read-up).
- A subject at a given security level must not write to any object at a lower confidentiality level (no write-down).

If we assume that all observed normal access behaviors follow BLP and Biba rules, then the corresponding dependency network defines two partial orders with respect to confidentiality and integrity.

Observation 3.3. If the sink object a depends on the source object b, i.e., $a \Rightarrow b$, then,

- the integrity of b is greater than or equal to the integrity of a.
- the confidentiality of a is greater than or equal to the confidentiality of b.

With the dependency network and the assumption that normal access behaviors obey BLP and Biba rules, we are able to make pair-wise comparisons between the importance of two connected system objects with respect to confidentiality or integrity. Herein, "connected" means there is a path connecting two objects in the directed network. Thus, the dependency network can shed light on the importance of various system objects among access behaviors. Since centrality is devised to measure the importance of vertices in network science, we believe centrality applied to the dependency network can provide a quantitative measure of the importance of system objects in terms of confidentiality and integrity respectively.

3.2 Measuring Security Importance on Dependency Networks

Measuring the importance of vertices in a graph has been studied in social network analysis for a long time. Various centralities were proposed to measure the relative importance of vertices and many of them reflect their sociological origin [19]. We leverage these centralities to measure the importance of objects in dependency networks by exploring their meanings in security.

3.2.1 In/Out-Degree centrality

In/Out-Degree centrality is the simplest centrality measure in a network, which is just the in/out-degree of a vertex, the number of in/out-edges. In a social network, it is reasonable to suppose that individuals with many connections to others are more important than those with fewer connections [19]. Although these centralities are simple, they can be very illuminating in analyzing the importance of vertices in the networks. Moreover, as basic centralities on network structure, most other centrality measures exhibit high correlations with these two centralities.

In the dependency network, vertices have large in-degrees as a consequence of many processes reading them (files or registries) or processes writing many files or registries. Thus according to the Biba model, these objects should have high integrity levels. Infection or damage of these objects by malware, can result in serious problems since the information may flow to many other connected objects. From this perspective, objects with larger in-degree are more important w.r.t. integrity. On the other hand, vertices exhibit large out-degrees as a consequence of many processes leaving or storing information into them (files or registries), or processes reading many files or registries. Thus according to the BLP model, these objects should have high confidentiality levels. If these objects were read by malware, it would cause information leakage. From this perspective, objects with large out-degrees are more important w.r.t. confidentiality.

3.2.2 PageRank

Another centrality metric, PageRank goes beyond degree centrality with the intent to account for other vertices. It corresponds to stationary distribution of a random walk by considering the adjacency matrix as the transition matrix. PageRank and its variants has been widely studied [4]. It provides an effective and efficient technique for information retrieval in world wide web, and continues to play a central role in many of web search tools nowadays. Mathematically, the metric is calculated as follows.

Let A denote adjacency matrix, then the PageRank vector $\mathbf{p}$ is the solution of:

$$\mathbf{p} = \frac{(1-c)}{n}\vec{\mathbf{1}} + c\left[[\frac{1}{d_1},\ldots,\frac{1}{d_n}]A + D\right]\mathbf{p}, \quad (1)$$

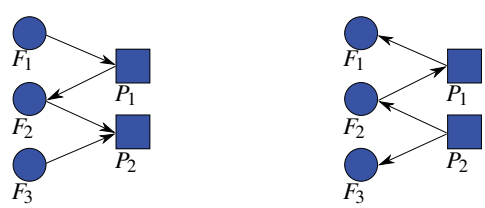
$$D = \frac{1}{n}\left[\mathbb{1}(d_1 = 0), \ldots, \mathbb{1}(d_n = 0)\right]\vec{\mathbf{1}}^T, \tag{2}$$

$$p_i = \frac{(1-c)}{n} + c\Big(\sum_{j:d_j \neq 0} p_j \frac{A_{ji}}{d_j} + \sum_{j:d_j = 0} \frac{p_j}{n}\Big), \tag{3}$$

where c is a damping factor indicating the restart in random walk. d_j is the out-degree of the object j. D is a matrix representing the dangling objects, $\mathbb{1}(A)$ is an indicator function that takes value one if predicate A is true and 0 otherwise. $\vec{\mathbf{1}}$ is a column vector with value one for all entries.

We use PageRank to derive two metrics for integrity and confidentiality respectively. The importance metric w.r.t. integrity is the PageRank vector on the dependency network, while the importance metric w.r.t. confidentiality is the PageRank vector on the transpose dependency network. The reasons are listed as follows: we find the cascading relationships between objects using Rule 3.1 and 3.2. From the perspective of integrity, the directions of edges in the dependency network are the non-decreasing directions. While, from the perspective of confidentiality, the reverse directions of edges in the dependency network are the non-decreasing directions. Strictly, we should apply partial order theory to derive the importance of the objects [16], since the Rule 3.2 and 3.1 present partial orders between objects. However, the integrity and confidentiality of the objects are dynamic in different observed executions. We assume each execution provides an observation on the integrity and confidentiality of corresponding objects, so we cat encode executions into random walks along the non-decreasing directions among objects. Hence, we measure the importance of objects by the stationary distribution of this random walk which is the PageRank.

We illustrate an example of the file-process dependency network in Fig. 1. With PageRank on the adjacent matrix shown in Fig. 1a and Fig. 1b, we obtain the measures of importance from the integrity and confidentiality perspectives, respectively.



(a) Non-decreasing directions of importance w.r.t. integrity

(b) Non-decreasing directions of importance w.r.t. confidentiality

Figure 1: An example of file-process dependency network

3.3 Importance Metrics and Malware Analysis

According to Rule 3.1, reading objects with lower integrity levels or writing objects with higher integrity levels violate the Biba model. According to Rule 3.2, reading objects with higher confidentiality levels or writing objects with lower confidentiality levels violate the BLP model. These two rules protect system resources according to the differences in secure levels of processes and the objects they access.

In fact, the true malicious actions of malware are related to the writing of objects with higher integrity levels (damaging/tampering information) or reading of objects with higher confidentiality levels (stealing information).

While a process is executing, its integrity or confidentiality level may change with time, which is a very common phenomenon in real computational environments. In these situations, we examine differences of security levels (importance in this paper) between read objects and write objects with respect to integrity or confidentiality. If there are contradictions to the Biba or BLP model, there may be an anomaly in the running of the process.

Thus, we have observations for the suspicious behavior of processes with respect to importance metrics as follows.

Observation 3.4. For a process,

- if the importance of written objects is high (integrity)
- or, if the importance of read objects is high (confidentiality)
- or, if the importance of read objects is lower than that of written objects (integrity)
- or, if the importance of read objects is higher than that of written objects (confidentiality)

then the process is suspicious.

Our malware detection approach is based on modeling the integrity or confidentiality levels, represented as importance metrics, of accessed objects.

4. IMPORTANCE-METRIC BASED MALWARE DETECTIONS

Based on observations in previous section, we believe malicious behaviors can be identified by examining the importance of accessed objects under metrics. We leverage statistical learning methods to mine the different behavioral patterns between benign and malicious processes characterized by the importance metrics.

4.1 Overview

We first give an overview of our malware detection method. Fig. 2 illustrates the framework of our importance metrics based malware detection, which we treat as a classification problem. Using a training set consisting of benign access traces, we construct the dependency network $G(V,E)$, since we assume the access traces of these benign process provide the actual importance of objects. We compute importance metrics on the dependency network using the definitions in Section 3. Then, we extract importance metrics based feature vectors of processes shown in Section 4.2. We train the classifier, which can be a generative model or a discriminative model, with the feature vectors from both benign and malicious processes in the training set. Meanwhile, feature vectors can be extracted for the testing processes. Finally, malware detections are performed on instances in the testing set with the help of the trained classifier.

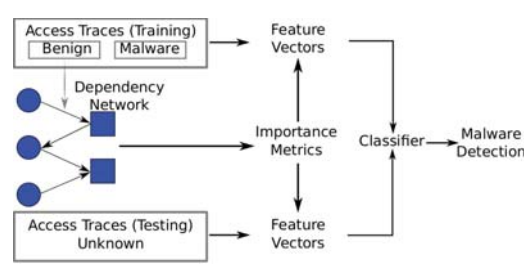


Figure 2: The framework of importance metrics based malware detection

4.2 Importance-Metric Based Descriptions

Here, we propose the description for processes by the type of access behaviors and the importance of accessed objects. The *description* of processes is also called the *feature vector* in statistical learning. We use these two terms interchangeably.

4.2.1 The complete behavior model on all objects

The access behaviors of a process consist of "in-going" edges and "out-going" edges within the dependency network. Taking registry objects as examples, "in-going" edges of the process are equivalent to the *write* operations on registries, while "out-going" edges are equivalent to the *read* operations on registries.

To form a description of a process, first, we rank all the system objects that appear in the access behaviors of benign programs according to their importance values in descending order. Objects with the same importance value occupy the same position in the rank. The importance values are derived from the importance metrics, i.e., degree or Pagerank centralities, of corresponding vertices in the dependency network. Second, we use the number of different system objects accessed in each ranking position by a process as the description or feature vector of the process.

More specifically, we represent process i in a general form, with both file objects and registry objects, as

$$\mathbf{X}_i = \left[\mathbf{x}_i^{(\text{file,read})}, \mathbf{x}_i^{(\text{file,write})}, \mathbf{x}_i^{(\text{reg,read})}, \mathbf{x}_i^{(\text{reg,write})}\right], \tag{4}$$

$$\mathbf{x}_i^{(\cdot,\star)} = \left[x_{i1}^{(\cdot,\star)}, \ldots, x_{iR(\cdot,\star)}^{(\cdot,\star)}\right], \tag{5}$$

where $\mathbf{x}_i^{(\cdot,\star)}$ represents a vector of numbers of different system objects accessed in each ranking position. $\mathbf{x}_i^{(\text{file,read})}$ and $\mathbf{x}_i^{(\text{file,write})}$ represent vectors of numbers of different system objects read and written in each ranking position of the file-process dependency network respectively. $\mathbf{x}_i^{(\text{reg,read})}$ and $\mathbf{x}_i^{(\text{reg,write})}$ represent vectors of numbers of different system objects read and written in each ranking position of the registry-process dependency network respectively. Each entry $x_{ij}^{(\cdot,\star)}$ is the number of $\cdot$ objects accessed by $\star$ behaviors with rank j. $R(\cdot,\star)$ is the number of ranking positions under the corresponding importance metric for $\cdot$ objects.

We examine different feature vectors to reveal the anomaly of processes according to Observation 3.4. Specifically, four types of feature vectors are explored corresponding to four situations in Observation 3.4:

- *S1*: Under integrity related importance metrics, the write feature vector, $\mathbf{X}_i = \left[\mathbf{x}_i^{(\text{file,write})}, \mathbf{x}_i^{(\text{reg,write})}\right]$;
- *S2*: Under confidentiality related importance metrics, the read feature vector, $\mathbf{X}_i = \left[\mathbf{x}_i^{(\text{file,read})}, \mathbf{x}_i^{(\text{reg,read})}\right]$;
- *S3*: Under integrity related importance metrics, the read/write feature vector in Eq. (4);
- *S4*: Under confidentiality related importance metrics, the read/write feature vector in Eq. (4).

4.2.2 The partial behavior model on critical objects

Considering the many system objects involved in various access behaviors, it is not practical to devise elaborate security rules for every object. Importance metrics can provide a basis for selecting the most important set of system objects on which to perform selective protection. To show the feasibility of selective protection based on importance metrics, we propose a description of processes with critical objects selected based on the importance metrics.

We define *critical objects* as the top ranked objects under importance metrics. Correspondingly, we represent a process only with objects in the top T positions, which is in contrast to every ranking position as defined in subsection 4.2.1. More specifically, we represent process i as

$$\mathbf{X}_i = \left[\mathbf{x}_i^{(\cdot,\star)}\right] = \left[\left[x_{i1}^{(\cdot,\star)}, \ldots, x_{iT(\cdot,\star)}^{(\cdot,\star)}\right]\right], \tag{6}$$

where $T(\cdot,\star)$ is the number of top rank positions of $\cdot$ objects that we selected, $T(\cdot,\star) \leq R(\cdot,\star)$, where $R(\cdot,\star)$ is the number of rank positions of all *file/registry* objects. The selections of $T(\cdot,\star)$ provide trade-offs between limited resources and accuracy.

4.2.3 Baseline

The traditional approach to the characterization of access behaviors for a process is based on the path of accessed objects. Thus we also propose an object path based descriptions as the baseline to compare with our importance-metric based descriptions. We represent process i as

$$\mathbf{X}_i = \left[\mathbf{x}_i^{(\cdot,\text{read})}, \mathbf{x}_i^{(\cdot,\text{write})}\right] = \left[\left[x_{i1}^{(\cdot,\star)}, \ldots, x_{iS(\cdot,\star)}^{(\cdot,\star)}\right]\right], \tag{7}$$

In Eq. (7), each accessed object is uniquely identified according to its path. We assign each unique path a unique integer and use $1, \ldots, S(\cdot,\star)$ to represent all $S(\cdot,\star)$ unique file or registry objects. $x_{ij}^{(\cdot,\star)}$ is binary valued indicating whether the process i accessed $\star$ object of path j by $\cdot$ behaviors or not.

4.3 Classifier Descriptions

4.3.1 The generative classifier

In the generative model, we assume the feature vector of process i, $\mathbf{x}_i$, is generated by a multinomial distributed random variable, whose values are the ranks of accessed objects under the corresponding importance metric

$$\mathbf{x}_i^{(\cdot,\star)} \sim \text{Multi}(n_i^{(\cdot,\star)}; p_1^{(\cdot,\star)}, \ldots, p_{R(\cdot,\star)}^{(\cdot,\star)}), \tag{8}$$

where $n_i^{(\cdot,\star)}$ is the number of access objects, i.e., files or registries, by the specified behavior, i.e., "read" or "write". $p_j^{(\cdot,\star)}$ is the probability that accesses the object with the ranking value j under the importance metric by the specified behavior, $\sum_j p_j^{(\cdot,\star)} = 1$. When we adopt the integrity related metric, we describe a process from the perspective of integrity. And when we adopt the confidentiality related metric, we are from the perspective of confidentiality. Meanwhile, we assume the number of accessed objects, in different executions of the program, is generated by a Poisson distributed random variable,

$$n_i^{(\cdot,\star)} \sim \text{Poisson}(\lambda^{(\cdot,\star)}). \tag{9}$$

Processes with the same label (benign or malicious) have same parameters, $\mathbf{p}$ and λ,

$$\mathbf{x}_i^{(\cdot,\star)} | y_i = l, n_i^{(\cdot,\star)} \sim \text{Multi}(n_i^{(\cdot,\star)}; p_{l1}^{(\cdot,\star)}, \ldots, p_{lR(\cdot,\star)}^{(\cdot,\star)}) \tag{10}$$

$$n_i^{(\cdot,\star)} | y_i = l \sim \text{Poisson}(\lambda_l^{(\cdot,\star)}), \tag{11}$$

where l is the label of the process, i.e., benign or malicious.

The goal is to determine the label of the process, which is the posterior probability. We derive this posterior probability using Bayes rule, with assumptions of independences for different ob-

jects and different behaviors, as follow.

$$P(y|\mathbf{X},\mathbf{n}) \propto P(\mathbf{X},\mathbf{n}|y)P(y) = P(y)\prod_{\mathbf{x}^{(\cdot,\star)}\in\mathbf{X}} P(\mathbf{x}^{(\cdot,\star)}|y,n^{(\cdot,\star)})P(n^{(\cdot,\star)}|y) \quad (12)$$

To train this model, we apply maximum likelihood estimation (MLE) to estimate λs,

$$\hat{\lambda}_l^{(\cdot,\star)} = \frac{\sum_{i=1}^N n_i^{(\cdot,\star)}\mathbb{1}(y_i=l)}{\sum_{i=1}^N \mathbb{1}(y_i=l)}. \quad (13)$$

With respect to the parameters in multinomial distributions, we also apply MLE to estimate $\mathbf{p}$, i.e.,

$$\hat{p}_{li}^{(\cdot,\star)} = \frac{\sum_j x_{ji}^{(\cdot,\star)}\mathbb{1}(y_j=l)}{\sum_j n_j^{(\cdot,\star)}\mathbb{1}(y_j=l)}. \quad (14)$$

The different values of estimated parameters exhibit the different patterns of benign and malicious processes. e.g., $\hat{\lambda}_l^{(\cdot,\star)}$ captures the different number of access behaviors between benign and malicious process on objects, $\hat{p}_{li}^{(\cdot,\star)}$ represents the probability of the process, with label l, accessed the object with rank i under the importance metric. We expect a big difference in the top ranks between benign and malicious processes, when examining $\hat{p}_i^{(\cdot,\text{write})}$ under integrity related importance metrics, or $\hat{p}_i^{(\cdot,\text{read})}$ under confidentiality related importance metrics, because of our first two points in Observation 3.4. However, due to independence assumptions, the generative model fails to capture the last two points in Observation 3.4.

4.3.2 The discriminative classifier

We also propose to leverage discriminative models for malware detection based on our importance metrics, since it does not assume the form of the joint distribution and is more flexible than the generative model. We apply discriminative classification on the feature vector of process i, $\mathbf{X}_i$, to distinguish malicious programs from benign programs. There are choices for the discriminative classifiers. Discriminative models achieve better performance in the face of high-dimensional and sparse data than the generative models.

We are able to capture more differences between benign and malicious processes under the Observation 3.4 with discriminative models than generative models. The discriminative model captures more information about the relationships between features, e.g., the difference of the importance under the metric between read objects and written objects.

5. EXPERIMENTAL RESULTS

5.1 Data Set

5.1.1 Benign Programs

In this paper, we employ Process Monitor [17] to collect process behavior data under eight different users' normal usages without interfering with their daily use, while running Microsoft Windows XP SP3. Data collections lasted over a period of 7 to 16 days. Row 2 and 3 in Table 1 describe the collected data. For example, the second column indicates that the recorded logs from user 1 were collected over a period of 10 days, and the logging time was 98 hours corresponding to the actual usage period of the user. All the collected data provide us access behaviors of 27,840 processes from 534 benign applications/programs.

Furthermore, we show the description of dependency networks constructed based on the collected access behaviors of each user in Table 1. While rows 4-6 exhibit the structural characterizations of *File-Process bipartite dependency networks*, rows 7-9 exhibit those of *Registry-Process bipartite dependency networks*.

Table 1: Descriptions of collected data and dependency networks

User	1	2	3	4	5	6	7	8
hours	98	110	60	111	83	125	80	68
days	10	14	8	16	7	13	9	14
Programs	151	133	128	130	97	86	58	163
Files	19,242	134,609	5,589	27,344	35,419	14,814	31,373	39,634
Relationships (# of edges)	33,448	150,854	40,202	57,175	57,367	24,986	48,254	68,422
Programs	132	121	113	121	87	73	46	153
Registry	150,408	31,474	28,991	46,796	42,555	41,742	32,926	103,045
Relationships (# of edges)	260,172	77,871	77,225	145,828	90,210	99,666	78,696	267,085

5.1.2 Malicious Programs

For malicious programs, we focus on 32-bit portable executable (PE) samples obtained from VxHeaven [25], which is a well known malware repository. We collected 7,257 malware samples, consisting of 2,112 Trojan, 2,563 Worms and 2,582 Virus. We develop a sandbox, in VMWare with no network connection, to record the access behavior of each sample. We run the sample on Windows XP sp3 in the virtual machine. After each run, we revert the virtual machine to a clean snapshot so that malware samples will not interfere with each other.

5.1.3 Pre-processing

The structures of the file systems differ across machines on which data was collected. We build a unified structure for the file system to resolve differences when representing same file objects observed on different machines according to the following steps:

- Ignore the drive letter, and use a root "/" to represent all drive letters.
- Treat all directories named "program files" as a single node ignoring their parent directories.
- Treat the system directory "windows" as a child of "/" ignoring its original parent directory.
- "%user profile%" represents the directory of user profile, which is a child of "/".

In order to offer an unbiased description of the access behaviors of malicious programs, we remove all captured access behaviors related to the directory where the image files of the malicious programs are located. Since we run all malicious programs at the same directory in our sandbox, a home directory for all malicious programs can become a serious artifact which will lead to untruthful good performance of malware detection.

It is very common to encounter behaviors that do not appear in the training set, especially when new behaviors/objects with the emergence of novel malware. Hence, we propose a permission inference based smoothing to describe the behaviors of new appearing objects and avoid assigning zero probability to unseen behaviors. Smoothing is used to correct inaccuracies due to data sparsity [28]. Taking the file system as a example: a user who has the permission to read/write a file may not own the permission to read/write the directory of the file; while, a user who has the permission to read/write a directory also has the permission to read/write the files under this directory. Then, from the perspective of read operations, the confidentiality of the directory is no less than that of files under it. While, from the perspective of write operations, the integrity of the directory is no less than that of files under it. Thus, we assign the maximum value of the importance metric, implying integrity/confidentiality, of all files to the directory. For

an unobserved file, we assign it an importance value equal to that of its directory. The same smoothing method can be applied on registry objects.

5.2 Methodology

5.2.1 Experimental Settings

We design two groups of experiments with the complete behavior model on all objects and the partial behavior model on critical objects.

In the first group of experiments, we examine the capabilities of importance-metric based models in malware detection with respect to the four different feature vectors of *S1*, *S2*, *S3* and *S4* defined in Section 4.2. We evaluate the performance of each type of feature vector in terms of the file object alone, the registry object alone, and both objects combined.

In the second group of experiments, we examine the effectiveness of selective protection on identified critical objects in malware detection compared with complete behavior models. We examine the performance of the model with respect to $S1$ and $S2$. We evaluate the setting of the file object alone and the registry object alone with each type of the feature vectors.

In both groups of experiments, we explore two importance metrics, i.e., degree centralities and PageRank centralities[1)], for each type of feature vector to examine their capabilities. Meanwhile, we implement the generative classifier as shown in Section 4.3.1. We use random forests, implemented in *Scikit-learn* [20], as the discriminative classifier in experiments. The random forests classifier is an ensemble learning method that constructs a multitude of decision trees [7], which exhibits good performance in many applications.

5.2.2 Training and testing sets

We run experiments to explore the capabilities of our models. Each experiment consists of eight *sub-experiments*. More precisely, for each sub-experiment, we pick the log from one of the users as the testing set, and treat logs from seven other users as the train set. Hence, we run eight sub-experiments since we have logs from eight users. We construct the dependency network using the access behavior logs from the seven other users and measure the importance values on objects. Then, the records of processes extracted from the seven users are treated as the benign instances in the training set. With respect to the malicious processes, we randomly select a fraction of the 7,257 malware to generate the malicious instances in the training set. And we use the remaining malware samples to generate the malicious instances in the testing dataset. Finally, we train the classifier for detection. We evaluate the performance of the model on the records of access behaviors from the picked-out user (to determine the false positives) and the malicious testing set (to determine the true positives).

5.2.3 Performance

We evaluate the performances with the ROC curve and area under the ROC curve (AUC). ROC curves allow us to visualize the performance of the model on malware detection by exhibiting false positive rates and true positive rates, while AUCs summarize the performance.

5.3 Detection Results

[1)]The damping factor is set to $c = 0.85$ which is the same as importance ranking for WWW web pages [8].

5.3.1 The Baseline

We conduct baseline experiments based on describing the accessed object, in the form of paths in file system, of processes. We only employ the generative classifier to it to discriminate benign processes from malicious processes. The number of all objects distinguished by paths, S shown in Eq. 7, is huge (about 10^5 shown in Table 1). Considering the memory and time consumptions, it is unpractical to employ the discriminative classifier on such high dimensional feature vectors. We compare the baseline results to importance-metric based models in following sections.

5.3.2 The complete behavior model on all objects

With randomly selecting 80% malicious instances as the malicious training set, we first show experimental results of the complete behavior model on all objects. For each sub-experiment, we can achieve an ROC and the corresponding AUC indicating the performance of the sub-experiment. Table 2 presents the average AUCs for all 8 sub-experiments with different feature vectors. Here, we present the results with feature vectors $S1$ and $S2$. In Table 2, G denotes the generative classifier, while D denotes random forest classifier. *Deg* corresponds to the use of feature vector with degree centralities, while *PR* corresponds to the use of PageRank centralities.

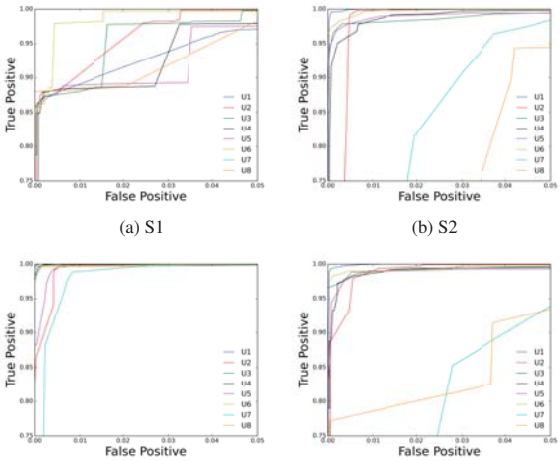
Table 2: Average AUCs for all 8 sub-experiments with different settings on all behaviors

Feature	Objects	Baseline	G(Deg)	D(Deg)	G(PR)	D(PR)
S1	File	0.7948	0.6930	0.9544	0.7847	0.9775
	Registry	0.7306	0.4838	0.9210	0.6908	0.9405
	File & Registry	0.6662	0.5491	0.9670	0.6978	0.9811
S2	File	0.6028	0.5314	0.9482	0.6088	0.9775
	Registry	0.6341	0.5099	0.9613	0.5945	0.9711
	File & Registry	0.6391	0.5194	0.9897	0.6701	0.9935

From Table 2, with the generative classifier, the complete behavior model under PageRank centrality based importance, exhibits similar performance to the baseline model, which indicates PageRank centralities contain enough information of access behaviors for the purpose of malware detection. Meanwhile, the model using PageRank centrality outperforms one using degree centrality. Comparing results of different objects, the model performs slightly better on file objects than on registry objects. The results show protection of file objects from the system wide perspective performs better than that of registry objects. Combining file and registry objects outperforms either of them. Moreover, examining "write" behavior achieves better performances than "read" behavior. The reason is that most malware damage systems by tamping/writing system objects.

To evaluate the different feature vectors, i.e., $S1$, $S2$, $S3$, $S4$, Fig. 3 presents the performance of the random forest on both files and registries under PageRank centrality based metrics, since the best performance can be found under this setting. The average AUCs of 8 ROCs of the sub-experiments on S1-S4 are 0.9811, 0.9935, 0.9996, 0.9951.

With randomly selecting 80% malicious instances as the training set. According to Observation 3.4, different feature vectors reveal the anomaly of processes from different perspectives. Only examining the writing objects under the importance metric corresponding to the integrity or the reading objects under the importance metric corresponding to the confidentiality, which are S1 or S2 respectively, provide us evidences of the suspicious behaviors since malware would write objects with high integrity or read objects with high confidentiality. However, some benign program, e.g., system programs, anti-virus programs, would also write/read



(a) S1 (b) S2

(c) S3 (d) S4

Figure 3: ROC of the random forest classifier with PageRank centrality based metrics on both file and registry objects

objects with high integrity/confidentiality since they own high integrity/confidentiality. Hence, there are limitations of models with S1 or S2, i.e., introducing false positives. The rapid increase of the false positive rates after some true positive rates can be found in Fig. 3a.

The limitations in S1 and S2 are due to the neglect of the importance level of the process, i.e., integrity and confidentiality. It is difficult to determine that in a time variant and evolving operating system. We address these limitations by combining both "reading" and "writing" objects, which provides a way to examine differences of importance levels between reading objects and writing objects under importance metrics. In Fig. 3c, 3d, improvements can be found compared with S1 and S2, in Fig. 3a, 3b, respectively. These improvements show the correctness of our models, S3 and S4 address limitations in S1 and S2.

Meanwhile, compared with S2 and S4, the better performances of S1 and S3 suggest that integrity is a better indicator than confidentiality for malware detection on our malware samples. This is consistent with previous work that integrity protections provide effective ways of defending against attacks [22, 23].

Among different feature vectors, the best performance can be found in S3, shown in Fig. 3c. Table 3 illustrates details of the performance, which are true positive rates at different false positive rates in 8 sub-experiments.

Table 3: Performance of the model shown in Fig. 3c

FPR	True Positive Rate								Average
	U1	U2	U3	U4	U5	U6	U7	U8	TPR
0%	99.52%	82.34%	97.58%	98.22%	88.16%	99.40%	0%	99.73%	83.11%
0.1%	100%	90.02%	99.27%	99.85%	92.92%	99.53%	70.21%	99.73%	93.94%
0.5%	100%	99.60%	99.86%	100%	99.57%	99.66%	95.32%	99.73%	99.21%
1.0%	100%	99.93%	100%	100%	99.86%	99.66%	99.16%	99.86%	99.80%

In Table 3, the encouraging results, 83.11% TPR at 0 FPR, 93.94% TPF at 0.1% FPR, etc., exhibit the effectiveness of the importance metrics and the importance-metric based model. It is worth noting that there is a decline in the result of U7 at the false positive rate less than 0.5%. By exploring the data and results, we find this user was studying the Windows kernel during the period of data collections. The top 5 false positives (which yield 5/1789 [2)]=0.3% FPR) stem from the program named *ml.exe*, which is in charge of compiling the code in assemble languages into *object* files (*.obj), when he compiled files in *Windows Research Kernel(WRK)*. The structure of directories and files in WRK is similar to that in Windows. So, our model treat them as modifying important files. Meanwhile, these 5 misclassified access traces are due in part to the number of access behaviors, about 10. We correctly classify the other 32 access traces of the same program whose number of access behaviors are about 40.

[2)]There are a total of 1789 benign instances in the testing set of U7.

5.3.3 The size of malicious training sets

The size of the training set influences the performance of the model in the statistical learning. Thus, we set different probabilities to randomly and uniformly select the malicious instances, and ran 8 sub-experiments. Table 4 illustrates the performance of the random forest with the feature vector $S3$ under different sizes of malicious training sets.

Table 4: Average AUC by random forest with different sizes of training sets (S3)

Obj	M	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9
File	Deg	0.9976	0.9987	0.9988	0.9988	0.9986	0.9985	0.9986	0.9983	0.9986
& Reg	PR	0.9990	0.9992	0.9992	0.9995	0.9995	0.9993	0.9994	0.9996	0.9994

Slight differences can be found in Table 4 for different sizes of malicious training sets. This indicates that the importance metric captures the key information of the access behaviors for the malware detection. The model still achieve good performances, even when only small portions of malicious instances are known.

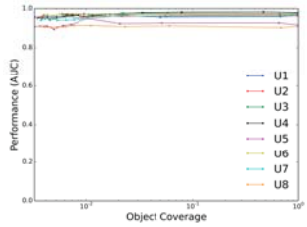
5.3.4 The partial behavior model on critical objects

Here, we illustrate the experimental results of the partial behavior model on critical objects, shown as Section 4.2.2. We randomly select 80% malicious instances as the malicious training set. Fig. 4 illustrates AUCs of classifiers under different metrics with S1 and S2. As shown in Section 4.2.2, the partial behavior model reduces the length of feature vectors for processes, since we only examine the top T critical objects. In Fig. 4, the x-axis represents the *object coverage* logarithmically. Object coverage is the fraction of critical objects that we examine among all objects, i.e., T/R as defined in Section 5.

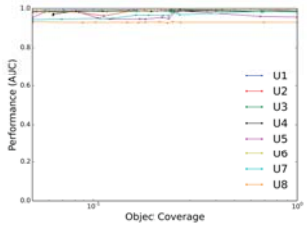
We see in Fig. 4, that ignoring the least important objects by reducing the coverage of protections on objects does not affect malware detection in our model. More specifically, considering the performance degradation by reducing the coverage of protections, the results of examining integrity (S1) is better than that of examining confidentiality (S2), since less degradation can be found in Fig. 4a-4d than Fig. 4e-4h as object coverage decreases. Meanwhile, there is less degradation on critical file objects than on registry objects. Fig. 4b tells us when we protect against attacks by integrity protections, even if we devise models on only 10% objects, selected by ranking positions under importance metrics, we can still achieve similar results as considering all objects.

These provide evidences of selective protections based on our importance metrics. Moreover, there are better performances even when we reduce the coverage of the protections in some cases. Hence, the importance metrics provide guidances or heuristics for the selective protections. The security experts can pay their limited attentions on devising access control rules on the critical objects. The elaborately devised rules will offer effective and efficient protections.

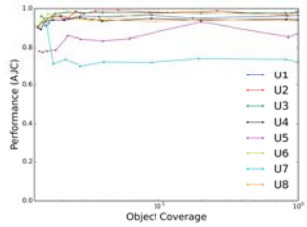
5.3.5 Comparisons with other techniques



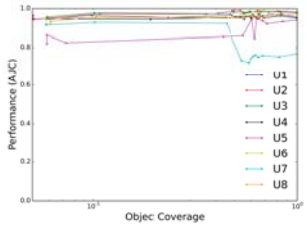
(a) Degree based importance metrics on files(S1)



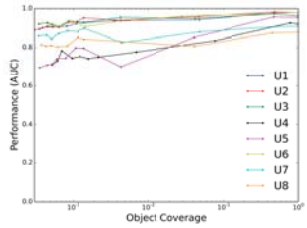
(b) PageRank based importance metrics on files(S1)



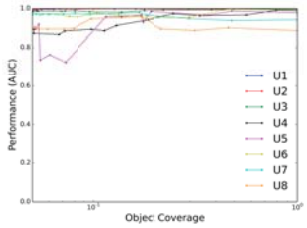
(c) Degree based importance metrics on registries(S1)



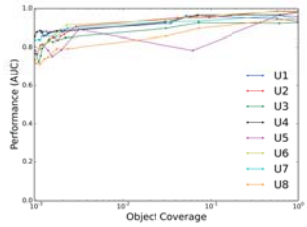
(d) PageRank based importance metrics on registries(S1)



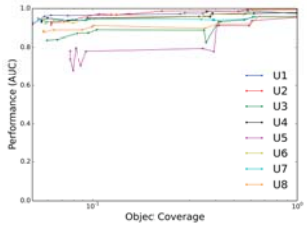
(e) Degree based importance metrics on files(S2)



(f) PageRank based importance metrics on files(S2)



(g) Degree based importance metrics on registries(S2)



(h) PageRank based importance metrics on registries(S2)

Figure 4: AUC of classifiers with different metrics on critical objects (S1 & S2)

We compare our model with other dynamic analysis techniques. So far, we found an online sandbox, Comodo instant malware analysis (CIMA) [10]. It runs dynamic analysis on executable files and reports whether files are suspicious or not. Submitting our malware samples to this online engine, the true positive is 73.24% (5,315/7,257). For the false positives, we collect all applications appeared in our benign data set in PE format. Eventually, CIMA attained 5.37% (24/447)[3)] false positive rate. As a comparison, our model achieves a malware detection performance of 93.94% TPR at 0.1% FPR.

6. RELATED WORKS

6.1 Access Control Based Malware Protection

Work has been done to devise access control rules, with respect to information flow or access behaviors, for objects [3, 15, 22]. Sun et al. defined two integrity levels and monitored the access violation in integrity, they proposed the practical proactive integrity preservation (PPI) for malware defense [22]. They devised policies and derived objects' integrity labels based on the dependencies between objects and subjects. We propose a centrality based importance metrics, in terms of integrity or confidentiality, for system objects, which assign importance levels to objects automatically and systematically.

Another similar work is done by Tobias et al., which is a quantitative graph representation for the data flow in system wide [26]. They proposed a weighted graph to capture data flows between system objects, e.g., processes, files, registries, etc. They quantitatively characterize the data in bytes by the weight of edges. Meanwhile, they proposed heuristics to detect malicious processes based on observations under the data flow graph. However, the manually crafted heuristics are limited in the face of the increase of novel malware. Our model aims to address these limitations by combining the heuristic in access controls with statistical learning algorithms.

6.2 Behavior Based Malware Detection

In dynamic analysis, many works explore the specification of programs via behavior models. The model of malware detections can be devised by leveraging statistical learning techniques on these specifications [1, 13]. In system call level, the n-gram is a classical specification for programs from the spatial perspective in the sequence of system calls [11]. Although this specification show its ability in intrusion detections, it suffers from fair performance in malware detections [15].

Other specifications explore more about the correlations between system calls of programs by capturing both spatial and temporal information [12,14,18]. Monitoring the process execution by Qemu, Martignoni et al. propose a behavior graph based malware detector, and tried to bridge the sematic gap between low-level events and high-level actions [18]. Fredrikson et al. extracted behavior specifications in graph form [12], analyzed the behavior graph by a graph mining algorithm, to distinguish benign programs and malware with achieving good performance.

Previous work of behavior based malware detection usually employ statistical learning algorithm to construct the behavior models or specifications of programs. Canali et al. presented a systematic approach to measure how the choice of behavioral specifications influences the quality of a malware detector [9]. Statistical learning algorithms do not clearly explain the security meaning of a specified behavior. However, in this paper, we propose importance-metric based models, which explicitly encode the security meaning of specifications for programs.

7. CONCLUSION AND FUTURE WORK

In this paper, we propose bipartite dependency networks to characterize the complex interactions between system objects in access behaviors in a computer system. By analyzing the connections between centrality metrics in dependency networks and security properties of confidentiality and integrity, we devise a systematic method to quantitatively evaluate the importance of system objects using centrality metrics in network science. Then, we propose importance-metric based behavior models for malware detection. Experimental results on real-world data demonstrate the effectiveness of our importance-metric based malware detections and the feasibility of the proposed importance metrics. Meanwhile, the importance metrics provide guidelines for selective protection schemes which has been shown in the experiments achieving comparable or even better results in malware detection when compared

[3)] Some of benign applications cannot be found.

with complete behavior models.

Using networked models such as the dependency network, we will explore other fine-grained objects such as sockets and memory blocks which may help us achieve a better understanding of their roles in system-wide data flows or control flows. Also, new approaches to risk assessments can be derived from our importance metrics and deserves in-depth exploration in future work.

Acknowledgements

The authors would like to thank the anonymous reviewers for their valuable comments and suggestions to improve the paper.

The research is supported by NFSC (61175039, 61221063, 6140-3301), 863 High Tech Development Plan (2012AA011003), Research Fund for Doctoral Program of Higher Education of China (20090201120032), International Research Collaboration Project of Shaanxi Province (2013KW11) and Fundamental Research Funds for Central Universities (2012jdhz08), NSF grant CNS-1065133, ARL Cooperative Agreement W911NF-09-2-0053.

8. REFERENCES

[1] F. Apap, A. Honig, S. Hershkop, E. Eskin, and S. Stolfo. Detecting malicious software by monitoring anomalous windows registry accesses. In *Proceedings of the 5th international conference on Recent advances in intrusion detection*, RAID'02, pages 36–53, Berlin, Heidelberg, 2002. Springer-Verlag.

[2] D. E. Bell and L. J. LaPadula. Secure computer systems: Mathematical foundations. ESD-TR 73-278, MITRE Corp., Bedford, MA, 1973.

[3] S. Bhatkar, A. Chaturvedi, and R. Sekar. Dataflow anomaly detection. *IEEE Symposium on Security and Privacy (S&P)*, pages 48–62, 2006.

[4] M. Bianchini, M. Gori, and F. Scarselli. Inside pagerank. *ACM Transactions on Internet Technology (TOIT)*, 5(1):92–128, 2005.

[5] Biba. Integrity considerations for secure computer systems. ESD-TR 76-372, MITRE Corp., 1977.

[6] E. Borgonovo. A new uncertainty importance measure. *Reliability Engineering & System Safety*, 92(6):771–784, 2007.

[7] L. Breiman. Random forests. *Machine Learning*, 45:5–32, 2001. 10.1023/A:1010933404324.

[8] S. Brin and L. Page. The anatomy of a large-scale hypertextual web search engine. *Computer Networks and ISDN Systems*, 30(1-7):107–117, 1998.

[9] D. Canali, A. Lanzi, D. Balzarotti, C. Kruegel, M. Christodorescu, and E. Kirda. A quantitative study of accuracy in system call-based malware detection. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, pages 122–132. ACM, 2012.

[10] Comodo. http://camas.comodo.com, 2013.

[11] S. Forrest, S. A. Hofmeyr, A. Somayaji, and T. A. Longstaff. A sense of self for unix processes. In *Security and Privacy, 1996. Proceedings., 1996 IEEE Symposium on*, pages 120–128. IEEE, 1996.

[12] M. Fredrikson, S. Jha, M. Christodorescu, R. Sailer, and X. Yan. Synthesizing near-optimal malware specifications from suspicious behaviors. *IEEE Symposium on Security and Privacy (S&P)*, pages 45–60, 2010.

[13] K. A. Heller, K. M. Svore, A. D. Keromytis, and S. J. Stolfo. One class support vector machines for detecting anomalous windows registry accesses. In *In Proc. of the workshop on Data Mining for Computer Security*, 2003.

[14] C. Kruegel, D. Mutz, F. Valeur, and G. Vigna. On the detection of anomalous system call arguments. *Computer Security–ESORICS 2003*, pages 326–343, 2003.

[15] A. Lanzi, D. Balzarotti, C. Kruegel, M. Christodorescu, and E. Kirda. Accessminer: using system-centric models for malware protection. In *Proceedings of the 17th ACM conference on Computer and Communications Security (CCS)*, pages 399–412, New York, NY, USA, 2010. ACM.

[16] D. Lerche and P. B. Sørensen. Evaluation of the ranking probabilities for partial orders based on random linear extensions. *Chemosphere*, 53(8):981–992, 2003.

[17] B. C. Mark Russinovich. Process monitor, 2013. http://technet.microsoft.com/en-us/sysinternals/bb896645.

[18] L. Martignoni, E. Stinson, M. Fredrikson, S. Jha, and J. Mitchell. A layered architecture for detecting malicious behaviors. In *Recent Advances in Intrusion Detection (RAID)*, volume 5230, pages 78–97. Springer Berlin / Heidelberg, 2008.

[19] M. Newman. *Networks: an introduction*. Oxford University Press, 2010.

[20] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, et al. Scikit-learn: Machine learning in python. *The Journal of Machine Learning Research*, 12:2825–2830, 2011.

[21] M. Sahinoglu. Security meter: A practical decision-tree model to quantify risk. *IEEE Security and Privacy*, 3(3):18–24, 2005.

[22] W. Sun, R. Sekar, G. Poothia, and T. Karandikar. Practical proactive integrity preservation: A basis for malware defense. *IEEE Symposium on Security and Privacy (S&P)*, pages 248–262, 2008.

[23] W.-K. Sze and R. Sekar. A portable user-level approach for system-wide integrity protection. In *Proceedings of the 29th Annual Computer Security Applications Conference*, pages 219–228. ACM, 2013.

[24] H. Tong, B. A. Prakash, C. Tsourakakis, T. Eliassi-Rad, C. Faloutsos, and D. H. Chau. On the vulnerability of large graphs. In *Data Mining (ICDM), 2010 IEEE 10th International Conference on*, pages 1091–1096. IEEE, 2010.

[25] VXHeaven. http://vx.netlux.org/, 2010.

[26] T. Wüchner, M. Ochoa, and A. Pretschner. Malware detection with quantitative data flow graphs. In *Proceedings of the 9th ACM SIGSAC symposium on Information, computer and communications security*. ACM, 2014.

[27] K.-K. Yan, G. Fang, N. Bhardwaj, R. P. Alexander, and M. Gerstein. Comparing genomes to computer operating systems in terms of the topology and evolution of their regulatory control networks. *Proceedings of the National Academy of Sciences*, 2010.

[28] C. Zhai and J. Lafferty. A study of smoothing methods for language models applied to information retrieval. *ACM Transactions on Information Systems (TOIS)*, 22(2):179–214, 2004.

Scalability, Fidelity and Stealth in the DRAKVUF Dynamic Malware Analysis System

Tamas K. Lengyel
University of Connecticut
tklengyel@engr.uconn.edu

Steve Maresca
Zentific, LLC
steve@zentific.com

Bryan D. Payne
Nebula, Inc.
bdpayne@acm.org

George D. Webster
TU Munich
webstergd@sec.in.tum.de

Sebastian Vogl
TU Munich
vogls@sec.in.tum.de

Aggelos Kiayias
University of Athens
aggelos@di.uoa.gr

ABSTRACT

Malware is one of the biggest security threats on the Internet today and deploying effective defensive solutions requires the rapid analysis of a continuously increasing number of malware samples. With the proliferation of metamorphic malware the analysis is further complicated as the efficacy of signature-based static analysis systems is greatly reduced. While dynamic malware analysis is an effective alternative, the approach faces significant challenges as the ever increasing number of samples requiring analysis places a burden on hardware resources. At the same time modern malware can both detect the monitoring environment and hide in unmonitored corners of the system.

In this paper we present *DRAKVUF*, a novel dynamic malware analysis system designed to address these challenges by building on the latest hardware virtualization extensions and the Xen hypervisor. We present a technique for improving stealth by initiating the execution of malware samples without leaving any trace in the analysis machine. We also present novel techniques to eliminate blind-spots created by kernel-mode rootkits by extending the scope of monitoring to include kernel internal functions, and to monitor file-system accesses through the kernel's heap allocations. With extensive tests performed on recent malware samples we show that *DRAKVUF* achieves significant improvements in conserving hardware resources while providing a stealthy, in-depth view into the behavior of modern malware.

Categories and Subject Descriptors

D.4.6 [**Security and Protection**]: Invasive software

General Terms

Dynamic Malware Analysis, Virtual Machine Introspection

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.
ACSAC '14, December 08 - 12 2014, New Orleans, LA, USA
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-3005-3/14/12 $15.00.
http://dx.doi.org/10.1145/2664243.2664252.

1. INTRODUCTION

Over the last decades malware has become a central tool used by criminal organizations to conduct crime over the Internet. Based on the financial implications of the widespread occurrence of security breaches, an in-depth understanding of malware internals is critical for designing and deploying effective defensive solutions. However, the sheer number of malware samples poses a challenge that cannot be matched by manual reverse engineering efforts. The use of static analysis to create signatures for fast detection has been effectively undercut by the increasingly metamorphic nature of modern malware which enables the restructuring of the binary between infections [24]. In conjunction with anti-debugging techniques, the time it takes to formulate effective solutions is further exacerbated [4, 13].

Since the proliferation of metamorphic malware, dynamic malware analysis has become an effective approach to understand and categorize malware by observing the execution of the malware sample in a quarantined environment [11,37]. The interaction between the executing malware sample and the host OS allows dynamic malware analysis systems to collect behavioral characteristics that aid in formulating defensive steps. Consequently, dynamic malware analysis relies on the *breadth* and *fidelity* of the collected data.

As dynamic malware analysis systems have become widely deployed, malware has evolved to detect and evade such systems by either refusing to execute in a sandboxed environment, or by modifying its runtime behavior to lead the analysis system astray [1]. Consequently, it is critical for dynamic malware analysis systems to provide a *stealthy* environment to hide the presence of the data collection from the executing sample [9]. However, even recent analysis systems that meet the stealth requirements [8] have considerable resource requirements and require manual interaction with the malware samples, which constitutes a barrier to scalable, automated dynamic malware analysis.

In this paper we present the *DRAKVUF* ['drɔχvuf] dynamic malware analysis system designed to take advantage of the latest hardware virtualization extensions to provide a *transparent* and *scalable* environment and enable *in-depth* analysis of malware samples. By building on the effectiveness of virtualization extensions to hide the monitoring environment, *DRAKVUF* gains a deep insight into the execution of both user- and kernel-land malware. Furthermore, *DRAKVUF* is designed to minimize the resource require-

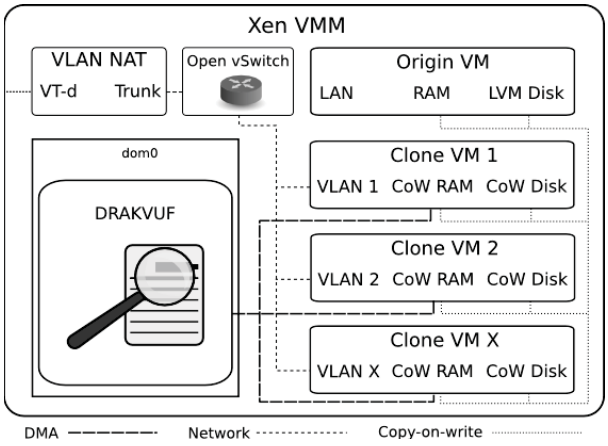


Figure 1: Overview of system components used with *DRAKVUF*.

ments and to enable the fast deployment of virtual machines for analysis by employing copy-on-write techniques for both memory and disk.

Our work makes the following contributions:

- We provide the implementation details of our stealthy and tamper resistant monitoring engine that provides deep insight into the execution of both user- and kernel-mode malware and rootkits. *As a further contribution to the information security community, the source-code is released under GPL at http://drakvuf.com.*
- We introduce a novel technique to directly monitor kernel heap allocations to detect hidden structures created by rootkits. With this novel technique *DRAKVUF* also gains the unique ability to observe filesystem accesses purely via memory introspection.
- We showcase the application of a novel technique to initiate the execution of malware samples without leaving a trace in the analysis environment with the use of *active* virtual machine introspection.
- We evaluate the system using shared hardware resources for large-scale malware analysis via copy-on-write memory and disk.
- We present detailed experiments on a set of in-the-wild malware samples that highlight the capabilities of the system.

The remainder of the paper is organized as follows. In Section 2 we establish our requirements and outline our system's design. In Section 3 we provide in-depth details about our implementation. Section 4 describes the experiments performed using recent, in-the-wild malware samples and also the performance evaluation of the system. Section 5 describes related work. Section 6 describes future work and finally Section 7 provides brief concluding remarks.

2. SYSTEM DESIGN

The goal of *DRAKVUF* is to provide a platform for automated dynamic malware analysis. The analysis has to be performed with transparency and efficiency such that the runtime behavior of the system components in the analysis virtual machines (VMs) can be observed in-depth. *DRAKVUF* is implemented with the use of virtualization technology but in this paper we consider attacks on the hypervisor (VMM) and on lower level system components to be out of scope. It is a standard assumption in virtualization based malware research to consider the VMM to be trustworthy [9], thus in *DRAKVUF* we also consider the limited interface exposed by the VMM to unprivileged guests to provide a reasonable barrier to uncontrolled malware propagation.

The requirements we set out for our system are defined as follows:

(R1) **Scalability**: The performance overhead for analyzing a sample should be kept to a minimum while the ability to concurrently analyse a large corpus of samples should be maximized.

(R2) **Fidelity**: The data collection has to capture a wide scope of runtime information while also providing resistance against tampering such that the state of the analysis system can be accurately examined.

(R3) **Stealth**: The monitored environment should not be able to detect the presence of *DRAKVUF*.

(R4) **Isolation**: *DRAKVUF* should be strongly isolated from the analysis VMs to protect against tampering.

DRAKVUF is implemented with the use of virtualization technology as virtualization provides several benefits that dynamic malware analysis can take advantage of. Creating and reverting analysis containers can be completed more rapidly as compared to imaging and reverting physical machines. Virtualization also provides an avenue to observe malware execution by providing external access to the state of the virtualized hardware components, commonly referred to as virtual machine introspection (VMI).

To utilize these benefits mentioned above, *DRAKVUF* is built on the open-source Xen VMM. The high-level organization of system components is illustrated in Figure 1. To allow fast deployment of analysis VMs, *DRAKVUF* creates full VM clones via Xen's native copy-on-write (CoW) memory interface and the Linux logical volume manager's (LVM) copy-on-write disk capability. The use of full VM clones via CoW enables the optimized use of the hardware as additional resources are only allocated when needed, matching our requirement set forth in **(R1)**. While the static components of the analysis VMs' memory and disk are shared, the use of copy-on-write prevents clone VMs from interacting with each other as they don't have access to exclusive resources allocated to other clones, hence **(R4)** is supported.

DRAKVUF runs in the control domain (dom0) to make use of direct memory access (DMA) through the LibVMI library [22]. Within the control domain *DRAKVUF* also has access to hypervisor features to control virtualization extensions provided by the CPU, such as the Extended Page Tables (EPT). In order to facilitate access to events associated with the execution of the analysis VMs, *DRAKVUF* uses a combination of techniques to trigger a transfer of control to the hypervisor (VMEXIT) when required. The core technique we employ is the use of *breakpoint injection* in which a #BP instruction (`INT3`, instruction opcode 0xCC) is written into the VM's memory at code locations deemed of

interest. The method by which these locations are automatically determined will be further described in Section 3.2. By configuring the CPU to issue a VMEXIT when breakpoints are executed and configuring Xen to forward such events to the control domain, *DRAKVUF* is capable of trapping the execution of any code within the analysis VM. The #BP injection technique has thus far only been used for stealthy debugging [8]. In *DRAKVUF* we are the first to apply the technique for automatic execution tracing of the entire OS and demonstrate how it is a key component in enabling *active* VMI. With this technique *DRAKVUF* gains deep insight into both kernel and user-land code execution, thus matching **(R2)**

In prior research [9] time skew introduced by trapping the guest into the hypervisor has been effectively countered by altering the guest's view of the time stamp counter register (TSC). While additional time sources could still be available, especially if network access is allowed, we consider this problem to be out of scope for this paper. In *DRAKVUF* we address the previously overlooked problem of starting the execution of a malware sample without leaving a trace: up to now the task had to be either performed manually, which hinders scalability [8, 9]; or with the use of in-guest agents that can be detected [5]. *DRAKVUF* addresses this shortcoming by enabling the automatic execution of a sample without the use of in-guest agents, as the presence of such agents could be used to detect the monitoring environment. Instead *DRAKVUF* uses *active* VMI via #BP injection to hijack an arbitrary process within the VM to initiate the start of the malware sample, a technique further described in Section 3.3. By using existing processes running within the VM, *DRAKVUF* does not introduce new code or artifacts into the analysis VM, thus greatly improving stealthiness, set forth in **(R3)**.

As malware is known to use external input and resources to function, providing network access to the analysis VMs is also required. In order to maintain isolation between *DRAKVUF* and the analysis VMs, as set forth in **(R4)**, network traffic passes through a domain running Open vSwitch and exits through a VLAN NAT domain containing a physical network card passed through using Intel VT-d. The clones are placed on separate VLANs, therefore the only network access they have is through the NAT engine which actively prevents the analysis VMs from discovering each other over the local network. This setup is aimed at minimizing the number of components within dom0 that expose an interface to the infected clones, as we consider emulated device backends to be a more likely attack surface as compared to the minimal interface exposed by the VMM.

To summarize and reflect upon our design requirements: **(R1)** is met by the combined use CoW memory and disk, and the use of #BP injection. **(R2)** is met by employing #BP injection at essential OS code locations which directly reveal the state of the runtime environment. By protecting the breakpoints with EPT page permissions and not having any in-guest agents *DRAKVUF* satisfies **(R3)**. Removing the network stack from the control domain and employing VLAN isolation on the clones provides improved isolation, as required by **(R4)**.

3. IMPLEMENTATION

In the following, we provide in-depth implementation details on the techniques used by *DRAKVUF*.

3.1 Scalability

The construction of defensive solutions requires the rapid analysis of malware samples, however, the sheer amount of data that needs to be processed presents a challenge. It is common that AV vendors need to process more than a 100,000 samples a day [14]. Our focus in *DRAKVUF* thus had been to maximize the *throughput* of the physical machine used for the analysis. While the performance overhead imposed on the execution of individual malware samples is also important and should be kept to a minimum, in our opinion such overhead is only of concern if it actively interferes with the analysis. In our system we take advantage of Xen's copy-on-write (CoW) memory feature to limit the memory requirements, and the LVM CoW disk capability to limit the harddrive space requirements on the host machine. With the combined use of these CoW techniques we are able to greatly increase the number of analysis sessions that can be performed concurrently on a given physical machine.

The CoW techniques employed by *DRAKVUF* require the presence of a static domain whose disk and memory can be used as a reference point, referred to as the *origin domain*. The origin domain can be configured as a regular domain before it is cloned, then once a clone is created it remains statically frozen. When cloning is initiated, first the LVM CoW disk is setup, followed by Xen creating the domain for the clone VM. The content of the origin domain's memory is piped into the newly created clone domain, immediately followed by memory de-duplication via memory sharing. After memory sharing, the only memory overhead is the memory allocated for QEMU to provide disk and network I/O to the clone, and the pages where breakpoints are injected. In our tests, the creation of a full VM clone with 2GB of CoW memory and CoW disk was performed in less than 10 seconds.

While such full VM clones have the advantage of requiring the minimal hardware resources for execution, providing concurrent access to the network poses a challenge, as the clones' IP and MAC addresses are identical. When multiple such clones are placed upon the same switch, MAC address collision occurs and the shared IP address prevents NATing of the upstream traffic. Reconfiguring the IP stack within a VM however has traditionally required an in-guest agent, an approach we specifically set forth to avoid with *DRAKVUF*. To overcome this problem, we use Open vSwitch to provide VLAN isolation to the clones by assigning a unique VLAN ID to each clone.

A VLAN NAT is built using the *pcap* library which reads the raw packets directly off the trunk interface and the uplink, gaining access to the Layer 2 network information, bypassing the Linux network stack. With this setup, the VLAN NAT engine gains access to the VLAN ID that uniquely identifies a clone analysis VM. By performing NAT with the extra unique ID, our system successfully bypasses the issue of MAC and IP collisions. Additionally, as we don't use the Linux networking stack, we also implemented ARP reply injection into the VLAN of each analysis VM when a request is issued for the preconfigured default gateway. By dynamically injecting ARP replies we avoid having to set a static ARP entry in the analysis VMs. This eliminates the possibility of the static ARP entry reveiling the analysis environment, while also opens the door for some novel analysis scenarios which we will discuss in Section 6.

3.2 Monitoring

As dynamic malware analysis relies on observing the live execution of malware, the *fidelity* of the collected data is essential. Furthermore, as rootkits employ a variety of techniques to hide their presence on a system, the broader the scope of data collection, the more likely the analysis will reveal useful features. In the following we discuss a set of data collection mechanisms implemented for 32-bit and 64-bit versions of Windows 7 SP1. In our current prototype system we focus on Windows but our system could easily be extended to monitor Linux and other operating systems as well, as the underlying monitoring techniques are OS agnostic.

3.2.1 Execution tracing

A key feature of existing dynamic malware analysis systems is the ability to trace the execution of processes by monitoring system calls. However, monitoring *only* system calls limits the execution trace to the interaction between user-space programs and the kernel, which does not include the execution of kernel-mode rootkits. To overcome this issue, in *DRAKVUF* we took an alternative approach by directly trapping *internal* kernel functions via #BP injection. With direct trapping *DRAKVUF* is able to monitor malicious drivers as well as rootkits, which was previously not possible with just system call interception.

The location of the kernel functions are determined by extracting information from the debug data provided for the kernel. The use of debugging information has been an established method in the forensics community and it is the most convenient avenue to gain insight into the state of the operating system. In *DRAKVUF* we make use of the Rekall forensics tool [28] to parse the debug data provided by Microsoft to establish a map of internal kernel functions.

At runtime, *DRAKVUF* locates the kernel automatically in memory without having to perform signature based scans, which improves resiliency as compared to existing forensics tools [25,35]. To automatically locate the kernel in memory we make the observation that Windows 7 uses the FS and GS registers to store a kernel virtual address pointing to the `_KPCR` structure, which is always loaded into a fixed relative virtual address (RVA) within the kernel, identified by the `KiInitialPCR` symbol. As we have obtained the RVA for all kernel symbols, including `KiInitialPCR`, we only have to subtract the known RVA of the symbol from the address found in the vCPU register to obtain the kernel base address.

Once the kernel base address is established, *DRAKVUF* can trap all kernel functions via #BP injection. With internal kernel functions being trapped, the logs thus generated provide a full trace of the execution of the OS from the moment the malware sample is executed.

3.2.2 Tackling DKOM attacks

Rootkits notoriously modify internal kernel structures to hide their presence on a system, commonly referred to as direct kernel object manipulation (DKOM).Standard DKOM attacks are performed by unhooking structures from kernel linked-lists (like the running process' list), which effectively prevents tools that use these lists to enumerate the structures from discovering the additional elements. Forensics tools have long discovered that objects within the Windows kernel heap are created with an additional header attached (`_POOL_HEADER`). This header contains a four-character description of the structure that can be used to detect unhooked structures by simply performing a brute-force string search for these tags in physical memory.

As pooltag scanning became a standard approach in forensics, malware is known to attempt to overwrite the header to prevent scanning tools from later finding these structures [10]. Other rootkit techniques hide the structures by disconnecting the allocated object from its header by changing the requested object size to be greater than 4096 KB, as such allocation requests result in the object being placed into the *big page pool* where no such header is attached to the object. This technique effectively prevents basic pool tag scanning routines from fingerprinting the object.

However, having access to these internal kernel structures is critical in understanding the runtime state of the system, therefore in *DRAKVUF* we took a new approach to tackle DKOM attacks. As we see from the description of the attack methods, the root cause of the problem with DKOM attacks is that the location of the structures becomes unknown within the kernel's heap. If the location can be accurately determined, DKOM is effectively defeated [29]. With *DRAKVUF* we track the kernel heap allocations directly with #BP injection at internal Windows kernel functions responsible for allocating memory for structures used by Windows: `ExAllocatePoolWithTag` and `ObCreateObject`. *DRAKVUF* dynamically extracts the return address from the stack of the calling thread at function entry and traps it to catch the event when the allocation routine returns. Monitoring heap allocations allows us to detect the location of all kernel structures without malware being able to tamper with our view into the system.

In the current implementation *DRAKVUF* tracks the allocation of all objects on the kernel heap, and generates logs based on the associated tag of the structure. If the tag of the structure is one of the already known 2,254 tags, the log contains further details about the type of the object to aid the analyst in identifying allocations that may be of further interest. To detect for example a hidden process, an analyst can now apply a cross-view check to determine if the allocated structures are also accessible via standard lists [16]. *DRAKVUF* further traps the routines responsible for freeing these structures, thus providing a full-view into the life-cycle of the structures. In the next section we further illustrate how this approach enables us to track the active usage of `_FILE_OBJECT`s

3.2.3 Monitoring filesystem accesses with memory events

Monitoring filesystem accesses is one of the core feature of any dynamic malware analysis system, however, prior agentless VMI approaches have attempted to monitor filesystem accesses by modifying the disk emulator to intercept events [26]. While such an approach is effective, reconstructing high-level file-system accesses (like path and permissions) from the low-level disk-emulation perspective is in itself a form of the semantic gap problem and requires extensive knowledge of file-system internals. However, the internal kernel structures that *DRAKVUF* tracks reveal highly valuable information about the execution state of the system, such as the complete set of running processes, kernel modules, threads, and even objects allocated for filesystem accesses by the OS.

The process by which we catch filesystem accesses is shown

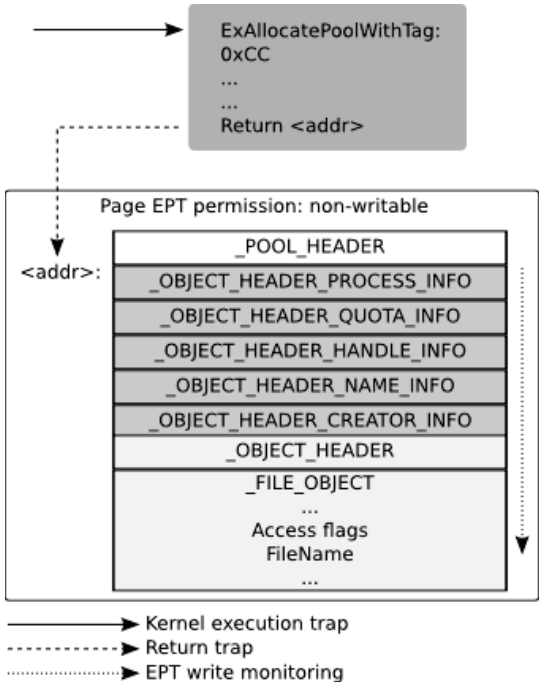


Figure 2: Tracking file accesses by monitoring the allocation of `_FILE_OBJECT`s in the Windows kernel heap.

in Figure 2. When a file is accessed, either by the OS or by a user-land process, a `_FILE_OBJECT` is allocated within the kernel heap with the accompanying tag, `"Fil\xe5"`. When the allocation address is caught, we mark the page on which the structure is allocated as non-writable in the EPT. As the `_FILE_OBJECT` is preceded by a set of optional object headers (shown with a gray background), we derive the exact location of the access flags and file name by subtracting the known size of the `_FILE_OBJECT` from the end of the heap allocation. This allows us to determine the full path of the file as well as the access privilege with which the file is accessed, such as read, write and/or delete permission, without the need to have any deeper understanding of the filesystem itself.

3.2.4 Carving deleted files from memory

A common feature of malware droppers is the rapid creation and deletion of temporary files used during the infection process [3]. These temporary files can potentially contain the unpacked malware binary before it is loaded into memory, or other forensically relevant information. However, malware authors are well aware of this fact and it is standard procedure to clean up the temporary files after the dropper finishes installing the malware.

Existing malware analysis systems implemented with the use of in-guest agents can simply retrieve these files when file-deletion is initiated. From a VMI perspective, retrieving these temporary files is complicated by the fact that Windows defaults to write caching being enabled on all harddrives. When files are created and destroyed rapidly, as is often the case when malware is being dropped on a system, the files are never written to disk. As a result simply mounting the analysis VM's disk in the control domain would not give access to these files and the only possible scenario is to carve the files directly from memory.

In *DRAKVUF*, the carving of deleted files is implemented by intercepting specific internal kernel calls that are responsible for file deletion, such as the `NtSetInformationFile` and `ZwSetInformationFile` routines. Once the functions are intercepted, the file is identified by examining the arguments passed, among them the file handle information. This handle does not point directly to the file object, it is only a reference number to an entry in the handle table of the owning process. By parsing the handle table of the process we can locate the corresponding `_FILE_OBJECT` and automatically carve it from memory with Volatility [35].

3.3 Stealth

The stealth of virtualization based analysis systems has been commonly considered in the context of the detection of the monitoring environment itself and not the detection of virtualization in general. The argument for this division is that the already wide-scale deployment of virtualization in commodity systems creates an economic incentive for malware not to exclude virtualized systems. With these assumptions in place *DRAKVUF* expands upon prior research that showed that the combined use of breakpoint injection with EPT protection is an effective technique to hide from even the most advanced anti-debugging techniques [8]. Time skew introduced by trapping the guest into the hypervisor has also been effectively countered in prior research by altering the guest's access to hardware time sources [9], although external time sources could still be available to the malware if network access is allowed.

In *DRAKVUF*, we have turned our attention to a critical step so far overlooked in automated dynamic malware analysis: we start the execution of the malware sample without leaving an identifiable trace of the monitoring environment. With systems where an in-guest component is used, the execution can be initiated by the monitoring agent itself, but the same in-guest component could be potentially used to detect monitoring, even if it is only an autostart script. On the other hand, when no in-guest agent is present, the sample has to be started manually. Therefore, in order to avoid creating any artifacts within the analysis VMs but to allow automated execution, we implemented an injection mechanism that hijacks an actively running but arbitrary process within the VM to initiate the start of the sample on our behalf.

On Windows, a new process can be created by any user-space application via the CreateProcessA function, which is part of the standard Windows API exposed by the kernel32.dll library provided by the operating system. While not every application on Windows has kernel32.dll loaded, generally only a view system processes are the exception, thus in practice *DRAKVUF* can hijack any normal application.

The injection mechanism relies on a set of events, shown in Figure 3, to successfully hijack a process without causing system instability or altering the state of the machine in a way that would reveal the monitoring environment. As the first step after the clone analysis VM is created, *DRAKVUF* traps write events that happen to the control register CR3 to catch when a process context switch occurs. When an event is caught, we examine what libraries are loaded in the address space of the now running process by walking the

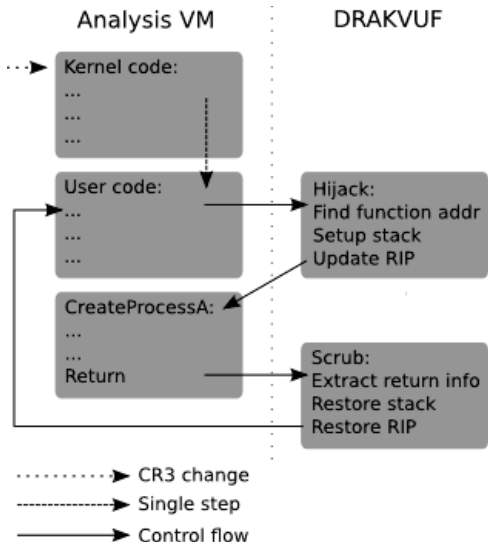


Figure 3: Process hijacking employed in *DRAKVUF* to externally start the execution of the malware sample.

list of loaded modules within the process. If the process has kernel32.dll loaded into its address space, the execution of the VM is switched into single-step mode until the process starts executing user-level code.

The hijack mechanism takes over the execution at the first instruction executed in user-level, and locates the CreateProcessA routine in kernel32.dll's export table. The stack of the process is updated to contain the input arguments required for calling the function, as well as the RCX, RDX, R8 and R9 registers on x86_64, while the original content of the RIP register is pushed as the return address on the stack. The return address is further injected with a breakpoint to notify us when the routine finished. The execution of the analysis VM is resumed after placing the address of CreateProcessA in the RIP register.

When the return breakpoint is hit, we can determine if the process has been successfully created by examining the RAX register, and if it was successful, we also obtain the PID and the handle information of the process that will be used by the executing malware sample. As a context switch could occur while CreateProcessA is executing, the return trap checks if the process at the return trap is the one that was hijacked. Before resuming the original execution of the hijacked process, the stack and vCPU registers are restored, thus seamlessly resuming the execution of the process.

In our implementation we use this mechanism to start malware samples in clean virtual machines. By using this hijacking routine, no artifacts are left on the system that could be detected as a fingerprint of the monitoring environment. This property may not hold if a malware sample is started in an already infected machine, a use-case that we considered out-of-scope for our implementation.

4. EXPERIMENTAL RESULTS

In the following, we discuss an extensive set of experiments performed to establish performance metrics and to evaluate the effectiveness and throughput of the systems. The experiments were performed on an Intel Second Generation i7-2600 CPU running at 3.4GHz.

Unless specified otherwise, the samples were executed on a Windows 7 SP1 x64 analysis platform with a runtime of 60 seconds. During these tests *DRAKVUF* was set to monitor the execution of each internal kernel function that starts with Nt or Zw. The functions starting with Nt are the functions available to user-space applications through regular system calls as listed in the SSDT and monitoring the Zw version of these functions reveals the execution of kernel-level code. We also trap two additional kernel internal-functions, `ExAllocatePoolWithTag` and `ObCreateObject`, which are responsible for kernel heap allocations, as described in Section 3.2.2. While monitoring all internal functions would have been possible during these tests, we reduced the scope of tracing as to reduce the verbosity of the collected data without hindering the insight into the execution of the system.

4.1 TDL4

The first sample we tested was TDL4[1]. This sample was chosen since an in-depth technical write-up has been already created by an antivirus company after reverse engineering the sample [15], which provides a contrast to our automatic analysis. The dropper itself had a 45/46 detection ratio on VirusTotal (VT) [33]. After executing the sample in a Windows 7 SP1 x64 analysis VM, we obtained two additional temporary files created by the dropper in the "C:\Windows\System32\sysrep" folder: cryptbase.dll[2] and syssetup.dll[3].

These temporary files were carved from memory as they were created by the dropper and never flushed to disk before deletion. After submitting the files to VT, the detection ratios were reported as 19/50 and 22/50 respectively. Further investigation into the nature of these temporary files, unmentioned in the original analysis report, revealed them as being part of a known method to elevate privileges by circumventing user access control (UAC) on Windows 7 and 8 [17]. After the exploit installed its payload, a system shutdown was initiated, at which point 1.1GB of memory out of the 2GB assigned to the VM remained shared.

4.2 Zeus

The next sample we analyzed was a recent sample of Zeus[4] with a VT detection ratio of 44/50. In our analysis no files were deleted. However, in our logs we see the sample interacting with files in a temporary folder, GoogleUpdate.exe having a VT detection ratio of 43/50, and FlashPlayer.exe which had no VT detection and was identified as a real Adobe installer. A DLL was also located in the temp folder, msimg32.dll[5], with a VT detection ratio 25/47 that has never been submitted before. The DLL was later revealed by our logs to be dropped into "\Windows\System32" by the executing flash player installer. The sample's installation behavior very closely follows that of ZeroAccess [39], also analyzed by HP [30]. After executing the sample for 60 seconds, the analysis VM had 1.4GB memory in shared state.

[1] md5sum a1b3e59ae17ba6f940afaf86485e5907
[2] md5sum d39d6893117cf1a80c77de1f7ff3d944
[3] md5sum 9b6e5b9c0deb825d5aed343beb090853
[4] md5sum ea039a854d20d7734c5add48f1a51c34
[5] md5sum e051308c2f0c1b280514c99aabd36e34

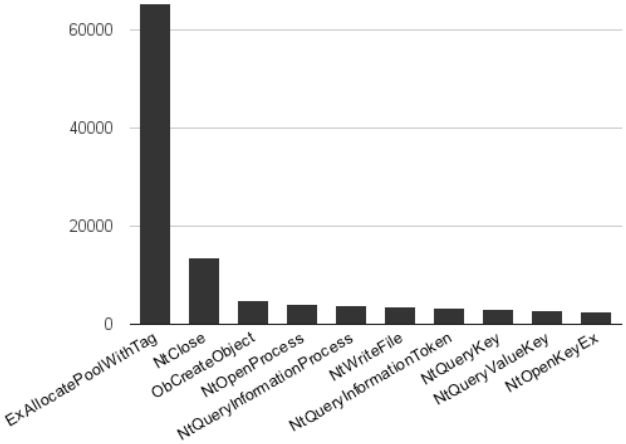


Figure 4: Top 10 monitored kernel functions in terms of average number of observed executions.

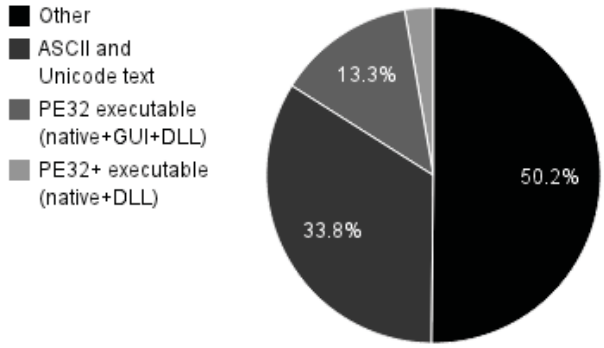


Figure 5: Breakdown of 8797 intercepted file deletion requests by type in recent malware samples.

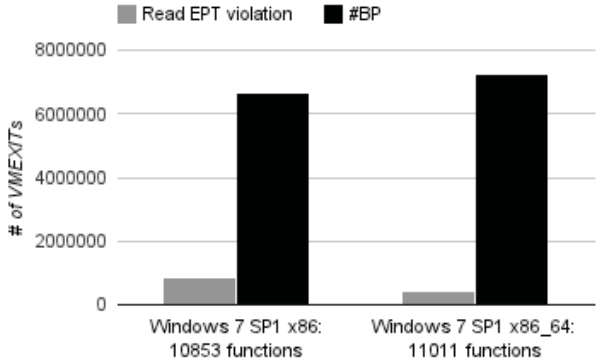


Figure 6: Number of Read EPT violations versus breakpoints hit within 60 seconds when trapping all internal kernel functions.

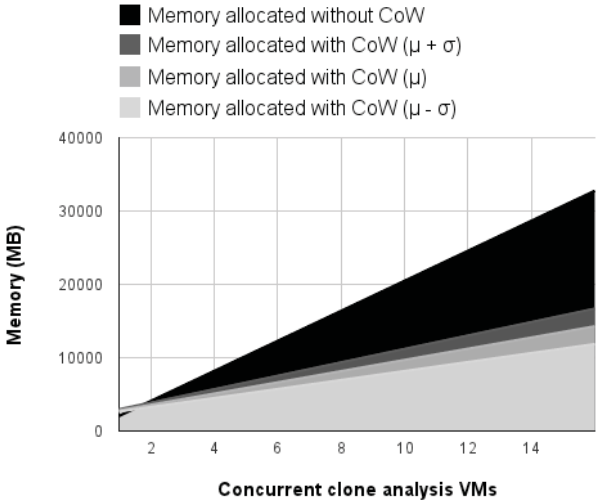


Figure 7: Projected copy-on-write memory allocations (μ = 764MB, σ = 151MB).

4.3 Samples from ShadowServer

In order to test *DRAKVUF* at scale, we obtained 1000 recent malware samples from ShadowServer [32]. The samples were selected with the AlienVault YARA signature that implies the use of anti-virtualization techniques.

Out of the 1000 samples, 241 failed to execute via CreateProcessA injection. These executions failed not because the malware shut itself down but because Windows failed to execute the samples. On average, 159,222 breakpoints had been hit in 60 seconds of execution, and on average 67,950 were pool allocation requests. Figure 4 illustrates the top 10 API calls that were hit across all samples with the internal functions used for heap allocations among the top three. From this figure we can see that some internal kernel functions are executed significantly more often, thus allowing the analyst to pick and choose those functions that are required for the analysis goal, as we did for these experiments.

A total of 8797 unique files were extracted from the analysis VMs by carving them from memory before deletion. To better understand the nature of the deleted files in the analysis VMs, we categorized the files by their type, shown in Figure 5. The 1,412 PE files were also submitted to VT. 561 were new submissions (39%), and nearly all of the files had at least one AV detection. On average, only 20.4% of the AVs categorized these files as malware.

4.4 Measuring overhead and throughput

In this section, we turn our attention to measuring the overhead of our system, a standard metric calculated for any monitoring engine, and to measuring the effective throughput we achieved.

To illustrate the source of the overhead in our system, we measured how many VMEXITs are triggered by our monitoring when all internal kernel functions are trapped. By using the debug symbols for the Windows 7 SP1 kernel all 10,853 internal functions for x86, and 11,011 functions

for x86_64 were trapped. The traps were further protected with EPT execute-only permissions, which on x86 resulted in trapping 727 pages, and 915 pages on x86_64, respectively. The VM was unpaused for 60 seconds and was running idle with only the default Windows system processes being active. As we can see in Figure 6, breakpoints were the main source of VMEXITs we triggered in the VM for both versions of Windows.

To measure the overhead in terms of CPU cycles by the VMEXITs caused by #BP injection, we performed a benchmark similar to that in SPIDER [8]. The test consisted of the monitored domain calling a function within a loop where the function increments a counter. The loop iterates from 10^4 to 10^6 with a step value of 10^4. Timing information was collected by reading the TSC register before and after the loop. The benchmark was performed with and without trapping at the function entry point to determine the overhead. The overhead thus measured was on average a factor of 10502, which is comparable to the overhead in SPIDER. The difference in the overhead likely arises from using different hypervisors, Xen in *DRAKVUF* and KVM in SPIDER respectively, as by design Xen requires an extra VMENTRY/VMEXIT for each trap that is forwarded to the control domain.

Both of these experiments showed that our approach imposes acceptable overhead on the analysis of a single sample. In our opinion, overhead on a per sample base is mainly a *stealth* concern for dynamic malware analsys. That is, overhead should not reveal the presence of the monitoring environment.

From a *scalability* perspective the throughput of the system is of particular importance, which we further evaluated based on the experiments described in Section 4.3. The main hardware constraint in our system was the amount of available RAM: 16 GB. Considering the standard 2 GB of RAM recommended for running Windows 7, the maximum number of concurrent sessions with existing open source tools [5] would be limited to 8 sessions (not counting memory allocated for the control domain). In our experiments, we achieved on average an effective memory saving of 62.4% by using copy-on-write memory, with a standard deviation of 7.3%. Projecting the memory savings as concurrent sessions, shown in Figure 7, we can see the immediate memory savings that can be achieved via CoW memory (the area shown in black). In our case, the number of concurrent analysis sessions has improved by a factor of two, a significant improvement in throughput.

5. RELATED WORK

CWSandbox [37] was one the of first dynamic malware analysis systems to utilize a sandbox environment for monitoring the interaction between the OS and the malware. CWSandbox operates by injecting a kernel driver into Windows that hooks all exported APIs to intercept the system call performed by user-space programs. Beside being vulnerable to detection and tampering, the system call monitoring interface provided by CWSandbox is insufficient for tracking kernel-mode rootkits. Cuckoo [5] is currently one of the most popular open-source dynamic malware analysis systems which uses the same approach as CWSandbox. Cuckoo supports a wide-range of hypervisors; however, neither Cuckoo nor CWSandbox takes advantage of the VMM beyond isolation, as they both rely on an in-guest agent to perform monitoring of the malware's execution. As no special protection is provided to the in-guest agent from the hypervisor, stealth or tamper resistance cannot be guaranteed, potentially leading to incorrect analysis results. In *DRAKVUF*, we directly addressed these shortcomings by observing kernel internal functions from an external point of view, thus providing a stealthy, in-depth view into the execution of the system.

Virtual machine clones were first deployed effectively in 2005 by Vrable et. al. [36] who implemented a highly scalable honeynet system named Potemkin, also based on Xen. Potemkin solved the scaling issue associated with running a large number of nearly identical VMs by introducing memory sharing that de-duplicates identical memory pages present across VMs. While Potemkin was limited to paravirtualized (PV) Linux systems, later works, such as SnowFlock [19], support fully virtualized (HVM) systems as well. As of the latest version of Xen, Potemkin-style VM cloning of fully virtualized systems is natively supported, although only in a "technology preview" stage. Our contribution to existing work in this area is the creation of the network stack via automatic VLAN tagging and NATing such that the clone domains' MAC and IP addresses don't require reconfiguration. In prior work, such reconfiguration has been performed via in-guest agents or by manual iptables configuration [21], a problem we have successfully solved in a transparent and automatic manner.

Ether [9] also made use of hardware virtualization extensions to perform virtual machine introspection. Ether, built on a modified version of Xen, exploited the fact that pagefaults can be configured to trigger VMEXITs on specific code locations. Ether used this to effectively trace system calls in the observed VM. While Ether has made significant efforts to hide these modifications, its effective stealth in practice has been called into question by Pek et. al. [27] who pointed out implementation issues that may still reveal the presence of Ether to the guest OS. In *DRAKVUF*, we also build on the virtualization extensions; however, the techniques employed in *DRAKVUF* to trace system execution allow for analyzing kernel-mode rootkits as well, which were invisible to Ether's system call tracing method. Furthermore, in *DRAKVUF*, we also addressed scalability of the analysis and stealth concerns regarding starting the malware sample without leaving a trace. As an added benefit, unlike Ether, *DRAKVUF* does not require any modification to be made to Xen.

CXPInspector [38] builds exclusively on the use of EPT for execution tracing. As EPT adds an additional layer of paging where translation between guest physical and machine physical address takes place, any access violation within this layer will trigger a VMEXIT, thus remaining transparent to the guest. By marking certain pages non-executable, CXPinspector is able to monitor the execution of the VM. However, the use of EPT for execution tracing has been known to still add significant overhead to the execution of the VM, mainly as a result of the granularity EPT can be set to trigger violations on. In *DRAKVUF*, we also make use of EPT page permissions, but we use it only to add protection to our injected traps, thus avoiding the performance overhead of multiple execute violations on the page.

Process implanting (PI) [12], kernel-module injection (X-TIER) [34] and process hijacking (SYRINGE) [7] have been proposed in prior art as methods to perform introspection

into the state of virtual machines. For such introspection to be secure, extra protection needs to be provided by the hypervisor to deny the runtime modification of the stack and heap data of the implanted code. SYRINGE for example executes the injected function call atomically within the guest. While such methods can ensure the integrity of the injected code, they cannot ensure the integrity of the data that is provided to them by a potentially compromised OS. Therefore, if an attacker provides false information to the monitoring application, by performing for example DKOM attacks, these techniques would be unable to detect the tampering. In *DRAKVUF*, we implemented process hijacking for Windows for the first time and we use it to enable the stealthy execution of malware samples only, thus avoiding any issues that arise from relying on untrustworthy data within the observed OS.

SPIDER [8] evaluated the use of #BP injection to enable stealthy debugging of processes within virtual machines using the KVM hypervisor. However, starting a malware sample with SPIDER still requires either manual action or an in-guest agent, which either prohibits scalability or hinders the stealth capability of the system. Similarly, determining where to place breakpoints with SPIDER is a manual process, as would normally be done with debugging. Our contribution in *DRAKVUF* is applying the core technique of #BP injection to automatically trace the execution of the OS and to extract forensically relevant high-level state information without any manual action being required.

6. FUTURE WORK

While dynamic malware analysis systems can lift the burden of analyzing the ever-increasing number of samples, the inherent limitation of the approach is that the execution of the samples is indeterministic [2]. Whether a malware sample exhibits its true malicious nature (i.e., unpacks itself or just stalls) in a given period has generally been considered undecidable [31], although recent work on the topic raised hopes that the problem could one day be decidable if space and time constrains are taken into consideration [6]. Possible solutions that can be integrated into *DRAKVUF* in the future are multi-path exploration via aggressive snapshotting of the analysis VMs [23] as well as stall prevention via VMI based flow-control changes [18].

Having access to layer 2 network information in our VLAN NAT engine opens the door for new malware evaluation scenarios. While in our current setup the routing engine only injects an ARP reply if the request is for the default gateway, we could also inject an ARP reply to any local IP requested. By further integrating the routing engine with *DRAKVUF*, we could deploy a new analysis VM to construct dynamic LAN environments, providing an easy setup to test how malware would propagate on the LAN.

Memory sharing of the analysis VM is currently performed before the execution of the malware sample is initiated, therefore memory savings gradually decrease as the analysis VM is executing. However, more aggressive memory sharing can be performed by trapping updates made to the guest page tables and evaluating which physical pages can be safely deduplicated. This approach would also enable us to reboot the analysis VM, which in some cases is required to trigger the execution of an infection, without giving up on the advantages of memory sharing.

Our process injection mechanism focused on starting the execution of a malware sample by placing the samples on disk before the clone VMs are created. While this approach was practical in our prototype, in future work we plan to place the malware binary directly in memory by employing process hollowing techniques [20].

DRAKVUF currently follows a single path of the malware's execution. As it is possible that the malware is looking for certain conditions in order to execute or exhibit its malicious behavior, by monitoring the checks that are performed by the sample could be used to potentially explore multiple execution paths [23].

7. CONCLUSION

In this paper, we presented *DRAKVUF*, a novel dynamic malware analysis system. As part of *DRAKVUF*, we introduced several novel techniques to improve the *scalability*, *fidelity* and *stealth* of malware analysis by building on the latest hypervisor technology and hardware virtualization extensions. By monitoring kernel-internal functions *DRAKVUF* gains a complete view into the behaviour of both user- and kernel-land malware. Using active VMI to hijack an arbitrary processes within the analysis VM enables *DRAKVUF* to initiate the execution of malware samples without leaving a trace. *DRAKVUF* also showcases the unique ability to trace filesystem accesses by memory introspection and to extract temporary files never flushed to disk. In our experiments, we evaluated *DRAKVUF* using a wide scope of in-the-wild malware samples. These experiments showed that the system is both effective in analyzing modern malware and capable of improving hardware utilization by maximizing the number of concurrent analysis sessions running on shared physical hardware.

Acknowledgements

The authors would like to thank Peiter "Mudge" Zatko for the DARPA Cyber Fast Track program which made this project possible. Further thanks to Jonas Pfoh for the many productive discussions, to Borbala Mahr for naming the project and to the many anonymous reviewers for their insightful observations. The last author was partly supported by ERC project CODAMODA.

8. REFERENCES

[1] D. Balzarotti, M. Cova, C. Karlberger, E. Kirda, C. Kruegel, and G. Vigna. Efficient detection of split personalities in malware. In *NDSS*, 2010.

[2] U. Bayer, E. Kirda, and C. Kruegel. Improving the efficiency of dynamic malware analysis. In *Proceedings of the 2010 ACM Symposium on Applied Computing*. ACM, 2010.

[3] B. Bencsáth, G. Pék, L. Buttyán, and M. Félegyházi. Duqu: Analysis, detection, and lessons learned. In *ACM European Workshop on System Security (EuroSec)*, volume 2012, 2012.

[4] R. R. Branco, G. N. Barbosa, and P. D. Neto. Scientific but not academical overview of malware anti-debugging, anti-disassembly and anti-vm technologies, 2012.

[5] J. Bremer. Blackhat 2013 workshop: Cuckoo sandbox - open source automated malware analysis. `http://cuckoosandbox.org/2013-07-27-blackhat-las-vegas-2013.html`, 2013.

[6] D. Bueno, K. J. Compton, K. A. Sakallah, and M. Bailey. Detecting traditional packers, decisively. In *Research in Attacks, Intrusions, and Defenses*. Springer, 2013.

[7] M. Carbone, M. Conover, B. Montague, and W. Lee. Secure and robust monitoring of virtual machines through guest-assisted introspection. In *Research in Attacks, Intrusions, and Defenses*, volume 7462 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2012.

[8] Z. Deng, X. Zhang, and D. Xu. Spider: Stealthy binary program instrumentation and debugging via hardware virtualization. In *Proceedings of the 29th Annual Computer Security Applications Conference*, ACSAC '13, New York, NY, USA, 2013. ACM.

[9] A. Dinaburg, P. Royal, M. Sharif, and W. Lee. Ether: malware analysis via hardware virtualization extensions. In *Proceedings of the 15th ACM conference on Computer and communications security*. ACM, 2008.

[10] B. Dolan-Gavitt, A. Srivastava, P. Traynor, and J. Giffin. Robust signatures for kernel data structures. In *Proceedings of the 16th ACM conference on Computer and communications security*. ACM, 2009.

[11] M. Egele, T. Scholte, E. Kirda, and C. Kruegel. A survey on automated dynamic malware-analysis techniques and tools. *ACM Computing Surveys (CSUR)*, 44, 2012.

[12] Z. Gu, Z. Deng, D. Xu, and X. Jiang. Process implanting: A new active introspection framework for virtualization. In *Reliable Distributed Systems (SRDS), 2011 30th IEEE Symposium on*. IEEE, 2011.

[13] F. Guo, P. Ferrie, and T.-C. Chiueh. A study of the packer problem and its solutions. In *Recent Advances in Intrusion Detection*. Springer, 2008.

[14] Z. Hanif, T. Calhoun, and J. Trost. Binarypig: Scalable static binary analysis over hadoop, 2013.

[15] D. Harley. `http://www.welivesecurity.com/2012/02/02/tdl4-reloaded-purple-haze-all-in-my-brain/`, February 3 2014.

[16] X. Jiang, X. Wang, and D. Xu. Stealthy malware detection through vmm-based out-of-the-box semantic view reconstruction. In *Proceedings of the 14th ACM conference on Computer and communications security*. ACM, 2007.

[17] P. Kleissner. The art of bootkit development, 2011.

[18] C. Kolbitsch, E. Kirda, and C. Kruegel. The power of procrastination: detection and mitigation of execution-stalling malicious code. In *Proceedings of the 18th ACM conference on Computer and communications security*. ACM, 2011.

[19] H. A. Lagar-Cavilla, J. A. Whitney, A. M. Scannell, P. Patchin, S. M. Rumble, E. De Lara, M. Brudno, and M. Satyanarayanan. Snowflock: rapid virtual machine cloning for cloud computing. In *Proceedings of the 4th ACM European conference on Computer systems*. ACM, 2009.

[20] J. Leitch. Process hollowing. `http://www.autosectools.com/process-hollowing.pdf`, November 4 2013.

[21] T. K. Lengyel, J. Neumann, S. Maresca, and A. Kiayias. Towards hybrid honeynets via virtual machine introspection and cloning. In *Network and System Security*. Springer, 2013.

[22] LibVMI. `https://code.google.com/p/vmitools`.

[23] A. Moser, C. Kruegel, and E. Kirda. Exploring multiple execution paths for malware analysis. In *Security and Privacy, 2007. SP'07. IEEE Symposium on*. IEEE, 2007.

[24] A. Moser, C. Kruegel, and E. Kirda. Limits of static analysis for malware detection. In *Computer Security Applications Conference, 2007. ACSAC 2007. Twenty-Third Annual*. IEEE, 2007.

[25] J. S. Okolica and G. L. Peterson. Extracting forensic artifacts from windows o/s memory. Technical report, DTIC Document, 2011.

[26] B. D. Payne, M. de Carbone, and W. Lee. Secure and flexible monitoring of virtual machines. In *Computer Security Applications Conference, 2007. ACSAC 2007. Twenty-Third Annual*. IEEE, 2007.

[27] G. Pék, B. Bencsáth, and L. Buttyán. nether: In-guest detection of out-of-the-guest malware analyzers. In *Proceedings of the Fourth European Workshop on System Security*. ACM, 2011.

[28] Rekall. `https://github.com/google/rekall`.

[29] J. Rhee, R. Riley, D. Xu, and X. Jiang. Kernel malware analysis with un-tampered and temporal views of dynamic kernel memory. In *Recent Advances in Intrusion Detection*. Springer, 2010.

[30] A. Roberts, R. McClatchey, S. Liaquat, N. Edwards, and M. Wray. Introducing pathogen: A real-time virtual machine introspection framework. Technical report, HP, 2013.

[31] P. Royal, M. Halpin, D. Dagon, R. Edmonds, and W. Lee. Polyunpack: Automating the hidden-code extraction of unpack-executing malware. In *Computer Security Applications Conference, 2006. ACSAC'06. 22nd Annual*. IEEE, 2006.

[32] ShadowServer. The shadowserver foundation. `https://shadowserver.org`, February 4 2014.

[33] VirusTotal. Free online virus, malware and url scanner. `http://virustotal.com`, February 4 2014.

[34] S. Vogl, F. Kilic, C. Schneider, and C. Eckert. X-tier: Kernel module injection. In *Network and System Security*. Springer, 2013.

[35] Volatility. `https://github.com/volatilityfoundation/volatility`.

[36] M. Vrable, J. Ma, J. Chen, D. Moore, E. Vandekieft, A. C. Snoeren, G. M. Voelker, and S. Savage. Scalability, fidelity, and containment in the potemkin virtual honeyfarm. In *ACM SIGOPS Operating Systems Review*, volume 39. ACM, 2005.

[37] C. Willems, T. Holz, and F. Freiling. Toward automated dynamic malware analysis using cwsandbox. *Security & Privacy, IEEE*, 5(2), 2007.

[38] C. Willems, R. Hund, and T. Holz. Cxpinspector: Hypervisor-based, hardware-assisted system monitoring. Technical report, Ruhr-Universitat Bochum, 2013.

[39] J. Wyke. The zeroaccess rootkit. `http://sophosnews.files.wordpress.com/2012/04/zeroaccess2.pdf`, 2012.

Towards Automated Integrity Protection of C++ Virtual Function Tables in Binary Programs

Robert Gawlik and Thorsten Holz
Horst Görtz Institute for IT-Security (HGI), Ruhr-University Bochum
{firstname.lastname}@rub.de

Abstract

Web browsers are one of the most used, complex, and popular software systems nowadays. They are prone to dangling pointers that result in *use-after-free* vulnerabilites and this is the de-facto way to exploit them. From a technical point of view, an attacker uses a technique called *vtable hijacking* to exploit such bugs. More specifically, she crafts bogus virtual tables and lets a freed C++ object point to it in order to gain control over the program at virtual function call sites.

In this paper, we present a novel approach towards mitigating and detecting such attacks against C++ binary code. We propose a static binary analysis technique to extract virtual function call site information in an automated way. Leveraging this information, we instrument the given binary executable and add runtime policy enforcements to thwart the illegal usage of these call sites. We implemented the proposed techniques in a prototype called T-VIP and successfully hardened three versions of Microsoft's Internet Explorer and Mozilla Firefox. An evaluation with several zero-day exploits demonstrates that our method prevents all of them. Performance benchmarks both on micro and macro level indicate that the overhead is reasonable with about 2.2 %, which is only slightly higher compared to recent compiler-based approaches that address this problem.

1. INTRODUCTION

Within the last decade, we observed a shift in how attackers compromise systems. Instead of targeting server applications such as common Windows network services, adversaries nowadays often launch attacks against client applications. Especially office applications and web browsers are affected, mainly since they are complex software products and thus prone to software defects. Most modern web browsers are developed in C and C++ and consist of several million lines of code. These languages are prone to programming mistakes, which can result in exploitable vulnerabilities hard to spot in such huge programs (both on the source code and on the binary level).

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.
ACSAC '14, December 08 - 12 2014, New Orleans, LA, USA
Copyright 2014 ACM 978-1-4503-3005-3/14/12 ...$15.00
http://dx.doi.org/10.1145/2664243.2664249.

Particular kinds of programming mistakes that are prevalent today result in so called *use-after-free* vulnerabilities, which are often abused by adversaries [2]. In such bugs, a program path exists during which a pointer to an object that was previously freed, is used again. This dangling pointer could cause a program to behave unexpectedly, or an adversary could even execute arbitrary code. Use-after-free software defects are frequently used during targeted attacks. For example, government agencies recently used a zero-day exploit that takes advantages of such vulnerabilities against Mozilla Firefox to identify users of the TOR network to deanonymize suspects [32]. As such vulnerabilities are prevalent in browsers, researchers sucessfully demonstrated exploits against Internet Explorer, Mozilla Firefox, and Google Chrome during the yearly *pwn2own* competition. Furthermore, several recent zero-day exploits for Microsoft's Internet Explorer were based on such use-after-free vulnerabilities [29]. In fact, a recent study suggests that 69 % of all vulnerabilities in browsers and 21 % of all vulnerabilities in operating systems are related to such bugs [11]. To profit from such vulnerabilities in object-oriented code, attackers typically utilize a technique called *vtable hijacking*: If an object of a polymorphic class is freed, but a reference is kept, an adversary can hijack the control flow at virtual function call sites (see §2.3).

Since vtable hijacking attacks are a frequent problem in practice, several compilers recently started to include protection techniques (see §6). Unfortunately, source code is often not available (e.g., proprietary code used in commercial products, third-party libraries, or legacy code). Hence, we present a lightweight approach to provide vtable integrity for COTS binaries implemented in C++ code. We perform our analysis on the binary level since we aim to protect executables for which we do not have source code, debugging symbols, or runtime type information, such as for example web browsers for Windows. To this end, we introduce a generic method to identify virtual call sites in C++ binary code. More specifically, we lift the assembler code to an *intermediate language* (IL) and then perform backward slicing on the IL level such that we can spot different kinds of C++ virtual function dispatches. In a second step, we instrument each virtual call site and add integrity policies. These can check if a vtable pointer points to a writable memory page (which indicates that an integrity violation happened) or if a random chosen vtable entry actually *is* a code pointer.

We implemented our approach in a framework called T-VIP (*Towards Vtable Integrity Protection*) that consists of a slicer called vExtractor and a binary rewriting engine called PeBouncer. Experimental results demonstrate that this tool is capable of mitigating all tested zero-day attacks

against Microsoft's Internet Explorer and Mozilla Firefox, while requiring a performance overhead of about 2.2 %.

In summary, our main contributions in this paper are:

- We introduce an automated method to identify virtual function call sites in C++ binary applications based on an intermediate language and backward slicing. This enables us to determine the potential attack surface for use-after-free vulnerabilities in binary executables implemented in C++.
- We present a generic binary rewriting framework for PE executables with low overhead called PeBouncer that we utilize to implement integrity policies for virtual call sites.
- To the best of our knowledge, we are the first to present virtual table integrity protection for binary C++ code *without* the need for source code, debugging symbols, or runtime type information.
- We show that T-VIP protects against sophisticated, real-world use-after-free remote code execution exploits launched against web browsers, including zero-day exploits against Microsoft Internet Explorer and Mozilla Firefox. A performance evaluation against GCC's virtual table verification feature [44] with micro- and macro-benchmarks demonstrates that our approach introduces a comparable performance overhead.

2. TECHNICAL BACKGROUND

In the following, we briefly review necessary C++ features, discuss how this enables vtable hijacking, and explain why we need an intermediate language to perform our analysis.

2.1 C++ Inheritance and Polymorphism

Inheritance is a general concept in *Object Oriented Programming* (OOP) languages. Data structures called *classes* can contain data *attributes* and functions named *methods*. Working with classes is mostly done on their instances, which are referred to as *objects*. Classes can serve as base classes when they are inherited, creating derived classes, which inherit the base's attributes and methods in addition to their own attributes and functions. Classes can inherit from multiple base classes, and also, derived classes themselves can serve as base classes, such that a (potentially very complex) class hierarchy is created between them.

A programmer can change the functions of base classes inside derived classes by overloading or implementing them. They must be declared as *virtual*, and any class containing virtual functions is a *polymorphic* class (see Figure 1, left). The binding of virtual functions to a class' instance is performed dynamically during runtime if the compiler cannot determine the instance's type statically. Depending on the dynamically determined type of the instance, the appropriate function is selected. For more details, we refer the reader to the literature [6, 42].

When compiling C++ code that contains virtual function dispatches, most compilers generate machine code instructions containing indirect calls. These are interesting constructs for attackers to gain control of the instruction pointer by controlling the call's target register. We elaborate on this attack vector in the following sections.

2.2 Virtual Function Calls

For each class that defines virtual functions, a *virtual function table* (abbr. *vtable*) is created during compile time. It

```
class A{
  virtual int Fn(){..};
};
class B{
  virtual int Fc(){..};
};
/*single inheritance*/
class C: public B{
  virtual int Fc(){..};
};
/*multiple inheritance*/
class D: public A, public B
  { .. };
```

```
/* call of overloaded
virtual function */

C* p = new C();
p->Fc();

① mov R, [p]
② add R, offsetFc
③ mov R, [R]
④ mov  this , p
⑤ call R
```

```
/* call of base's virtual
function */

D* p = new D();
p->Fc();

① mov R, [p+offsetVtB]
② add R, offsetFc
③ mov R, [R]
④ lea  this , [p+offsetVtB]
⑤ call R
```

Figure 1: Example for single and multiple inheritance with polymorphic classes (left). C++ and assembly code of dispatching an overloaded virtual function (middle) and an inherited base class' virtual function dispatch (right). Registers are denoted with *R*.

contains the addresses of all virtual functions that a class provides. During runtime, when an instance is created, a pointer to a vtable is inserted into the instance's layout similar to a data attribute. A class instance's lifetime can involve the usage of several vtables depending on the number of base classes with virtual functions it inherits from.

Figure 1 shows the low level instructions on two virtual function dispatches. At first, a vtable address is loaded into a register (①). Then an indexing offset is added to it to let the register point to the address of the virtual function (②). This is omitted if the virtual function is the vtable's first entry. Afterwards, it is selected by dereferencing the vtable's entry (③) and dispatched with an indirect call (⑤). Additionally, a *this* pointer is created and passed as parameter to the virtual function (④), either via the register `ecx` [30], or as the first parameter. As first parameter, location or registers are used, which are specified in the corresponding calling convention, i.e., the stack or the register `rdi`. The *this* pointer constitutes the instance's address and is adjusted in case of multiple inheritance.

These semantic steps can then be generalized: let *obj* be the address of an instance and i the displacement offset to the vtable pointer at *obj*. The length of the vtable in bytes is indicated as $|vtable|$. Then, on a 32-bit system, $j \in [0, \frac{|vtable|}{4} - 1]$ denotes the index into the vtable, where an address of a virtual function vf resides. A memory dereference is stated with *mem*. Thus, we get:

$$\forall\, vf\ \exists\, mem\ :\ mem(mem(obj + i) + j * 4) = vf \qquad (1)$$

$$\forall\, this\ \exists\, obj\ :\ (obj + i) = this \qquad (2)$$

(1) and (2) holds for virtual functions called indirectly, where (1) comprises steps ① - ③ and ⑤, and (2) describes step ④.

Compilers usually translate calls into these five steps [14]. Highly optimized code, such as modern browser libraries, can omit step ④, combine several steps into single instructions, and have multiple unrelated instructions in between. There are syntactical varieties in steps ① - ③ and ⑤ dependent on optimization levels. However, the manifestation of semantic step ④ into assembly strongly depends on the used compiler and is independent of the optimization (see Table 1). These subtleties were observed in our 32-bit test binaries originating from C++ code with virtual, single, and multiple inheritance and polymorphic classes, as well as in COTS browser code.

While the syntax may differ, virtual function dispatches reveal themselves in generalized semantics, at least in binary code stemming from GCC, LLVM, and MS Visual C++. As

Table 1: Variations of step (4) based on compilers: Variadic virtual functions retrieve the *this* pointer via the stack, either by a push instruction or by stack pointer addressing (FPO). For non-variadic functions, the `ecx` register is used.

Compiler	Passing *this* to virtual function via:	
	non-variadic function	variadic function
GCC (MinGW)	ecx	stack(FPO)
Clang (LLVM)	stack(FPO)	stack(FPO)
MS Visual C++	ecx	stack(push)

Table 2: Disassembly and corresponding semantic steps of virtual dispatches in vulnerable modules. All examples were used in real attacks to gain control of the instruction pointer at step (5) by loading a fake vtable at step (1) first.

Disassembly with Virtual Dispatches	Semantics
`0x612dc754: mov ecx, [eax]`	①
`0x612dc756: push eax`	④
`0x612dc757: call dword [ecx+0x4]`	②③⑤
CVE-2013-1690: xul.dll 17.0.6.4879 @ 0x611d0000	
`0x7167d53e: mov eax, [ebx]`	①
`0x7167d540: and dword [ebp-0x18], 0x0`	
`0x7167d544: lea ecx, [ebp-0x18]`	
`0x7167d547: push ecx`	
`0x7167d548: push dword 0x7167d58c`	
`0x7167d54d: push ebx`	④
`0x7167d54e: mov edi, 0x80004002`	
`0x7167d553: call dword [eax]`	②③⑤
CVE-2013-3897: mshtml.dll 8.0.7601.17514 @ 0x714c0000	
`0x706c3857: mov ecx, [esi]`	④
`0x706c3859: mov edx, [ecx]`	①
`0x706c385b: mov eax, [edx+0xc4]`	②③
`0x706c3861: call eax`	⑤
CVE-2013-3893: mshtml.dll 9.0.8112.16421 @ 0x702b0000	

GCC and Visual C++ are standard compilers for browsers on MS Windows, our framework is able to extract virtual function dispatches from their generated binary code (see §4.1). We refer to the low-level assembly semantics of a C++ virtual function call as *virtual dispatch*, which includes vtable loading, virtual function selection, and passing the *this* pointer as hidden or first parameter to the virtual function. The assembly instruction which performs the indirect call of the virtual function we refer to as *virtual call*.

Recent in-the-wild exploits, including two targeted zero-day attacks against Internet Explorer [29] and one against the Firefox version included in the Tor Browser Bundle [32], remotely executed code by abusing virtual dispatches. Table 2 illustrates the different manifestation of the five semantic steps for the three virtual dispatches, utilized to exploit CVE-2013-3897, CVE-2013-3893, and CVE-2013-1690.

One of our goals is to identify virtual dispatches in a given executable in an automated way. This is a complex task due to the large number of ways to express virtual dispatches in x86 assembly. Thus, we opted to abstract away from the assembler level and perform our analysis based on an *intermediate language* (IL).

2.3 Threat Model: Vtable Hijacking

In the following, we explain the basic stages of use-after-free exploitation, based on Figure 2. It is most commonly used by attackers to execute arbitrary code in-the-wild.

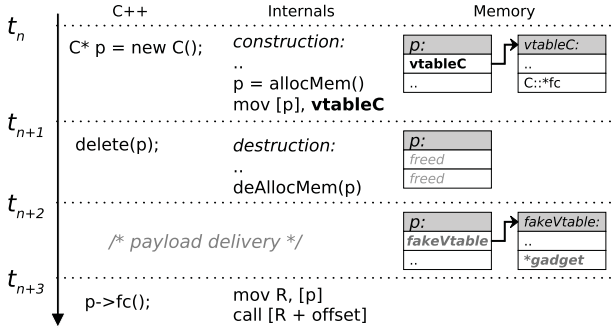


Figure 2: The C++ stages, internal low-level operations, and resulting instance's memory layout of a use-after-free exploitation process utilizing vtable hijacking.

Use-after-free memory corruption vulnerabilities are based on dangling pointers. During runtime, a C++ application requests a new instance of class `C` at time t_n which has implemented a virtual function `fc`. Internally, constructing an instance invokes the memory manager to allocate needed memory. The instance's structure is built, including a vtable of `C`. Furthermore, a pointer `p` to the instance is created. At subsequent execution time t_{n+1}, the instance is removed but the pointer is still kept. If the programmer did not reset the pointer or if an alias was created, the reference to the freed memory still exists. Hence `p`, or the alias becomes dangling.

At time t_{n+2}, an adversary can deliver payload content of her choice to the memory where `p` is pointing to (e.g., via heapspraying [13, 40] and similar techniques), and inject a fake vtable. In practice, this vtable resides in writable memory, whereas a legitimate vtable always resides in non-writable memory. Surprisingly, checking for *non-writable* on vtable pointers during runtime before their usage already prevents many of the in-the-wild exploits as our evaluation shows (see §5.3 for more details).

The fake vtable needs to be carefully crafted to have an entry pointing to the adversary's first chosen chunk of code [12]. Later at t_{n+3}, the virtual function `fc` is dispatched, leading to the instances's pointer `p` dereference, the fake vtable's pointer dereference, and the call of the adversary's code. Thus, this initiates the first step of a code execution attack by retrieving control of the instruction pointer and redirecting the logical program flow.

Note that an adversary can utilize code reuse methods to bypass *no-execute* (NX) protection and may craft memory leaks [37] beforehand to bypass *address space layout randomization* (ASLR). For more details, we refer the reader to the available literature on code reuse techniques [8, 26, 36, 38]. In case *return-oriented programming* is utilized, the adversarie's first chosen code chunk is typically a so called *stack pivot gadget* used to exchange the stack pointer with a controlled register to further redirect program flow to her injected payload [12]. Many recently detected zero day exploits utilize vtable hijacking to exploit vulnerabilities in web browsers as shown in Table 3.

Table 3: Zero-day attacks using vtable hijacking in-the-wild.

CVE	Targeted Application	Module	Vulnerability
2013-1690	Fx 17.0.6 (TorBrowser)	xul.dll	use-after-free
2013-3893	Internet Explorer 9	mshtml.dll	use-after-free
2013-3897	Internet Explorer 8	mshtml.dll	use-after-free
2014-0322	Internet Explorer 10	mshtml.dll	use-after-free
2014-1776	Internet Explorer 8-11	mshtml.dll	use-after-free

3. HIGH-LEVEL OVERVIEW

In the following, we introduce our approach called T-VIP (*Towards Vtable Integrity Protection*). The main idea behind our protection scheme is to mitigate an exploitation attempt at the entry point, meaning, after the loading of a fake vtable pointer, but *before* it is used further to select a virtual function. Thus, the execution of the subsequent virtual call can be stopped, preventing the attacker from obtaining control of the instruction pointer, and impeding successive malicious computations. T-VIP consists of vExtractor, the unit which identifies virtual dispatches in binary code, and PeBouncer, which transforms the original executable in order to allow only legitimate virtual calls to be executed. We give a brief description of each component and present an overview of their interactions.

3.1 Extraction of Virtual Function Dispatches

The first component of T-VIP is vExtractor, a static instruction slicing and extraction framework. It takes an executable as input whose vtable load instructions before virtual calls should be protected, disassembles it and generates a control flow graph (CFG).

The disassembly is then transformed into an intermediate language (IL) to boil down the complex instruction set into a RISC-like syntax, while preserving the semantics and the CFG of the original code. Next, all addresses of indirect call instructions are extracted and defined as slicing criterions [7, 39]. vExtractor then performs backward program slicing on the IL to determine if an indirect call is a *virtual call*. It extracts all instructions which fulfill the low-level semantics of a virtual dispatch, thus, we retrieve virtual dispatch slices.

This is achieved via state machines, whereby one state consists of a set of IL instructions (see §4.1 for details). Furthermore, each state points to at least one successor state. Slicing starts at indirect call sites and the state whose instruction should be found next (in the backward IL instruction stream) is set as target state. If a target state is successfully matched against an instruction in the stream, then its successor is set as target state. As long as the last state of the state machine is not reached or is still satisfiable, slicing continues. It stops either if the last state is reached or if a state cannot be fulfilled. The latter disqualifies the indirect call site as a virtual call. In the former case, vExtractor classifies the call site as part of a virtual dispatch, extracts all instructions which are part of it, and associates its components such as x86 registers, offsets and addresses with instance, vtable and virtual function properties. Most important is the instruction which loads the vtable pointer of the instance into a register, as verification checks will be performed on these registers later on during runtime.

3.2 Protection of Virtual Function Dispatches

The information produced by vExtractor and the original executable are the input to PeBouncer, the second component of T-VIP. We designed PeBouncer as a generic and static binary rewriting engine for executables conforming the PE specification. Thus, while we use it to generate a protected executable, it is suitable to instrument instructions of interest similar to Pin [27] or DynamoRIO [4, 10]. Furthermore, it can be used to insert arbitrary code in order to enhance an executable with defense techniques similar to Vulcan [41] or SecondWrite [34].

A user who wishes to instrument an executable with PeBouncer has to specify the addresses of instructions to instrument. Each of these instructions is replaced by a forward jump redirecting to an instrumentation stub inserted into a newly created section in the executable. The original instruction is preserved by copying it to the beginning of the stub. The stub ends with a backward jump targeting the address after the redirection to continue the original program flow. When replacing instructions, edge cases are considered and measures are taken to rewrite them correctly (see §4.2).

The instrumentation can then implement an integrity policy $\mathscr{P}$ to perform checks on the virtual dispatches. Instrumentation stubs to enforce the policy can be developed in assembly code and can invoke functions of a user generated shared library, which we refer to as *service library*. Hence, major instrumentation code can reside inside the service library in order to remain customizable and still being able to accomplish complex tasks.

Using PeBouncer, we can instrument instructions which load a vtable in virtual function dispatches and generate distinct binaries with different integrity policies. As noted above, a vtable always resides in memory pages which are non-writable such as code or read-only data sections. They contain virtual function pointers pointing to read-only and executable pages. This basic insight can be leveraged to implement a simple integrity policy that checks if vtable pointers correctly point to non-writable memory pages.

The following kinds of integrity policies are possible:

1. $\mathscr{P}_{nw}$: Look up vtable pointers in a lookup table with bits set for non-writable memory pages of modules and unset otherwise. This offers performant validation. When determining the memory protection of a vtable address, the page it belongs to is queried instead of the vtable pointer itself.
2. $\mathscr{P}_{nwa}$: includes $\mathscr{P}_{nw}$. Additionally, one entry inside the vtable residing *above* the virtual function pointer about to be called, is randomly chosen. The entry is dereferenced, and the resulting address is queried for the non-writable flag. This is applied to all virtual dispatches calling a virtual function pointer that is not the first in the vtable.
3. $\mathscr{P}_{obj}$: Leverage type reconstruction of object-oriented code [23] in order to reconstruct all objects and the according class hierarchy. As an integrity check, we could verify if a vtable actually maps at virtual dispatch sites.

Note that $\mathscr{P}_{obj}$ is hard to implement in practice on binary code (in contrast to compiler-level implementation), as object recovery has yet to be shown practicable for huge COTS software like web browsers [23]. Hence, we did not implement this policy as part of our work.

The automated extraction of virtual dispatches can also yield slices not being virtual dispatches. For example, when binary code originated from nested C structs (see §5.1). As such, vExtractor might output instructions which seemingly load a vtable, but in fact represent other kinds of code constructs. This problem can be addressed with a profiling phase as follows: T-VIP first generates an executable instrumented with the checks using policy $\mathscr{P}_{nw}$, and runs it dynamically on tests in a trusted environment to visit (ideally) all instrumentation stubs. Hence, assumed vtables appearing in writable memory are the result of other constructs and are discarded in a second pass: To ensure vtable

integrity protection at runtime, T-VIP applies PeBouncer a second time to the original executable using policies $\mathscr{P}_{nw}$ or $\mathscr{P}_{nwa}$ to produce the final protection.

4. IMPLEMENTATION

We now describe the inner workings of T-VIP: This involves program slicing to retrieve virtual dispatch slices, the architecture of our generic binary rewriting engine and its usage relevant to instrumenting and protecting executables against vtable hijacking.

Currently vExtractor supports the IDA Pro disassembler internally to disassemble an executable and generate a CFG. Based on the disassembly, we search for all call instructions with registers or indirect addressing as operand. Addresses of indirect calls are stored and the disassembly is transformed into the platform-independent intermediate language REIL [15]. We utilize this RISC-like assembly language as IL to transform 32-bit x86 disassembly to an intermediate representation. As typical for RISC, there is only one dedicated memory *load* and memory *write* instruction. As registers, real x86 registers as well as an unlimited number of temporary registers can be used in an interchangeable way. We generalize real registers to R_i and temporary registers to r_j. An undetermined register (i.e., a register that is either R or r), is refered to as q. Any immediate value is denoted with m and operands being either q or m are denoted with v.

As discussed in §2.2, virtual dispatches have a certain semantic which can express itself differently in the x86 syntax. We exploit the advantages of an IL which converts syntactically greatly varying but semantically similar instruction streams into similar constructs. This facilitates the harvesting and classification of semantically similar but syntactically different instruction streams like virtual dispatches via backward slicing.

4.1 Amorphous Slicing

We implemented intra-procedural backward slicing on IL into vExtractor based on state machines. As program slicing is a common technique, we refer the reader to the literature for information about program slicing [7, 20, 39].

In our state machines, one state consists of a set of IL instruction patterns. Figure 3 shows our state design and a state transition. When an IL instruction in the instruction stream to search (①), fulfills a pattern's condition such as matching mnemonic and matching destination register, the state is triggered (②). The source of the matched instruction is taken and inserted into the successor state's instruction patterns as destination (③). A transition to the successor state is performed (④). Slicing continues (⑤), and the patterns of the successor state have to be fulfilled in order to trigger it.

While the semantics of virtual function dispatches are simple, they can manifest themselves in different and complex x86 syntax constructs (as prev. shown in Table 2). To extract virtual dispatches precisely, we developed several state machines to unveil instructions being components of them.

A first state machine is used to search backwards, starting from each indirect call instruction in the IL instruction stream for dereferences. On success, if the final state is reached, vExtractor detects the instance's dereferencing, offset additions, which are optional, and the vtable entry's

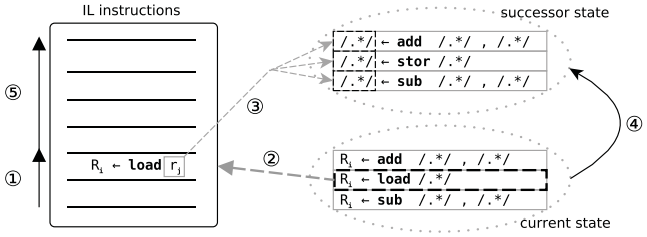


Figure 3: State design and state transition principle. Wildcard operands are denoted with `/.*/`

dereferencing. Thus, one successfull pass can yield the following IL example slice:

State:					
3	r_0	$\leftarrow$	**add**	q_3,	m_1
2	q_2	$\leftarrow$	**load**	r_0	
2	q_1	$\leftarrow$	**add**	q_2,	m_0
1	q_0	$\leftarrow$	**load**	q_1	
0			**call**	q_0	

When extracting the x86 disassembly corresponding to the IL slice, a dereferencing sequence arises, which exists in virtual dispatches when used with multiple inheritance:

mov $q_2, [q_3 + m_1]$
call $[q_2 + m_0]$

q_3 is the instance address (obj), q_2 contains the vtable address, to the virtual function to call.

While the first state machine extracts semantic components related to instance and vtable loading, we designed further state machines to detect a *thiscall*. Furthermore, additional state machines are used to detect if the found instance (obj) is passed as first parameter via the stack to the virtual function at dispatch time. These machines are set up dynamically based on information gained during slicing, to be able to detect more complex virtual dispatches. Due to space constrains, we refer to our technical report about the machinery details [17].

vExtractor uses this state machine approach to harvest virtual dispatch data: it walks the CFG of a binary of interest backwards, starting from indirect calls and tries to match states to IL instructions. On a match, object location and register, vtable register, virtual function offset, and the addresses of corresponding instructions are saved. Additionally, the corresponding disassemblies of virtual dispatches are gained as slices. This virtual dispatch data is then fed into PeBouncer to generate an instrumentation of the vtable load instructions to enforce specific policies.

4.2 Binary Transformations

We developed PeBouncer as a generic and automated instrumentation engine for PE executables. Thus, transformations are applied statically to produce an instrumented binary that realizes an integrity policy. The relevant enforcement checks become active during runtime.

4.2.1 Insertion of Instrumentation Checks

We statically create a new code section where the integrity policy checking code will reside. Addresses of instructions to instrument are fed into PeBouncer and disassembled. As an instruction will be replaced with a redirecting, relative 32-bit jump to its instrumentation stub, we need enough space for the jump opcodes, while keeping other instructions in

the neighborhood functional. Therefore, if the size of the original instruction's opcodes is greater than the jump, we replace it and insert NOPs to fill the remaining space. If the size is smaller than the size of the jump, we disassemble downwards starting from the original instruction until there is at least enough space to insert our jump. Thus, we replace the instruction to instrument and subsequent instructions. The replaced instruction(s) will be copied to the beginning of the associated instrumentation stub into the new section.

Some instructions such as relocations or relative control transfer targets cannot be overwritten without additional measures. To tackle these and similar corner cases, we stop our downward search for space at such instructions and traverse the disassembly upwards instead, starting from the instruction to instrument. Similar to the downward search, we stop as soon as we have enough space to insert our redirecting jump, while overwriting additional instructions if necessary. If the instruction to instrument is *enclosed* between two instructions of above mentioned edge cases, we overwrite one of them with an illegal instruction and install a vectored exception handler. It then serves as a trampoline to the instrumentation stub.

For each instruction to instrument, the redirecting jump's target address is automatically calculated to point to the next available free location in the new section. The replaced instructions are copied there, and the instrumentation code is placed below them into the stub. At the end of the stub, a relative 32-bit jump is inserted. Its target address is calculated to point to the original code, to the address right after the redirecting jump. Thus, the new code section is filled successively, stub after stub. vExtractor provides virtual dispatch slices including the addresses of vtable load instructions. For a C++ executable, we utilize these addresses to instrument and to protect them.

4.2.2 Generation of Instrumentation Stubs

For instructions of interest, instrumentation stubs can be supplied in position-independent assembly. Hence, relative addressing can be utilized. A stub starts with a prolog to save the register context and ends with an epilogue to restore it. The Netwide Assembler (NASM) is used as assembly backend. We created an annotation feature that serves PeBouncer as hint to modify the stub *after* it is assembled, but *before* it is inserted into our new code section. This allows one-time assembling of instrumentation code and many-time stubwise modification.

It works as follows: Instrumentation code is provided as assembly file and contains specific keywords inside angle brackets, which PeBouncer recognizes. The brackets including the annotation keywords are replaced with a x86 mnemonic and a hash of the keyword as operand. Depending on the keyword, corresponding mnemonics which allow at least a four byte operand are used. This way, the assembly syntax stays error free, and the keyword's information is preserved. Also, the occurrence of each keyword is counted. After the instrumentation code is assembled, it contains the binary representation of the hashes. Before the binary instrumentation code is about to be inserted as a stub, the hashes are searched and their occurrences are compared to the keywords' occurrences to prevent collisions. Then they are replaced with adjusted opcodes specific for a keyword and specific for an instrumentation stub. The reader may ask what benefit it has. We instrument instructions representing vtable loading and each of these may use a different mnemonic and a different register to load the vtable. With our annotation feature, PeBouncer can assemble one instrumentation code for all instructions to instrument, and modify it for each stub to include the specific register which holds the vtable. Thus, each instrumentation stub for every vtable load instruction will operate on its specific vtable register.

While creating instrumentation code in assembly is already convenient, the complete API provided by an operating system can be used as well. This is important for the usage of Windows API functions (e.g., `OutputDebugString`). Therefore, a shared library (*service library*) is compiled with exported functions which wrap the API functions to use. Then an instrumentation stub is developed in a certain structure. It starts with a prolog to save the register context and instructions follow which save the instruction pointer (IP) to the stack. Thus, the first stack value will point to the absolute virtual address of the beginning of the stub. To call a service library's function from instrumentation code, an instruction is specified to load the saved IP into a register. A subsequent indirect call with the register and an annotation keyword containing the library and function name follows. When PeBouncer encounters such a keyword, it replaces the keyword with its four byte hash, such that an indirect call instruction with base register and displacement emerges.

After the instrumentation code is assembled, the four byte hash is replaced with a binary displacement value. This value is calculated in such a way that the register with the saved IP and the displacement (when summed up) point to a custom *Import Address Table* entry. This entry will contain the address to the service library's function to use. During runtime – similar to an Import Address Table (IAT) generation – an additional data section is filled with pointers to the service library's functions. This resolution is performed as soon as the instrumented executable is loaded into the address space of an application. Thus, the instrumentation code in the executable can reference the service library's resolved functions addresses in our custom generated IAT. The service library itself can be loaded in many ways into an applications address space [21]. Either the application's entry point is patched to load the library in case the instrumented executable is known to be loaded *afterwards*, or the service library is specified to be loaded into every process' address space [28] in case the instrumented executable is known to be loaded at the application's startup. In any case, the service library is loaded *before* the instrumented executable's entry is about to be executed. Note that the above concept allows the full support of both ASLR and NX.

4.2.3 Virtual Dispatch Instrumentation

To mitigate vtable hijacking, virtual dispatches are instrumented with policies $\mathscr{P}_{nw}$ (12 assembly instructions) or $\mathscr{P}_{nwa}$ (23 assembly instructions). Each virtual dispatch consists of the low-level semantic steps described in §2.2. To protect against the use of fake vtables, we instrument vtable load instructions after step one of the virtual dispatch semantic to be able to check the register with the vtable address. We do this in the following way: We keep a read-only 64KB sized lookup bitmap in our service library, representing the complete usermode memory pages. This bitmap is made writable and set up when an instrumented module is loaded into the address space of its application. Then its

access permissions are set to read-only again. Each bit represents the write permissions of a page. A set bit means the page is non-writable, an unset bit means the page is writable. Thus, when loading the module, we find all non-writable module sections in complete memory and set the appropriate bits of corresponding pages in the bitmap. To keep it up to date, Windows loader functions are hooked to change bits when unprotected modules are loaded and unloaded. By now, instrumentation checks can query the page of a vtable address by a simple lookup instead of querying the vtable itself: During runtime, a vtable is loaded into a register. The control flow is then rerouted to its instrumentation stub. Then, the vtable address is transformed with simple operations to an index into the page bitmap. The bit for the page is queried and if it is not set, a violation of $\mathscr{P}_{nw}$ occurred. A set bit means that the page, and thus the vtable it resides in, is non-writable. However, an adversary could circumvent this check, if she manages to find an address which is non-writable and contains a pointer to a gadget of choice to start her ROP chain. Thus, to mitigate this type of attacks, after the page lookup of the vtable, there is an additional virtual method check ($\mathscr{P}_{nwa}$): As step two of virtual dispatch semantics provides the offset to the virtual function, a pseudo-random index up to that offset is generated with the help of `rdtsc`. The vtable is dereferenced at this index and the resulting value is looked up in the page bitmap. A violation can be detected, as *all* entries in a valid vtable *above* the offset of the virtual function about to be called are method pointers pointing to non-writable code.

5. EVALUATION

We have implemented prototypes for both vExtractor and PeBouncer. In what follows, we evaluate both tools regarding their precision, performance overhead, and prevention of real-world exploits.

5.1 vExtractor's Precision

As a first step, we wanted to gain insights into the precision and recall of vExtractor's virtual dispatch detection. The analysis is performed against all identified indirect call instructions of a given program. We leverage a simple classification metric which states that any virtual dispatch found not being a virtual dispatch is a false positive (FP) and missed virtual dispatches are false negatives (FN). True positives (TP) and true negatives (TN) are the correctly found and rejected virtual dispatches, respectively. Based on this, we can calculate *precision*, *recall*, and *F-measure*.

We used version 4.9.0 of the *MinGW-w64* GCC cross compiler as it contains a partly implementation of GCC's *virtual table verification* feature [44]. Also, we ported missing parts of GCC's vtable verification library (`vtv`) to Windows to be able to compile 32-bit PE files with MinGW-w64 resulting in functional vtable verification checks. This porting was necessary since we instrument proprietary web browsers on Windows (see §5.2 and §5.3) and, thus, we needed a `vtv` version on Windows to compare against[1].

Compilation of 32-bit PE files with the `-fvtable-verify` flag will insert verification calls at each virtual dispatch after the instruction which loads the vtable into a register. Other resulting code stays identical for the same program when this flag is omitted. The version with verification is used to build a ground truth for indirect calls as they are preceded with verification routines in case the indirect call is part of a virtual dispatch. Otherwise, they are non-virtual calls. Indirect calls in the version compiled without verification are then grouped exactly into virtual dispatches and non-virtual dispatches based on the information gained from the first version. We applied vExtractor to the second version and classified the outcome of slices function-wise to retrieve a classification on the binary.

We utilized the open source C++ cryptographic library Botan [9], which contains 90 cryptographic primitives. We chose Botan because of its extended use of C++ features. We compiled it with and without vtable verification. vExtractor traced a total of 6779 indirect calls and identified 6484 virtual dispatches (TP) with 62 being non-virtual dispatches and 179 false negatives. This yields a precision of 0.99, a recall of 0.97, and an F-measure of 0.98.

We analyzed the reasons for false positives and discovered that they are due to C code constructs. More specifically, C code can have semantics equal to virtual dispatches. Consider the following C code line with `st` and `innerSt` being pointers to structs and `sFn` being a function pointer:

```
st->innerSt->sFn(st, p1, p2)
```

The two dereferences and `st` as first parameter fulfill the virtual dispatch semantics when compiled down to binary code. However, these and similar constructs can be eliminated in a profiling phase, as we show later in §5.2. We discuss the reasons for false negatives in §7.

5.2 Runtime of Instrumented Programs

To assess the performance overhead, we compiled Botan with and without GCC's vtable verification and compared the runtime in micro- and macro-benchmarks to the plain build of Botan, and to the plain build instrumented with T-VIP. Moreover, we compiled the SPEC CPU2006 benchmark, instrumented it with T-VIP and compared the runtimes to the native build. Finally, we hardened browser modules and measured their runtime overhead. Benchmarks were performed on an Intel Quad Core i7 at 2.6GHz with 2 GB of RAM running Windows 7.

Comparison to GCC's virtual table verification.

We patched our port of `vtv`'s source code to measure the CPU cycles needed for each verification routine execution (`VLTVerifyVtablePointer`) in order to perform micro-benchmarks. Therefore, we inserted GCC's build-in `rdtsc` routine at the beginning and at the end of the verification routine and executed Botan's benchmark. The verification produced a median cycle count of 9,205. We binary-rewrote Botan using T-VIP to measure the cycle count of our vtable protecting check code. Thus, we added additional `rdtsc` calls to the start and end of our instrumentation checks consisting of policies $\mathscr{P}_{nw}$ and $\mathscr{P}_{nwa}$, and took the plain build of Botan. We ran the benchmark and retrieved a median cycle count of 8,225 for $\mathscr{P}_{nw}$ and 12,335 for $\mathscr{P}_{nwa}$.

To perform macro-benchmarks, we built Botan with and without `vtv` with our newly ported GCC, not using `rdtsc`. We protected the plain build with T-VIP using policy $\mathscr{P}_{nwa}$ and run the benchmarking capability of both, ten times each. Botan's benchmark consists of 90 highly demanding cryptographic algorithms. The version compiled with GCC `vtv` produced a median overhead of 1.0 % with 46 algorithms

[1] Our MinGW-w64 extension is available at `https://github.com/RUB-SysSec/WindowsVTV`

producing a median overhead smaller than 2.0 %. The version protected with T-VIP produced a median overhead of 15.9 % with 37 algorithms producing a median overhead smaller than 2.0 %. We investigated the rather high appearing overhead: T-VIP installs a vectored exception handler for instrumented instructions, which cannot be overwritten with a jump to an instrumentation stub (see §4.2.1). As an exception handler produces high overhead, algorithms executing it will run perceptibly slower.

Runtime overhead measurements.

We compiled the seven C++ benchmarks of SPEC CPU 2006 with MS Visual C++ 2010, applied vExtractor and gained virtual dispatch slices for all except two (i.e., only five benchmarks actually have virtual dispatches). We hardened them with policies $\mathscr{P}_{nw}$, $\mathscr{P}_{nwa}$, and an empty policy ($\mathscr{P}_e$) separately, using PeBouncer. $\mathscr{P}_e$ consist of a prolog and epilog only, and serves to measure the net overhead introduced by PeBouncer. The results are depicted in Table 4.

Table 4: Binary sizes, amount of instrumented virtual dispatches (#VD), median runtime over three runs, and overheads of C++ SPEC CPU2006 benchmarks.

CPU2006	Size	#VD	Runtime (in s) and overhead (in %)						
			Native	$\mathscr{P}_e$		$\mathscr{P}_{nw}$		$\mathscr{P}_{nwa}$	
			rt(s)	rt(s)	ov(%)	rt(s)	ov(%)	rt(s)	ov(%)
soplex	403K	746	232.25	231.05	-0.52	232.41	0.07	233.64	0.60
omnetpp	793K	1593	217.12	293.72	35.28	303.48	39.78	318.15	46.53
povray	1038K	154	164.27	164.22	-0.03	164.36	0.06	164.31	0.03
dealII	947K	272	360.97	361.75	0.22	363.01	0.57	363.14	0.60
xalancbmk	3673K	14061	182.97	294.29	60.84	331.98	81.44	372.26	103.45

Overheads are $\leq$ 0.6 % in `soplex`, `povray` and `dealII`, while high overheads for $\mathscr{P}_{nw}$ and $\mathscr{P}_{nwa}$ in `omnetpp` and `xalancbmk` are mostly due to the net overhead of our rewriting engine ($\mathscr{P}_e$ column in Table 4). Using our policies $\mathscr{P}_{nw}$ and $\mathscr{P}_{nwa}$ with another binary rewriter could lower the overhead. However, as we show with COTS browser modules, the overhead in macro-bechmarks is actually low in practice.

We applied vExtractor to `xul.dll` of Mozilla Firefox 17.0.6 and to `mshtml.dll` of Internet Explorer in versions 8, 9, and 10. We chose these modules because they contain the major parts of the browsers' engines and former zero-day attacks were related to code in these modules (see §5.3 for details). Table 5 shows the amount of indirect calls and extracted virtual dispatch slices.

We applied PeBouncer to each module to instrument all vtable load instructions, such that during runtime, the addresses of vtables, their memory page permission, and the addresses of the corresponding virtual call sites are gained with `OutputDebugString`. Less than 900 exception handlers had to be inserted for each module due to non-overwritable instructions, but *all* were instrumented without problems.

We then, at first, ran the two browser benchmarks *SunSpider* [3] and *Kraken* [33] to profile the browsers. Vtable addresses retrieved, not being vtables, show themselves as writable. This way, we can filter out all non-virtual dispatches like calls from nested C structs and eliminate all false positives (see Table 5 for details). These were removed from the virtual dispatch slices and each module was rewritten again by PeBouncer. This time, we used instrumentation checks based on our polices $\mathscr{P}_{nw}$ and $\mathscr{P}_{nwa}$, and policy $\mathscr{P}_e$ was used as well. All benchmarks were run again to measure the introduced performance overhead. The results can be seen in Figure 4 and yield an overall average performance overhead of approx. 2.1 % ($\mathscr{P}_e$), 1.6 % ($\mathscr{P}_{nw}$) and 2.2 % ($\mathscr{P}_{nwa}$).

Table 5: Number of indirect calls (#IC), extracted virtual dispatch slices, and filtered non-virtual calls. #Instr. indicates the number of slices fed into PeBouncer to harden listed modules.

App.	Module	#IC	#Slices	#Filtered	#Instr.
Fx 17.0.6	xul.dll	66,120	53,268	73	53,195
IE 8	mshtml.dll	23,682	19,721	3,117	16,604
IE 9	mshtml.dll	64,721	53,312	7,735	45,577
IE 10	mshtml.dll	56,149	44,383	5,515	38,868

5.3 Vtable Hijacking Detection

Real-world exploits for zero-day vulnerabilities utilized vtable hijacking to first load a fake vtable, and then dereference an entry to call a ROP gadget. In this way, attackers gained a foothold into victim systems via CVE-2013-3897, CVE-2013-3893, and CVE-2013-1690. The virtual dispatches were all found by vExtractor and successfully protected with policies $\mathscr{P}_{nw}$ and $\mathscr{P}_{nwa}$ by PeBouncer. We then attempted to exploit the protected web browsers with corresponding exploits from Metasploit and exploits gained from the wild. All attempts were detected successfully already with $\mathscr{P}_{nw}$.

Another critical vulnerability (CVE-2013-2556) in Windows 7 allowed remote code execution without any shellcode or ROP in conjuction with vtable hijacking. The culprit was the non-ASLR protected *SharedUserData* memory region containing function pointers [46]. Attackers used the region's address as fake vtable and an entry with a pointer to `LdrHotPatchRoutine` to gain remote code execution via virtual dispatches. This is detected by policy $\mathscr{P}_{nw}$, as it checks vtables for non-writable in modules. Another zero-day use-after-free vulnerability (CVE-2014-0322) was used in targeted attacks. While the vulnerability only allowed a one byte write, a vtable pointer of a *flash* object was modified to gain control [16]. As the precision of vExtractor is high, T-VIP can protect against this vulnerability when the corresponding virtual dispatch is extracted and then instrumented by PeBouncer.

6. RELATED WORK

Due to their prevalence and high practical impact, software vulnerabilities have received a lot of attention in the last decades. In the following, we briefly review work that is closely related to our approach and discuss how our approach differs from previous work in this area.

There are many methods that can be used to harden a given system against software vulnerabilities. A few examples include *data execution prevention* (DEP) [31], *address space layout randomization* (ASLR), SAFESEH/SEHOP to protect execption handlers, *instruction set randomization* (ISR) [5], and similar approaches [18, 25, 45]. They are all complementary to our approach, which primarily focuses on protecting the integrity of vtables.

A general concept to prevent memory corruption attacks that divert the control flow of a given program is *Control Flow Integrity* (CFI) [1]. The basic idea is to instrument a given program to verify that each control flow transfer jumps to a valid program location. Recently, several methods were proposed to implement CFI with low performance overhead [47, 48]. Efficient implementation incur a performance overhead of less than 5 %, but had to sacrifice some

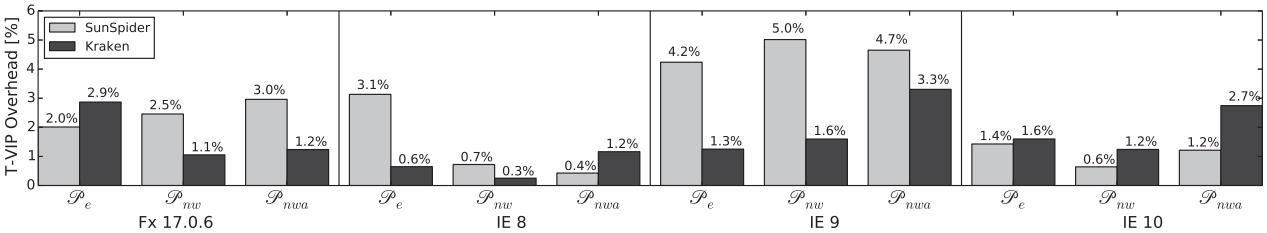


Figure 4: Runtime overhead for instrumented browsers on the browser benchmarks *SunSpider* and *Kraken*.

of the security guarantees given in the original CFI proposal [1]. Götkas et al. recently demonstrated circumventions of these CFI implementations [19]. Their proof-of-concept attack gains control over an indirect transfer by overwriting a vtable pointer with a heap address. This specific use case is detectable by our approach: We enforce policies at instructions which load vtable addresses *before* targets of indirect transfers are loaded. Bogus targets might seem legitimate in coarse-grained CFI protections, due to conforming to their CFI policies. We detect a violation if *any* indirect target comes from a fake vtable.

The main difference compared to existing work is that we specifically focus on the integrity of virtual dispatches, since vtable hijacking attacks have become one of the most widely used attack vectors recently. Instead of protecting all indirect jumps and inducing a performance impact that prevents widespread adoption [43], we focus on a specific subset of indirect jumps that are an attractive target for attackers.

Recently, several compiler extensions were proposed that protect vtables from hijacking attacks:

- GCC introduced the `-fvtable-verify` option [44] that analyzes the class hierarchy during the compilation phase to detect all vtables. Furthermore, checks are inserted before virtual call sites to verify the integrity of virtual dispatches.
- Closely related, SafeDispatch implements an LLVM extension that performs the same basic steps [22]. A class hierarchy analysis determines all valid method implementations and additional checks are inserted to ensure that only valid dispatches are performed during runtime. The measured runtime overhead is about 2.1 %.
- VTGuard by Microsoft [24] adds a guard entry at the end of the vtable such that (certain kinds of) vtable hijacks can be detected. This instrumentation is added during the compilation phase. If an information leak exists, an attacker could use this to obtain information about the guard entry, enabling a bypass of the approach.

The main difference to our approach is the fact that we operate on the binary level such that we can also protect proprietary programs where no source code is available. Since the full class hierarchy can be determined during the compilation phase, the security guarantee provided by the first two approaches is stronger than ours: These approaches can implement $\mathscr{P}_{obj}$ and perform a full integrity check. However, empirical results demonstrate that our policy can already defeat in-the-wild zero-day exploits. Our performance overhead is sligthly higher, but this is mainly due to the fact that we instrument binary programs.

7. DISCUSSION

In the following, we discuss the limitations and shortcomings of our approach and the current implementation.

It is crucial to identify virtual dispatches precisely in order to protect all virtual call sites. As the evaluation shows, vExtractor misses 2.6 % of virtual dispatches. Recall formulae (1) and (2) from §2.2. Manual investigation revealed that in rare cases, especially GCC creates multiple *aliases* for *obj*. While vExtractor already performs an alias analysis to some extent, cases can slip away if an alias of *obj* is used in instructions represented by (2), but cannot be connected to *obj* in (1). Also, at the time of writing, trying to compile Firefox with the original GCC 4.9.0 enabling GCC's vtable-verification, led to compiler crashes. Thus, we were not able to evaluate vExtractor's precision using Firefox as ground truth.

Currently, binaries have to be profiled in order to remove virtual dispatch-like code constructs. On the binary level, it is impossible to differentiate certain C code constructs from virtual dispatches, and thus we need this (automated) profiling phase, to filter all non-virtual dispatches.

As shown in our evaluation, T-VIP protects against real-world vtable hijacking attacks. However, policy $\mathscr{P}_{nw}$ could be circumvented by using a pointer residing in a non-writable module memory page and pointing to code of an attacker's choice. To mitigate this, we sacrify performance by generating a random index into a vtable in the implementation of $\mathscr{P}_{nwa}$. Hence, T-VIP guarantees that a *different* vtable entry is checked for *each* execution time at the *same* virtual dispatch. An attacker is thereby restricted to use non-writable function tables in order to reliably compromise a system. By itself, circumventing this is already very hard, but would be still possible if a valid vtable of a wrong class type is used at a virtual dispatch site. This is a limitation we have in common with VTGuard according to [22]. However, implementing $\mathscr{P}_{obj}$ would prevent even such attacks.

PeBouncer currently supports 32-bit PE files since the majority of web browsers uses 32-bit code and this is the primary target of use-after-free exploits. However, the concept of PeBouncer is usable for 64-bit code and the ELF file format as well, with only minor modifications. Some corner cases during rewriting are currently handled by an exception handler and introduce additional overhead. This could be solved by leveraging binary rewriting capabilities of ROSE [35] to insert checks inline.

8. CONCLUSION

In this paper, we introduced an approach to protect binary programs against vtable hijacking vulnerabilities, which have become the de-facto attack vector on modern browsers. To this end, we introduced an automated method to extract

virtual function dispatches from a given binary, which we implemented in a tool called vExtractor. Furthermore, we developed a generic, static binary rewriting engine for PE files called PeBouncer that can instrument a given binary with a policy that checks the integrity of virtual function dispatches. Empirical evaluations demonstrate that our approach can detect recent zero-day vulnerabilities and the performance overhead is only slightly higher compared to compiler-based approaches.

Acknowledgements

We thank Patrick Wollgast for porting `vtv` to MinGW-w64, Behrad Garmany and Carsten Willems for fruitful discussions, and the anonymous reviewers for their feedback. This work was supported by the German Federal Ministry of Education and Research (BMBF grant 01BY-1205A – JSAgents)

9. REFERENCES

[1] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-Flow Integrity. In *ACM Conference on Computer and Communications Security (CCS)*, 2005.

[2] J. Afek and A. Sharabani. Dangling pointer: Smashing the pointer for fun and profit. *Black Hat USA*, 2007.

[3] Apple. Sunspider 1.0.2. `https://www.webkit.org/perf/sunspider/sunspider.html`, 2014.

[4] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: a transparent dynamic optimization system. *ACM SIGPLAN Notices*, 35(5):1–12, 2000.

[5] E. G. Barrantes, D. H. Ackley, S. Forrest, and D. Stefanović. Randomized instruction set emulation. *ACM Trans. Inf. Syst. Secur.*, 8(1):3–40, Feb. 2005.

[6] P. Becker et al. Working draft, standard for programming language c++. Technical report, Technical Report, 2011.

[7] D. Binkley and M. Harman. A Survey of Empirical Results on Program Slicing. *Advances in Computing*, 62:105–178, 2003.

[8] T. Bletsch, X. Jiang, V. W. Freeh, and Z. Liang. Jump-oriented programming: a new class of code-reuse attack. In *ACM Symposium on Information, Computer and Communications Security*, 2011.

[9] Botan. Botan C++ crypto library. `http://botan.randombit.net/`, 2013.

[10] D. Bruening, E. Duesterwald, and S. Amarasinghe. Design and implementation of a dynamic optimization framework for windows. In *4th ACM Workshop on Feedback-Directed and Dynamic Optimization (FDDO-4)*, 2001.

[11] J. Caballero, G. Grieco, M. Marron, and A. Nappa. Undangle: early detection of dangling pointers in use-after-free and double-free vulnerabilities. In *International Symposium on Software Testing and Analysis (ISSTA)*, 2012.

[12] D. Dai Zovi. Practical return-oriented programming. *SOURCE Boston*, 2010.

[13] M. Daniel, J. Honoroff, and C. Miller. Engineering Heap Overflow Exploits with JavaScript. In *USENIX Workshop on Offensive Technologies (WOOT)*, 2008.

[14] K. Driesen and U. Hölzle. The direct cost of virtual function calls in c++. *ACM Sigplan Notices*, 31(10):306–323, 1996.

[15] T. Dullien and S. Porst. REIL: A platform-independent intermediate representation of disassembled code for static code analysis. *CanSecWest*, 2009.

[16] FireEye. Operation SnowMan. `http://goo.gl/NLOZmV`, 2014.

[17] R. Gawlik and T. Holz. Towards Automated Integrity Protection of C++ Virtual Function Tables in Binary Programs. Technical Report TR-HGI-2014-004, Ruhr-University Bochum, Oct. 2014.

[18] C. Giuffrida, A. Kuijsten, and A. S. Tanenbaum. Enhanced operating system security through efficient and fine-grained address space randomization. In *USENIX Security Symposium*, 2012.

[19] E. Göktas, E. Athanasopoulos, H. Bos, and G. Portokalidis. Out of control: Overcoming control-flow integrity. In *IEEE Symposium on Security and Privacy*, 2014.

[20] M. Harman and S. Danicic. Amorphous program slicing. In *5th International Workshop on Program Comprehension*, 1997.

[21] I. Ivanov. Api hooking revealed. *The Code Project*, 2002.

[22] D. Jang, Z. Tatlock, and S. Lerner. SafeDispatch: Securing C++ Virtual Calls from Memory Corruption Attacks. In *Symposium on Network and Distributed System Security (NDSS)*, 2014.

[23] W. Jin, C. Cohen, J. Gennari, C. Hines, S. Chaki, A. Gurfinkel, J. Havrilla, and P. Narasimhan. Recovering c++ objects from binaries using inter-procedural data-flow analysis. In *Proceedings of ACM SIGPLAN on Program Protection and Reverse Engineering Workshop 2014*, page 1. ACM, 2014.

[24] K. D. Johnson and M. R. Miller. Using virtual table protections to prevent the exploitation of object corruption vulnerabilities, 2010. US Patent App. 12/958,668.

[25] C. Kil, J. Jim, C. Bookholt, J. Xu, and P. Ning. Address space layout permutation (aslp): Towards fine-grained randomization of commodity software. In *Annual Computer Security Applications Conference (ACSAC)*, 2006.

[26] S. Krahmer. x86-64 buffer overflow exploits and the borrowed code chunks exploitation technique. `http://users.suse.com/~krahmer/no-nx.pdf`, 2005.

[27] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. *Acm Sigplan Notices*, 40(6):190–200, 2005.

[28] Microsoft. AppInit_DLLs in Windows 7. `http://goo.gl/BchJ4J`, 2013.

[29] Microsoft. MS13-080 addresses two vulnerabilities under limited, targeted attacks. `http://goo.gl/sCZNkL`, 2013.

[30] Microsoft. Thiscall Calling Convention. `http://goo.gl/5o48Ub`, 2013.

[31] I. Molnar. Exec shield, new linux security feature. *News-Forge, May*, 2003.

[32] Mozilla. Firefox 0-day found on Tor .onion service. `https://bugzilla.mozilla.org/show_bug.cgi?id=901365`, 2013.

[33] Mozilla. Kraken Benchmark Suite. `http://krakenbenchmark.mozilla.org/`, 2014.

[34] P. O'Sullivan, K. Anand, A. Kotha, M. Smithson, R. Barua, and A. D. Keromytis. Retrofitting security in cots software with binary rewriting. In *Future Challenges in Security and Privacy for Academia and Industry*, pages 154–172. Springer, 2011.

[35] D. Quinlan. ROSE: Compiler Support for Object-oriented Frameworks. *Parallel Processing Letters*, 10(02/03), 2000.

[36] G. F. Roglia, L. Martignoni, R. Paleari, and D. Bruschi. Surgically returning to randomized lib (c). In *Annual Computer Security Applications Conference (ACSAC)*, 2009.

[37] F. J. Serna. The info leak era on software exploitation. *Black Hat USA*, 2012.

[38] H. Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *ACM Conference on Computer and Communications Security (CCS)*, 2007.

[39] J. Silva. A vocabulary of program slicing-based techniques. *ACM Computing Surveys (CSUR)*, 44(3):12, 2012.

[40] A. Sotirov. Heap Feng Shui in JavaScript. *Black Hat Europe*, 2007.

[41] A. Srivastava, A. Edwards, and H. Vo. Vulcan: Binary transformation in a distributed environment. Technical report, technical report msr-tr-2001-50, microsoft research, 2001.

[42] B. Stroustrup. *C++*. John Wiley and Sons Ltd., 2003.

[43] L. Szekeres, M. Payer, T. Wei, and D. Song. Sok: Eternal war in memory. In *IEEE Symposium on Security and Privacy*, 2013.

[44] C. Tice, T. Roeder, P. Collingbourne, S. Checkoway, Ú. Erlingsson, L. Lozano, and G. Pike. Enforcing forward-edge control-flow integrity in gcc & llvm. In *USENIX Security Symposium*, 2014.

[45] R. Wartell, V. Mohan, K. W. Hamlen, and Z. Lin. Binary stirring: Self-randomizing instruction addresses of legacy x86 binary code. In *ACM Conference on Computer and Communications Security (CCS)*, 2012.

[46] Y. Yang. DEP/ASLR bypass without ROP/JIT. *CanSecWest*, 2013.

[47] C. Zhang, T. Wei, Z. Chen, L. Duan, L. Szekeres, S. McCamant, D. Song, and W. Zou. Practical Control Flow Integrity & Randomization for Binary Executables. In *IEEE Symposium on Security and Privacy*, 2013.

[48] M. Zhang and R. Sekar. BinCFI: Control Flow Integrity for COTS Binaries. In *USENIX Security Symposium*, 2013.

Leveraging Semantic Signatures for Bug Search in Binary Programs

Jannik Pewny[†], Felix Schuster[†], Christian Rossow[‡], Lukas Bernhard[†], Thorsten Holz[†]
[†]Horst Görtz Institute for IT-Security (HGI) Ruhr-University Bochum, Germany
[‡]Cluster of Excellence, MMCI Saarland University, Germany

ABSTRACT

Software vulnerabilities still constitute a high security risk and there is an ongoing race to patch known bugs. However, especially in closed-source software, there is no straightforward way (in contrast to source code analysis) to find buggy code parts, even if the bug was publicly disclosed.

To tackle this problem, we propose a method called Tree Edit Distance based Equational Matching (TEDEM) to automatically identify binary code regions that are "similar" to code regions containing a reference bug. We aim to find bugs both in the same binary as the reference bug and in completely unrelated binaries (even compiled for different operating systems). Our method even works on proprietary software systems, which lack source code and symbols.

The analysis task is split into two phases. In a preprocessing phase, we condense the semantics of a given binary executable by symbolic simplification to make our approach robust against syntactic changes across different binaries. Second, we use *tree edit distances* as a basic block-centric metric for code similarity. This allows us to find instances of the same bug in different binaries and even spotting its variants (a concept called *vulnerability extrapolation*). To demonstrate the practical feasibility of the proposed method, we implemented a prototype of TEDEM that can find real-world security bugs across binaries and even across OS boundaries, such as in `MS Word` and the popular messengers `Pidgin` (Linux) and `Adium` (Mac OS).

1. INTRODUCTION

Security vulnerabilities in software systems have plagued us since many years. There are numerous classes of security-relevant bugs such as buffer overflows, use-after-free conditions, and logical flaws. While it is possible to define a bug class on an abstract level (e. g., "a copy operation writes data beyond array bounds"), the concrete manifestation of a vulnerability in a program can be arbitrarily complex. As such, it is hard to apply the abstract definitions in practice.

Many approaches were proposed that perform an automated security analysis of a given piece of (binary or source) code to detect specific security vulnerabilities. Such methods either leverage characteristic patterns for identifying security vulnerabilities [10] or utilize program analysis techniques such as fuzzing [24], taint analysis [22, 25], and symbolic execution [3, 4] to find bugs. In general, the problem of identifying all bugs in a given program is undecidable and thus all approaches have certain shortcomings. For example, they can only detect predefined kinds of vulnerabilities or miss some manifestations of a bug (i.e., they have false negatives) during the analysis phase.

Finding similar bugs. A variant of the general problem of finding bugs is to identify bug doublets, given a concrete example of a buggy piece of code. The problem can be stated as follows: *Given an exemplary bug, find locations in a given program which have this or a similar bug.*

While finding completely new bugs is certainly the more ambitious goal, there is a great benefit in finding *similar* bugs. For example, it helps to find dormant bugs, identify false negatives, or to check if a binary supplied by a third-party still has a known bug or is patched already. Yamaguchi et al. [28] studied this problem on the source code level and they coined the term *vulnerability extrapolation*. The underlying assumption is the following: When code is similar to code which contains a bug, it is likely to contain the same bug as well [21]. Based on this assumption, the problem of vulnerability extrapolation can be addressed with a reasonable code similarity metric.

In the real world, there are many reasons for bug doublets. The underlying root cause is typically *code reuse*, which may happen due to copy'n'paste or project forks [17]. When maintaining a code base with different versions of the same code, it is almost certain that a bug found in one version of a program is not fixed in all versions. This problem grows when the code is not maintained by the same group of people or when code is shared among different projects (e. g., an open-source library). Copied code may be modified to suit the slightly different needs and bugs tend to happen in similar, but not equal, circumstances. Thus, looking for *exact* matches may severely limit usefulness.

When looking for bugs, there are many reasons to analyze binary code instead of source code. Most importantly, source code is not always available (e. g., proprietary code used in commercial products, third-party libraries, or legacy code). In addition, the compilation phase itself may introduce bugs that simply were not present in the source code [27]. Unfortunately, finding bugs gets a lot harder when the analysis is performed on binary code. Several techniques can detect code reuse in source code [12, 14, 18], but they cannot directly be applied to binary code: Transformations and optimizations performed by a compiler such as register assignments, function inlining, constant folding/propagation, or instruction reordering alter code structure and code syntax. Hence, identifying similar code regions is challenging.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ACSAC December 8-12, 2014, New Orleans, LA, USA
Copyright is held by the owner/author(s).
Publication rights licensed to ACM.
ACM 978-1-4503-3005-3/14/12 ...$15.00
http://dx.doi.org/10.1145/2664243.2664269.

Our approach. In this paper, we introduce a method called Tree Edit Distance based Equational Matching (TEDEM). Given a so called *signature*, TEDEM finds code locations in a binary that has similar bugs. TEDEM finds vulnerabilities in binary programs by identifying buggy code based on semantic signatures for a given security bug. In contrast to source code-based techniques that heavily rely on symbols (neglecting semantic information), we take another approach to address the problem. In a preprocessing phase, we first disassemble a binary and extract semantic information of the basic blocks in forms of *expression trees* (i. e., equations that summarize the results of the computations performed in the basic block). This symbolic representation makes our analysis robust against small syntactic changes across binaries.

Based on this information, we can search for similar code regions. To this end, we introduce a metric for fine-grained, basic block-centric comparison of binary code, which allows us to find bugs through code similarity. The metric takes most of the semantic information into account. In contrast, methods focusing solely on mnemonics [20], CFG structure [15], or API calls [1] are by design ignoring certain important semantic information. Thus, they often cannot reveal sufficient details to compare small code segments, as it is especially required for extrapolating vulnerabilities. If the granularity is too coarse, one can only look for aggregated, statistical effects over bigger structures, like functions or libraries — losing the flexibility to search for similar bugs contained in other substructures.

We evaluated TEDEM on multiple large and real-world binaries. For example, we found unpatched vulnerabilities in `PuTTY` forks across binary boundaries and in `Adium`'s libpurple even across OS boundaries. Furthermore, we found patch level discrepancies in `MS Office` products. Despite its fine-granular bug search, TEDEM scales reasonably well and could find bugs even in large programs such as `MS Word 2013` (6.3 MB and 416,736 basic blocks) in less than 18 min.

Contributions. In this paper, we make the following three main contributions:

- We designed and implemented a basic block-centric metric based on tree edit distances to measure the similarity of two pieces of binary code that is capable of tolerating small syntactic changes in the program.
- Using this as a building block, we implemented a scheme to measure similarity of larger code constructs, like sub-CFGs and functions.
- Empirical measurements demonstrate the viability of the approach and we found real-world bugs across binary and even across OS boundaries.

2. RELATED WORK

Finding bugs in software systems has attracted a lot of research. In the following, we briefly review prior work in this area and discuss how it relates to our approach.

2.1 Code Clones in Source Code

ReDeBug [11] is a source code-based system to identify unpatched code clones in a very fast way, which is due to the fact that they only search for close-to-exact matches. While it shares the goal with our work, it requires source code and is thus of no help for binary software.

Yamaguchi et al. proposed *extrapolation of vulnerabilities* on source code level [28]. Given source code written in C, they extract *abstract syntax trees* (AST) of all contained functions. Based on the properties of their ASTs, functions are projected into a high-dimensional vector space. Subsequently, methods from the realm of machine learning are applied in order to cluster functions, which reveals functions similar to ones known to be vulnerable. We achieve comparable results as reported by Yamaguchi et al. (cf. Section 5), just that our approach does not require source code.

2.2 Binary Code Comparison

BinDiff, as proposed by Dullien and Rolles [5], is the de facto standard commercial tool for comparing two pieces of binary code. *BinDiff* mainly relies on the structural similarity and makes only little use of code semantics. Thus, one of its weaknesses is that it conceptually struggles with matching functions that are only invoked indirectly and are thus not connected in a call graph (regularly the case for, e. g., C callbacks, C++ virtual methods, Objective-C methods). Furthermore, BinDiff regularly mismatches basic blocks on the foundations of similar structure, but different semantics.

Another group of related work explicitly assesses *semantic* similarities/differences of binary software. Many works are based on summarizing a basic block's effects on the program state as a set of equations (sometimes referred to as *symbolic execution* [25]) and we also utilize such an approach. *BinHunt* [7] tries to find semantic differences between a binary program and its patched version to pinpoint vulnerabilities. To determine how similar two basic blocks are, they test all possible pairs of equations from both sets for equality using a theorem prover. Based on identified semantically equal basic blocks, a custom backtracking algorithm finds the largest maximum common induced subgraph between two functions and derives the similarity between two functions from this. Ming et al. proposed *iBinHunt* [19], which adds taint analysis to *BinHunt* to allow for less basic block comparisons. As these tools are targeted for whole-binary, function-level "diffing", they are somewhat ill-suited for our use-case of finding small and fine-grained bug signatures: their idea would fail with function inlining and could not find out-of-context re-use of vulnerable code snippets. In addition, these two approaches can only verify if code is "equal" and have to rely on handcrafted heuristics to use their non-continuous similarity score as a metric. As we will show, a proper metric is vital in the context of bug searches.

BinHash is a sampling-based approach to identify semantically similar binary functions [13]. *BinHash* evaluates the sets of basic block equations concretely for randomly-chosen input vectors and all their permutations. Functions are then compared via locality-sensitive hashing over the output vectors of their basic blocks. Since there are generally $n!$ permutations for a given vector, scalability problems arise for basic blocks with large input dimensions. While evaluating TEDEM, we encountered basic blocks with an input dimension even larger than 50, which demonstrates scalability issues with approaches such as *BinHash*.

Lakhotia et al. presented *BinJuice* [16]. They normalize a basic block's equation set in several steps to extract its semantic "juice". Semantically similar basic blocks are then identified through lexical comparisons of juices. Since the authors did not give a quantifiable evaluation, it remains open if their metric is suitable. In addition, the authors did not expand their similarity computation beyond the scope of individual basic blocks, which is required for bug search.

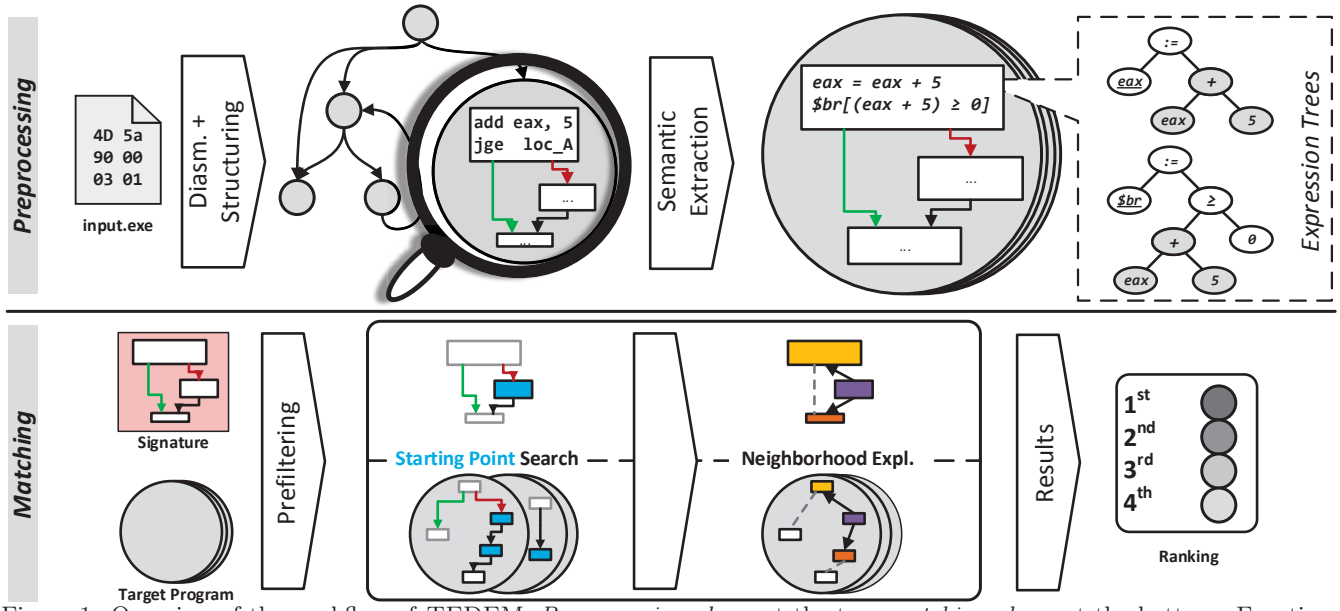


Figure 1: Overview of the workflow of TEDEM; *Preprocessing phase* at the top, *matching phase* at the bottom. Functions are gray circles, basic blocks are solid white rectangles.

Exposé [23] identifies library re-use in binary software. First, promising function matches between the library and a binary program are gathered by examining simple features like *number of arguments* or *cyclomatic complexity*. Second, candidate function matches are verified by symbolically executing both functions and leveraging a theorem prover. As *Exposé* aims to identify library matches, it is too coarse-grained for certain scenarios when extrapolating bugs.

2.3 Finding Unknown Bugs

Tools such as AEG [2] and Mayhem [3] aim at automatically detecting security vulnerabilities. They use binary analysis techniques such as symbolic/concolic execution to reason about vulnerable code paths. Our goal is different in the sense that we search for similar security vulnerabilities given an exemplary bug signature. In addition, many such approaches aim to find specific bug classes, such as control flow hijack vulnerabilities. In contrast, our system can be used to find any type of bug.

3. SYSTEM OVERVIEW

In the typical use case of our approach, an analyst knows about a concrete bug (maybe from a CVE advisory or from a patch) and aims to find similar bugs. In our terminology, the analyst extracts a *bug signature* and uses it to find further similar bugs in a *target program*.

Figure 1 visualizes the two phases of our Tree Edit Distance based Equational Matching (TEDEM). First, in the upper half, we preprocess program binaries for our bug-search. This phase extracts disassembly and structure (i. e., a control flow graph) from the programs and then summarizes semantics of the basic blocks in form of expression trees. Preprocessing only has to happen once per signature and — independently — once per target program. The preprocessed output can be reused in future bug searches.

Second, as shown in the lower half of Figure 1, our system searches for a signature within a target program and identifies binary code parts which are similar to the signature. For this, we identify single high-quality basic blocks matches in the signature and the target program. We use these pairs of basic blocks as starting points to compare the full signature with parts of the target program, such as functions or parts thereof. At the end of the workflow, TEDEM reports the similarity of each function from the target programs to the signature in a sorted list, revealing the functions with substructures similar to the signature.

3.1 Program Preprocessing

In practice, compilers complicate the process of binary comparison, even if the binaries have the same source code base. Two different compilers or compiler configurations typically create different versions of binary code from equal source code. Most compilers apply memory or performance optimizations so that the representations between compiled binaries of two configurations differ significantly. For example, the usage of registers can change, different calling conventions may be used, the control flow graph may be altered, or arithmetic operations could be tweaked to optimize performance. This can occur if another compiler (or version) is used, the optimization levels are changed, or even if slightly modified source code allows the compiler to perform better optimizations. All these modifications complicate the process of comparing two binaries at the disassembly level.

In this section, we describe the preprocessing steps that we take to compensate for these issues.

3.1.1 *Disassembly and Structuring*

First, we obtain a disassembly of the programs, i. e., the original (buggy) program and the target program. We then split each program into *strict basic blocks*. That is, we extract the basic blocks and further split these at function calls. Each strict basic block is thus only left at its final instruction, the terminator, and only its first instruction is a jump target. While this definition is used for basic blocks as well, the instruction after a call is an implicit target of a return instruction and is naturally not the first instruction

of a basic block. For the rest of this paper, whenever we mention a basic block, we refer to a *strict* basic block.

3.1.2 Semantic Extraction

Next, we transform the sequence of a basic block's instructions into equations. Each equation is an assignment where left-hand side and right-hand side both contain arbitrary computations. The left side of the equation is (or results in) a register or memory location. The equations are built such that each register, write-address, or jump condition only appears once on the left side of an assignment (like a single static assignment). We compute the equations by accumulating the computational steps performed on one such variable on the right side of the assignment. All inputs on the right side of the equations are not modified by preceding equations. For example, we consider the *input* variable `eax` as constant, even if a value is assigned to the *output* variable `eax` before the variable is used in a subsequent equation. Therefore, for a given input state, the equations can be evaluated independently from one another.

This preprocessing abstracts from instruction ordering and allows a natural sorting of the assignments. We chose an arbitrary but deterministic order to sort the equations, such that two equal basic blocks have the same order of equations.

Further, we add equations for the number of successors/predecessors. We also add a condensed terminator-equation, where we encode possible jump destinations and, if present, a branching condition. In addition, we distinguish various kinds of constants by their type of usage, e. g., code locations, data offsets, or arithmetic/logic constants.

We then simplify and normalize the assignments on a basic block level. For details, we refer to Section 4.1.2.

3.2 Bug Signature Matching

Once the binaries have been preprocessed, we continue with the original goal to find bugs in these programs. In this section, we define the notion of a *signature* (Section 3.2.1), which is searched for in the *target program*. We start by describing how we compare a basic block of the signature with a basic block of the target program (Section 3.2.2). We then outline our approach to prefilter basic blocks to gain search speedups, without decreasing effectiveness (Section 3.2.3). Lastly, we leverage the CFG to apply the signature (Section 3.2.4) on the target program.

3.2.1 Bug Signature

A *bug signature* describes a program part that is characteristic for a certain bug. We define a bug signature as *a subset of the CFG of a vulnerable function*. This means that an analyst only needs to identify the bug-relevant parts of a function. Essentially, the bug signature is binary code, i.e., contains basic blocks and their transitions and thus undergoes the same handling as the target programs. In the simple case of function-level code duplicates, the signature can be equal to the CFG of the vulnerable function. In more advanced bugs, more fine-grained signatures can be as small as parts of the equations of certain basic blocks. We will demonstrate in the evaluation section that – once a bug is known and has been understood – deriving a signature is straightforward with minimal manual effort.

However, too generic signatures risk to cause false positives. Therefore, instead of searching for entire bug classes, the signature focuses on finding specific bugs. For example, instead of only searching for writes to memory with lacking bound checks (to capture buffer overflows), we also search for contextual information that is specific to the buggy reference program (e.g., basic blocks surrounding the bug, concrete buffer lengths, and the specific way the memory is modified). Our search is thus *contextual*, as the signature is derived from a concrete bug context in a reference program. Despite the need for context, our approach can also find similar bugs in programs other than the reference program.

3.2.2 Basic Block Comparison

Given the signature, we want to find similar bugs in the target program. A cornerstone in this process is comparing basic blocks of the signature with basic blocks of the target program. We leverage the tree structure of the basic block equations for this comparison. After the preprocessing phase (Section 3.1), each basic block is represented as a list of equations. An equation is a tree, where the root node is an assignment or a basic block terminator, the leaf nodes are registers or constants (e. g., numbers, addresses), and intermediate nodes are operations. We thus leverage a tree comparison algorithm to measure similarity between two given basic blocks.

We chose to use a tree edit distance [26] (TED) to compare equations. The TED measures the minimum costs for transforming one tree (e. g., basic blocks from the signature) to another (e. g., basic blocks from the target program). That is, it measures the costs for modifying trees via node replacements and insertions/deletions of subtrees. We get low distances for similar, and higher distances for less similar inputs. The TED thus measures the syntactic difference in the condensed semantic representation of the basic block. The costs correlate with the number of nodes that need to be replaced, inserted or deleted, and are *at most* as large as the number of modified nodes. Tekli et al. provide a detailed description of the distance computation [26].

3.2.3 Prefiltering & Candidate Search

Naïvely, we could compare all basic blocks from the signature with all those from the target program and expand our search from there. However, especially if fine-grained similarity measures (as the TED) are used, this results in scalability problems. We thus leverage the fact that some basic blocks in the signature are more characteristic than others. For example, basic blocks that are too generic have likely similar counterparts spread among multiple parts of the program, and thus are not really useful to characterize a bug. Similarly, compared to a reference basic block in the signature, most of the basic blocks in the target program differ so much that fine-grained comparisons (e. g., using the TED) are not needed to tell that basic blocks are not similar. Guided by these observations, we (i) focus on the parts in the signature that are distinctive enough to serve as a marker for a bug, and (ii) reduce the number of basic block comparisons in the target program.

We found that a few *coarse-grained* basic block attributes are sufficient to indicate very low similarity of two basic blocks. That is, the coarse-grained similarity measure correlates with the fine-grained similarity computed via the TED. Coarse-grained basic block attributes are, for example, the number of equations, the depth of the equation trees, or the number of nodes in the tree. We thus evaluated if we can use these attribute-level comparisons in a potential prefiltering step. In particular, we measured how well these attributes reflect our notion of basic block similarity. We evaluated this

in three binary programs (`Adium` on Mac OS, `ImageMagick` on Linux, `MS Word 2013` on Windows) by computing the correlation between the similarity of single coarse-grained attributes (e. g., number of equations in basic block A and B) and the fine-grained TED similarity score between the basic blocks. We found strong correlation coefficients (averaged over the binaries 0.63, 0.92, 0.52, respectively) for all three basic block attributes and thus use them as prefilters.

Our basic block prefiltering works as follows. First, we identify *characteristic* basic blocks from the signature to use them as *starting points* for our bug search. That is, we focus on those basic blocks in the signature whose attributes are equal to at most t percent of the basic blocks in the target program. We denote the resulting set of starting points in the signature as SP. The lower the threshold t, the more basic blocks are filtered and, thus, the more efficient is the bug search, while the risk to miss relevant basic blocks increases. We use $t = 5\,\%$ in our experiments to reasonably trade off accuracy against scalability.

Then, for all basic blocks in SP, we search for *matching candidates* in the target program. That is, given a signature starting point $sp_i \in SP$, we calculate the fine-grained similarity (using the TED) to all basic blocks in the target program and choose the n basic blocks with the lowest distances. We denote this set of matching candidates as C_i, where i corresponds to the index of starting point sp_i. The lower n, the less often we need to invoke later stages of the bug signature matching, i. e., the lower the runtime. However, with lower n the risk to miss a good match rises. We chose $n = 20$ in our evaluation since we found in an empirical test that this value provides a good trade-off in practice.

We use the resulting set of starting points (and their matching candidates) in the following section to measure the signature's similarity to the target program (instead of only comparing single basic blocks).

3.2.4 Neighborhood Exploration

So far we have discussed how we compare binary code at the level of basic blocks. However, in order to find bugs, comparing individual basic blocks is not sufficient, as multiple basic blocks are typically characteristic for a bug. Furthermore, the *structure* of code is often also characteristic for a bug. Consider for example a typical buffer overflow bug where a certain basic block performing boundary check may be missing inside a memory-writing loop. Therefore, we further explore the CFG of a function in the neighborhood of individual basic block matches with the bug signature.

We do so in a CFG-driven, greedy, but locally optimal manner: In step i), we choose a pair of starting point ($sp_i \in SP$) and one of its matching candidates ($c_j \in C_i$) in the target program to initiate our search. We use the pair (sp_i, c_j) as the initial match, i. e., our algorithm considers it as a fixed pair and will not change their mapping. Once the algorithm terminates for this concrete pair, it is repeated for all other matching candidates of this starting point. In addition, we will repeat the algorithm with multiple starting points, i. e., using different initial pairs.

Then, in step ii), we compare the adjacent neighbors of the pair that has been fixed in step i). Adjacency in this context means that the basic blocks are either CFG-wise successors or predecessors of the fixed. When matching, we take the direction of the CFG into account, i. e., we do not compare preceding basic blocks with succeeding basic blocks, and vice versa. To this end, we compute the distances of all adjacent basic blocks. We then use a matching algorithm (the Hungarian algorithm [6]) to find ideal mappings between basic blocks adjacent to the previously fixed pair.

At this point, each optimally mapped pair of basic blocks (one in the signature, one in the target program), becomes a new candidate for *broadening* the overall match. We keep track of these candidates with a priority queue, sorted along the distance between the mapped pair. In step iii), we pick the best candidate pair (i. e., lowest distance) from the priority queue and greedily expand the already fixed pairs with this new pair. Usually, the newly fixed match will introduce further unexplored neighbors, such that the algorithm continues from step ii), while keeping track of already fixed basic blocks.

The algorithm terminates if the priority queue is empty, i. e., if no further neighbors can be explored. After terminating the search, the algorithm computes the distance between signature and target program by adding the TEDs for each fixed block pair. If basic blocks from the signature could not be matched (e. g., if the signature is larger than the target program) it adds the TED between unmatched blocks and an empty tree to the overall distance.

Our neighborhood exploration algorithm is greedy and avoids expensive backtracking steps. Given that we apply the algorithm for all pairs of starting point/matching candidate, there is a high chance to find reasonable matches between signatures and target programs. Our neighborhood exploration results in a sorted list of distances, revealing the code parts most similar to the signature, i. e., the bug.

4. IMPLEMENTATION

We now describe the implementation details that we have left out in the previous section for brevity reasons.

4.1 Program Preprocessing

TEDEM's preprocessing phase has two distinct phases, namely the *Disassembly and Structuring* (Section 4.1.1) and the *Semantic extraction* (Section 4.1.2).

4.1.1 Disassembly and Structuring

We use the Interactive Disassembler (IDA) to extract the disassembly and basic block structure from the programs. IDA supports many operating systems (e. g., Windows, Linux, Mac OS X). We chose to focus on disassembling x86 binaries given their high popularity. We make assumptions about data structures (e. g., register names and their widths), but with some engineering effort, our system could also be applied to x86-64 binaries.

Using IDA, for each (strict) basic block, we export the start address, the instructions, and record which function it belongs to. In addition, for each basic block, we use IDA to export the Control Flow Graph (CFG) to keep track of a list of succeeding and preceding basic blocks.

Note that we assume that binary programs are not obfuscated. Our use case is finding bugs in commodity software, which usually is not obfuscated. For software developers, there is also no real benefit to evade our system, so we think this is a reasonable assumption. Clearly, our approach fails for any obfuscated software, as obfuscation destroys most of the CFG and the syntactical information. However, we consider (typically obfuscated) malware out of scope, because

we primarily aim to secure legitimate software by finding locations that should be patched.

4.1.2 Semantic Extraction

We then transform the output of IDA into the intermediate representation (IR) that is part of METASM [9]. We chose METASM because it can symbolically accumulate assembly instructions in a compact and reasonably accurate form. In order to capture the effect of a basic block on the state of a program, namely registers, memory, branch conditions an successors, we use METASM's capability to accumulate the computational steps of a given sequence of assembler instructions in the form of compact equations. An equation may look like this: $eax := Ind(4, esi + edx * 4) + ebx * 2$, where $Ind(x, y)$ refers to the x-byte value at memory address y. These equations map the basic block's input state to its output state, condensing its semantic effect.

METASM uses typed equations which allow symbolic simplifications. We use easy simplifications such as constant folding, but also perform more complicated steps, like accumulating arithmetic/bitwise operations, to simplify, shorten, and normalize the expression. We further support machine-specific idioms such as `xor eax, eax` in order to reduce bogus dependencies on previous computations. In this concrete example, we model that `eax` becomes zero and we are not interested in the previous values of the register (although they, in principle, are used by `xor`).

We then convert these equations into S-Expressions. An S-Expression is a notation for a tree-like data structure, where a tree is noted as a bracketed, whitespace-separated list of its children. A child may be bracketed itself, denoting that it is a sub tree. For example, the equation above as an S-Expression is represented as
$(:= \ eax \ (+ \ (Ind \ 4 \ (+ \ esi \ (* \ edx \ 4))) \ (* \ ebx \ 2)))$.

4.2 Signature Matching

The actual signature matching consists of two separate steps. First, we find candidate basic blocks in the target program, which are very similar to a distinctive block from the signature (Section 4.2.1). Second, we explore the matches' neighborhoods for other good matches (Section 4.2.2).

4.2.1 Basic Block Comparison

One of our core ideas is using a TED to compare two basic block based on their lists of equations. We explored several notions of tree edit distances. The basic edit distance algorithm for trees performs node-wise operations only.

Tekli et al. [26] not only pre-compute costs for operations on subtrees with the dynamic programming approach used for classic string edit distances, but also add a notion of subtree commonality, which allows insertion/deletion of formerly adjusted subtrees. We chose this algorithm, as it recognizes and utilizes similarity in subtrees at a runtime of $\mathcal{O}(n^2)$, where n is the number of tree nodes. In the context of our work, tree nodes are subexpressions, i. e., operands and operations of an equation.

We ported the TED as proposed by Tekli et al. to C++. Figure 2 shows an example of the TED: In order to transform the left tree to the right tree, the TED first has to replace the left-most operand (`eax` by `edx`) and delete an operand ("2"). The power of the algorithm is demonstrated by the last edit operation, in which the common subtree (dashed square) replaces an operator ("+"). An algorithm with only node-wise manipulations would need to perform multiple single steps to move the subtree.

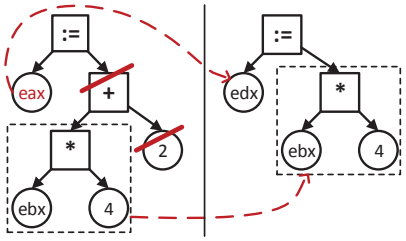


Figure 2: Exemplary tree edit distance with subtree-edits

As mentioned before, the equations that define a basic block's behavior are independent from one another. Because of the explicit computation order defined by the tree structure, the same is true for subexpressions. This allows reordering for equations and also for subexpressions (at least if the operation is commutative). We define an arbitrary but deterministic order of the elements and sort the trees accordingly, such that we avoid edit operations for basic blocks that only differ in the sequence of equations or elements.

4.2.2 Neighborhood Exploration

Based on these basic block-wise distances, we implemented the algorithm to match the bug signature (as described in Section 3.2.4). This section describes important performance impacts of the concrete implementation.

First, the *matching algorithm* finds a mapping between basic blocks to minimize the sum of all mapped distances. The implemented *Hungarian Method* [6] has a runtime of $\mathcal{O}(n^3)$ to match n nodes. Because we are only matching neighbors of one basic block from the signature with the neighbors of its candidate match in the target program, n is quite small — usually not larger than two. Apart from CFG considerations, this shows once more how important it is to operate the matching algorithm locally.

Instead, the computation of the basic block-wise edit distance dominates the runtime, especially when searching for promising starting points for the neighborhood exploration. Thus, we implemented a parallel search for starting points and matching candidates. In principle, also the other parts of our approach can run in parallel. For example, the signature matching algorithm can trivially be parallelized, as the metric splits up independently at the level of basic blocks — which we leave open to future work.

5. EVALUATION

We implemented a prototype of TEDEM that we evaluated on a variety of real-world bug examples. We group our test cases into three scenarios: (i) The identification of *shared code bugs* between related programs sharing one codebase; (ii) The identification of *fork bugs* between different programs sharing a (partly) similar codebase; (iii) The identification of *copy'n'paste bugs* in a single program.

To build our test cases, we first identified known vulnerabilities in binary software using IDA and *BinDiff*. We relied on these tools at this point because the initial identification of patched locations is out of scope for our work. We then manually created a signature for each test case. In most cases, using the entire vulnerable function as "easy" signature produced insightful and precise results. Such a signa-

App.	Bug/Patch	Sign.	Target	Rank #
Pidgin	CVE-2013-6484	51/ 51	Adium	110/ 6,019
		11/ 51		1/ 6,019
Adium		42/ 42	Pidgin	18/ 3,965
		11/ 42		1/ 3,965
Pidgin	CVE-2013-6485	131/131	Adium	1/ 6,019
Word 2010	MS13-072	59/ 59	Word 2003	1/28,564
			Word 2007	1/44,626
			Word 2013	1/21,624
			Comp. Tool	1/13,035
			Word Viewer	1/16,613
PuTTY	CVE-2011-4607	980/980	PuTTYcyg	1/ 1,382
			TuTTY	1/ 1,651
PuTTY	CVE-2013-4208	37/ 37	PuTTYcyg	12/ 1,382
			TuTTY	1/ 1,651
		33/ 33	PuTTYcyg	4/ 1,382
			TuTTY	1/ 1,651
ImageM.	CVE-2009-1882	613/613	GraphicsM.	118/ 2,815 ‡
ImageM.		1/613	GraphicsM.	1/ 2,815 ‡
GraphicsM.		210/210	ImageM.	1/ 3,299
Pidgin	CVE-2011-4601	5/ 19	Pidgin	1/ 1,242 †
ProFTPD	Backdoor	4/336	ProFTPD	4/ 1,711
		3/336		1/ 1,711

Table 1: Overview of evaluation results. The size of signatures is given in the form <*number used BBs/total BBs in vuln. function*>. The rankings marked with † and ‡ are further explained in Sections 5.2.2 and 5.3.1 respectively.

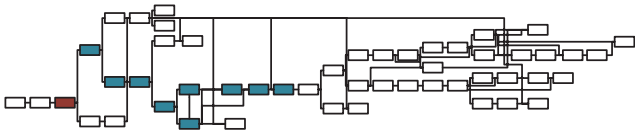


Figure 3: CFG (left to right, strict basic blocks) of `reply_cb()` in `Pidgin/libpurple` v2.10.7. The signature used to search for CVE-2013-6484 is highlighted. The patched block is highlighted on the far left.

ture can in certain cases be too broad, though, as it contains *noise* unrelated to the vulnerability, such as generic function prologues and epilogues that vary between different compilers and settings. Especially for smaller functions, signatures spanning entire functions can thus be too noisy/generic.

If not explicitly stated differently in the following, we used entire functions as signatures in our experiments. We chose to use entire functions as then the manual effort to create such signature is minimal — the analyst only needs to know the name/address of the vulnerable function. In principle, more fine-grained signatures obtain more accurate results. However, defining "good" signatures is a subjective process. Our choice to match entire functions allows for better comparability and repeatability of our work and guarantees sound experiments. We also perform experiments with fine-grained signature to show their strengths.

The evaluation results are summarized in Table 1 and we give detailed descriptions of the experiments in the following.

5.1 Shared Code Bugs

A typical use case for finding bug doublets arises when source code is shared among multiple software projects. Thus, we evaluate two such use cases in this section.

5.1.1 Libpurple

`Pidgin` v2.10.8 (2014-01-28) closes several vulnerabilities in libpurple. Among these are CVE-2013-6484 and CVE-2013-6485, two unrelated vulnerabilities that allow a remote attacker to crash a running Pidgin instance due to insufficient input verification. We used TEDEM to search for these two vulnerabilities in the most recent version of `Adium` (v1.5.9), a popular messenger for `Mac OS X` that also ships libpurple in compiled form. We manually confirmed that Adium still suffered from the two vulnerabilities that had already been patched in `Pidgin`.

For CVE-2013-6484, we first used the whole vulnerable function (51 basic blocks) in `Pidgin` as signature to search for the corresponding function in `Adium` (42 basic blocks). The size difference of the two functions resulted from `Pidgin` inlining a function. Given this coarse signature, TEDEM ranked the corresponding function in `Adium` (42 basic blocks) #18. On a relative scale this means that the analyst needs to manually inspect less than 0.5% of the functions in order to find the vulnerable code part. Next, we manually selected a signature of 11 basic blocks from the 51 basic blocks of the vulnerable function in `Pidgin` as depicted in Figure 3: Starting with the basic block whose branch condition was patched, a handful of succeeding basic blocks are selected, which validate network-received data. For this more fine-grained signature TEDEM ranked the corresponding function in `Adium` even #1.

For CVE-2013-6485, we used the entire unpatched function (131 basic blocks) from `Pidgin` as signature and unambiguously identified the corresponding function in `Adium` (136 basic blocks) at rank #1.

We also examined the other functions that TEDEM ranked high. We found that many of the functions were comparably exposed parsing network data. E. g., for CVE-2013-6485, a function that was previously reported to suffer from a similar memory corruption vulnerability[1] ranked #6.

This example across OS-boundaries also highlights the robustness of our approach in comparison to tools that largely rely on a coherent static call graph like *BinDiff*: When comparing the Windows and Mac versions of libpurple, *BinDiff* (v4.0.1) fails to correctly match the vulnerable functions to their counterparts. This is probably due to both functions being callbacks with no parents in the static call graph. However, even though our approach itself is not specific for a single OS, a particular bug signature may be very well.

5.1.2 Microsoft Word

Microsoft patched 13 vulnerabilities in various versions of `Word` for Windows[2]. We manually identified one of the patched locations in `Word 2010`, the most recent affected application. In `WRD12CNV.DLL` (5.21 MB) an additional boundary check on an array-index was introduced inside a loop in a certain function (virtual address `319421E7h`, 59 basic blocks). Apparently, this check tackles a memory corruption vulnerability. Given the patched function in `Word 2010` as signature, TEDEM unambiguously identified the equivalent function in all other patched applications (rank #1): `Word 2003` (`WINWORD.EXE`, 11.7 MB), `Word 2007` (`WWLIB.DLL`, 17.2 MB), `Office Compatibility Pack` (`WRD12CNV.DLL`, 4.46 MB), and `Word Viewer` (`WORDVIEW.EXE`, 8.4 MB). Also, TEDEM correctly highlighted an interesting circumstance: All other applications' *patched* versions were more similar to `Word 2010` *unpatched* than to `Word 2010` *patched*. Indeed, we could manually confirm that the other applications had been patched differently than `Word 2010`. Checks were added that were already present in `Word 2010` *unpatched*. The

[1] `http://www.pidgin.im/news/security/?id=34`
[2] MS13-072: `https://technet.microsoft.com/library/security/ms13-072`

array-index boundary check introduced in `Word 2010` *patched* was still missing in the *patched* other applications, though. The identification of this possibly critical difference in patch levels is an exemplary use case for TEDEM. Lastly, using the same signature, we could also pinpoint the corresponding function in a more recent version of `Word 2013` (`WRD12CNV.DLL`, 6.01 MB). We found the same checks to be present as in `Word 2010` *patched*.

We found that *BinDiff* repeatedly matched wrong functions between the different applications and between the different versions of the same application. For example, *BinDiff* could not identify the discussed patched function in `Word 2003`, neither through incremental version diffing nor through cross-application diffing with `Word 2010`.

5.2 Fork Bugs

When software projects are *forked*, it can be difficult to tell if a bug also affects forked projects. We evaluate TEDEM's ability to pinpoint bugs in two forked applications based on signatures extracted from their parent applications.

5.2.1 PuTTY

`PuTTY` is an open source SSH client for Windows. Prior to v0.62, passwords were not wiped from memory (CVE-2011-4607) and until v0.63 the same was the case for certain cryptographic data (CVE-2013-4208). We examined the `PuTTY` forks `TuTTY` v0.60.2.0 and `PuTTYcyg` r20101029 for the presence of these bugs using TEDEM.

Given the vulnerable function of CVE-2011-4607 (980 basic blocks) in `PuTTY` v0.61 as signature, TEDEM ranked the corresponding functions in `TuTTY` and `PuTTYcyg` #1. The bug described in CVE-2013-4208 is contained in two functions in `PuTTY` v0.61. Given these two functions as signatures, TEDEM ranked the corresponding functions #1 in `TuTTY`. For `PuTTYcyg`, we could only achieve ranks #12 and #4, because the CFG was heavily altered, e. g., due to `memcmp()` being inlined. However, these ranks are still in the top 1% and could thus be helpful to an analyst.

5.2.2 ImageMagick

`ImageMagick` is an open source image processing tool. Prior to v6.5.2-9, it suffered from an integer overflow bug in TIFF image processing code (CVE-2009-1882). The tool `GraphicsMagick` was forked from `ImageMagick` as early as 2002 and is still actively developed. We used TEDEM to check for the presence of the bug in `GraphicsMagick` v1.3.7.

Given the vulnerable function from `ImageMagick` v6.5.1-2 (613 basic blocks) as signature, TEDEM ranked the corresponding function in `GraphicsMagick` (210 basic blocks) only #118. This is due to a large and dominating function being inlined into `ImageMagick`'s vulnerable function. In turn, this inlined function's counterpart in `GraphicsMagick` comes #1 in the similarity ranking produced by TEDEM in the reverse direction. This function is (statically) called by only the vulnerable function in `GraphicsMagick`. Accordingly, it was easily possible to pinpoint the vulnerable function in `GraphicsMagick` given TEDEM's output. In general, this one-directional behavior of finding the bigger function with the inlined component, but not vice versa, is what one would expect in the face of inlining and shows that our approach tolerates inlining to some extent.

However, when we compile a signature of only a single basic block from `ImageMagick`, namely the one with the integer overflow, we can also rank the corresponding function in `GraphicsMagick` on #1.

Dist.	Function name
0.0	receiveauthgrant
0.0	parseicon
13.8	receiveauthrequest
21.8	receiveadded
23.7	receiveauthreply
36.9	memrequest
98.8	parseadd
120.8	msgack
150.3	clientautoresp
176.8	oscar_buddy_menu
190.1	oscar_get_ex..._status
202.7	purple_bosrights
215.3	purple_odc_send_im
252.2	incomingim

Dist.	Function name
260.9	infochange
284.0	oscar_send_im
292.5	find_acct
304.8	oscar_status_types
382.2	oscar_actions
395.2	parsemod
399.6	aim_info_extract
451.9	aim_request_login
566.0	infochange
571.1	oscar_..._display_icq
592.5	oscar_auth_sendrequest
596.3	aim_putsnac
599.4	purple_ssi_authreply
635.0	oscar_...and_status

Table 2: Functions in `Pidgin/libpurple` with the lowest distance to our signature extracted from CVE-2011-4601. Marked functions lack the necessary check.

5.3 Copy'n'Paste Bugs

Another source for bug doublets is caused by developers that adopt faulty source code and miss to patch it in case of upstream bugs. We evaluate TEDEM for two such cases.

5.3.1 Libpurple

We evaluated the same vulnerability (CVE-2011-4601) in `Pidgin`'s libpurple on binary level as Yamaguchi et al. did on source code level [28]: Prior to v2.10.0, a missing check in the function `receiveauthgrant()` enabled remote attackers to crash `Pidgin`. Yamaguchi et al. identified nine other functions suffering from the same vulnerability.

We compiled `Pidgin` v2.10.0 on Linux with default settings, but without optimization, because otherwise several vulnerable functions were inlined into one particular function. This would have made results incomparable. To exclude obvious boilerplate code, we chose only the immediate neighbor-blocks to the actual vulnerability, ending up with a signature of 5 of the 19 basic blocks in the original vulnerable function `receiveauthgrant()`. Table 2 lists those functions from the roughly 2360 functions in the binary libpurple that TEDEM identified as most similar to the signature. Indeed Table 2 resembles very much the listing reported by Yamaguchi et al. for their tool. For example, they also found 9 among the thirty best ranked functions to lack the necessary check. Interestingly, we found three functions that Yamaguchi et al. did not report on and vice versa. Overall, this demonstrates the ability of TEDEM to compete with source code based tools in terms of quality of results.

5.3.2 ProFTPD (Backdoor)

In a variant of `ProFTPD` v1.3.3c, a backdoor was found in a handler function for the *HELP* FTP command. The backdoor gave privileged system access to anyone issuing the command with a certain argument. We constructed an artificial use case from this incident by installing a similar backdoor in the handler of the *MKD* FTP command. Using the original backdoor (4 basic blocks) as signature, we used TEDEM look for the other backdoor in a `ProFTPD` binary compiled on Linux with default settings. TEDEM ranked the manipulated handler for *MKD* (78 basic blocks) #4. The manipulated handler ranked #1 for a more fine-grained signature composed of 3 basic blocks. This example again highlights that matching based on fine-grained signa-

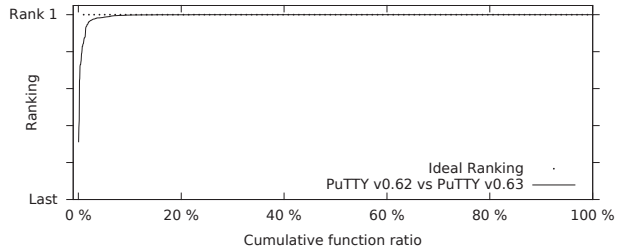


Figure 4: Rankings of function-level matching.

tures can in our use case often be more powerful than simple function-level matching.

5.4 Systematic Function-Level Matching

Lastly, we measure the overall applicability of our approach to match functions. Although function matching is not our considered use case, such a systematic experiment allows to illustrate the generality of our method. We chose to compare two similar binaries for which we have a ground truth of functions that should be mapped. In particular, we compare the binaries of two subsequent PuTTY versions (v0.62 and v0.63) that differ only slightly. We use their symbolic information to create the ground truth of functions that should be mapped. We do not use symbolic information as part of our algorithm, i.e., we use stripped binaries.

Figure 4 shows a CDF of the ranks when mapping the functions in PuTTY v0.62 in v0.63. Our experiments show that 77.8% of the functions were correctly mapped (rank #1). Only 6% of the functions ranked badly (rank > 20). We manually inspected the reasons for these non-optimal rankings and found that some of them were caused by slight modifications in the two PuTTY versions. Others were caused by functions that substantially changed their CFG, e. g., functions that had significantly more basic blocks than their counterpart. However, overall, the experiment shows that our method generally produces reliable matches and works beyond the concrete examples discussed above.

6. DISCUSSION AND FUTURE WORK

TEDEM can leverage semantic signatures on a binary level to find similar bugs in other programs or other functions within the same program. In this section we discuss limitations and future work directions of our work.

False Positives. Our distance metric can compare arbitrary divergent code. If the target program does not contain a vulnerable function, our approach still produces a ranking of functions with the distance to the signature. However, the distances are significantly larger than the distances of similar code. We will investigate if there are reasonable mechanisms to determine meaningful thresholds that can filter out false positives in future work.

Robustness. Although the evaluation has shown that our approach works well in most practical examples, we have to acknowledge our method's sensitivity to changes in binary code. This section discusses many other typical binary code modifications that our system design tolerates up to some degree. However, we exclude obfuscated binaries from the scope of our method explicitly.

Using a TED-based similarity metric allows to compare code even if it slightly changed. For example, although register renaming or register spilling would slightly increase the TED, the overall similarity remains high. This is also true if the source code was slightly modified (e.g., inserted instructions, change of constants, global variables, etc.). We can also deal with instruction reordering, as we use a deterministic ordering of the equations and their sub-expressions. Lastly, also function inlining is not problematic, because we dynamically broaden a match and thus can also match subgraphs (and not only entire functions). This increased robustness against syntactical changes is thanks to our accumulation and normalization into semantic representations.

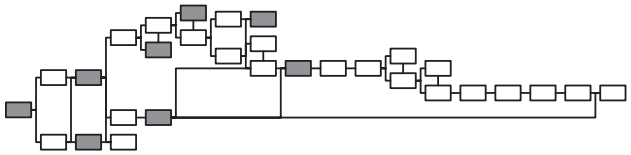


Figure 5: CFG (strict basic blocks, left to right) of *Aurora bug* constructor. Good matches from the copy constructor are highlighted. They are obviously not connected densely.

However, our bug search requires that some neighborhood properties are preserved in the CFG. For example, we found that CVE-2010-0249, the bug used in *Operation Aurora*, violates this requirement. The vulnerability was a *use-after-free* bug in Internet Explorer versions 6–8 that allowed remote code execution. A constructor and copy constructor of a C++ class were patched in a similar way to correctly update reference counters in other objects. The regular constructor is far larger than the copy constructor (26 vs. 8 basic blocks). Figure 5 shows that matching basic blocks are hardly connected to each other, which causes the neighborhood exploration algorithm to miss the constructor given the unpatched copy constructor as bug signature.

Many of the aforementioned syntactical changes can be induced by typical optimization strategies of compilers. However, note that this is also only relevant if the optimization strategy differs between signature and target program.

Fine-Tuning Bug Signatures. Our experiments are based on function-level signatures. As we have shown, fine-granular bug signatures can be much more accurate for identifying buggy code locations. As such, defining more specific bug signatures is a powerful way to boost the accuracy.

Another possible way to increase the accuracy is assigning weights to individual parts of the signature. Weights allow us to respect gradual importance of specific structures, e. g., giving the change in a code location constant or the use of a different register less importance than, say, changing the operand of a computation.

Lastly, creating bug signatures could be automated. For example, an analyst could flag vulnerable source code parts, compile the code, and use debug information (i.e., source code location information) to map the vulnerable source code to its binary code representation.

Finding Patched Code. Typically, patched code is quite similar to unpatched code. In general, it is not trivial to determine if the functionality that we find to be similar to our signature is actually vulnerable. As of now, we cannot determine if the conditions allowing an exploitation are satisfiable. However, first experiments have shown that we can define an additional signature spanning the patched code, which we can use for false positive testing. Intuitively, the patched signature matches more closely to patched binaries than to unpatched ones. By comparing both similarity scores it is thus possible to distinguish patched from unpatched binary code, as shown for `MS Word` (Section 5.1.2).

Signature	Target	Signature Size	Candidate Search	Neighborh. Exploration
ProFTPD	ProFTPD	3 BBs	1.5 s	0.02 s
PuTTY	TuTTY	14 BBs	11.6 s	1.12 s
PuTTY	PuTTYcyg	13 BBs	11.4 s	1.32 s
Libpurple	Adium	51 BBs	349.7 s	7.95 s
Word 2010	Word 2013	58 BBs	1043.1 s	34.32 s
GraphicsM.	ImageM.	210 BBs	1332.9 s	116.61 s
PuTTY	TuTTY	978 BBs	2482.5 s	1958.13 s

Table 3: Runtimes of signature searches

Scalability. The size of the signature influences runtime linearly. For our targeted use case of fine-grained search with small signatures, we achieve reasonable runtimes of less than a minute (see Table 3) on a workstation with Intel Core i7-2640M @ 2.8GHz with 8GB DDR3-RAM. Clearly, the search for initial candidate matches dominates the runtime. For larger signatures, like in the extreme case of `PuTTY` (978 basic blocks), the runtime rises to about 42 min. However, 33 min have to be accounted to the candidate search, as many pairwise basic block distance computations have to be performed. In contrast, the neighborhood exploration algorithm has a relatively low overall runtime, although it is invoked very often due to a high number of starting points and their matching candidates. For example, in `PuTTY`, the algorithm had to be invoked 19,560 times (978 basic blocks times $n = 20$ matching candidates), but the runtime added up to 6.5 min with an average of 20 ms per invocation.

The size of the target program influences the runtime of the candidate search linearly. However, as we have seen, e.g., for `MS Word`, TEDEM is still scalable even on commodity user hardware. In future work we will look into methods how to further improve the performance. For example, an M-Tree, a data structure for fast k-nearest neighbor search in metric spaces [8], could speed up the candidate search.

7. CONCLUSIONS

We have shown that re-finding similar bugs on the level of binary code is possible. By this we enhance existing research that aims to find similar code bugs on the source code level. Once a bug is known to an analyst, our concept of semantic signatures can be applied to search for similar bugs in the same binary or even in different programs. We searched for various real-world bugs (caused by forks, shared code, or copy'n'paste errors) with promising accuracy.

Acknowledgements

This work has been supported by the German Federal Ministry of Education and Research (BMBF) under support code 16BP12302, EUREKA-Project *SASER* and by the German Research Foundation (DFG) research training group *UbiCrypt* (GRK 1817).

8. REFERENCES

[1] F. Ahmed, H. Hameed, M. Z. Shafiq, and M. Farooq. Using Spatio-temporal Information in API Calls with Machine Learning Algorithms for Malware Detection. In *AISec '09*.

[2] T. Avgerinos, S. K. Cha, B. L. T. Hao, and D. Brumley. AEG: Automatic Exploit Generation. In *NDSS '11*.

[3] S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley. Unleashing Mayhem on Binary Code. In *IEEE S&P*, 2012.

[4] V. Chipounov, V. Kuznetsov, and G. Candea. S2E: A Platform for In-vivo Multi-path Analysis of Software Systems. In *ASPLOS XVI*, 2011.

[5] T. Dullien and R. Rolles. Graph-based Comparison of Executable Objects. *SSTIC*, 5, 2005.

[6] A. Frank. On Kuhn's Hungarian Method – A Tribute from Hungary. Technical report, Oct. 2004.

[7] D. Gao, M. K. Reiter, and D. Song. BinHunt: Automatically Finding Semantic Differences in Binary Programs. In *Info. and Comm. Security*. 2008.

[8] F. Gauthier, T. Lavoie, and Merlo. Uncovering Access Control Weaknesses and Flaws with Security-discordant Software Clones. In *ACSAC '13*.

[9] Y. Guillot and A. Gazet. Automatic Binary Deobfuscation. *Journal in Comp. Virology*, 2010.

[10] I. Haller, A. Slowinska, M. Neugschwandtner, and H. Bos. Dowsing for Overflows: A Guided Fuzzer to Find Buffer Boundary Violations. In *USENIX Sec. '13*.

[11] J. Jang, A. Agrawal, and D. Brumley. ReDeBug: Finding Unpatched Code Clones in Entire OS Distributions. In *IEEE S&P*, 2012.

[12] L. Jiang, G. Misherghi, Z. Su, and S. Glondu. DECKARD: Scalable and Accurate Tree-Based Detection of Code Clones. In *ICSE*, 2007.

[13] W. Jin, S. Chaki, C. Cohen, A. Gurfinkel, J. Havrilla, C. Hines, and P. Narasimhan. Binary Function Clustering Using Semantic Hashes. In *IEEE ICMLA 2012*.

[14] T. Kamiya, S. Kusumoto, and K. Inoue. CCFinder: A Multilinguistic Token-based Code Clone Detection System for Large Scale Source Code. *IEEE Trans. Softw. Eng.*, 28(7), July 2002.

[15] C. Kruegel, E. Kirda, D. Mutz, W. Robertson, and G. Vigna. Polymorphic Worm Detection Using Structural Information of Exec. In *RAID'05*.

[16] A. Lakhotia, M. D. Preda, and R. Giacobazzi. Fast Location of Similar Code Fragments Using Semantic 'Juice'. In *ACM PPREW 2013*.

[17] Z. Li, S. Lu, S. Myagmar, and Y. Zhou. CP-Miner: A Tool for Finding Copy-paste and Related Bugs in OS Code. In *USENIX OSDI '04*.

[18] C. McMillan, M. Grechanik, and Poshyvanyk. Detecting Similar Software Applications. In *ICSE '12*.

[19] J. Ming, M. Pan, and D. Gao. iBinHunt: Binary Hunting with Inter-Proced. Ctrl Flow. In *ICISC'12*.

[20] G. Myles and C. Collberg. K-gram Based Software Birthmarks. In *ACM SAC '05*.

[21] S. Neuhaus, T. Zimmermann, C. Holler, and A. Zeller. Predicting Vulnerable Software Components. In *ACM CCS 2007*.

[22] J. Newsome and D. Song. DTA for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software. In *NDSS '05*.

[23] B. H. Ng and A. Prakash. Expose: Discovering Potential Binary Code Re-use. In *COMPSAC 2013*.

[24] P. Oehlert. Violating Assumptions With Fuzzing. *IEEE S&P '05*.

[25] E. J. Schwartz, T. Avgerinos, and D. Brumley. All You Ever Wanted to Know About Dynamic Taint Analysis and Forward Symbolic Execution (but Might Have Been Afraid to Ask). In *IEEE S&P '10*.

[26] J. Tekli, R. Chbeir, and K. Yetongnon. Efficient XML Structural Similarity Detection using Sub-tree Commonalities. In *SBBD*, 2007.

[27] X. Wang, N. Zeldovich, M. F. Kaashoek, and A. Solar-Lezama. Towards Optimization-safe Systems: Analyzing the Impact of Undefined Behavior. In *ACM SOSP 2013*.

[28] F. Yamaguchi, M. Lottmann, and K. Rieck. Generalized Vulnerability Extrapolation Using Abstract Syntax Trees. In *ACSAC 2012*.

IntFlow: Improving the Accuracy of Arithmetic Error Detection Using Information Flow Tracking

Marios Pomonis
Columbia University
mpomonis@cs.columbia.edu

Theofilos Petsios
Columbia University
theofilos@cs.columbia.edu

Kangkook Jee
Columbia University
jikk@cs.columbia.edu

Michalis Polychronakis
Columbia University
mikepo@cs.columbia.edu

Angelos D. Keromytis
Columbia University
angelos@cs.columbia.edu

ABSTRACT

Integer overflow and underflow, signedness conversion, and other types of arithmetic errors in C/C++ programs are among the most common software flaws that result in exploitable vulnerabilities. Despite significant advances in automating the detection of arithmetic errors, existing tools have not seen widespread adoption mainly due to their increased number of false positives. Developers rely on wrap-around counters, bit shifts, and other language constructs for performance optimizations and code compactness, but those same constructs, along with incorrect assumptions and conditions of undefined behavior, are often the main cause of severe vulnerabilities. Accurate differentiation between legitimate and erroneous uses of arithmetic language intricacies thus remains an open problem.

As a step towards addressing this issue, we present *IntFlow*, an accurate arithmetic error detection tool that combines static information flow tracking and dynamic program analysis. By associating sources of untrusted input with the identified arithmetic errors, IntFlow differentiates between non-critical, possibly developer-intended undefined arithmetic operations, and potentially exploitable arithmetic bugs. IntFlow examines a broad set of integer errors, covering almost all cases of C/C++ undefined behaviors, and achieves high error detection coverage. We evaluated IntFlow using the SPEC benchmarks and a series of real-world applications, and measured its effectiveness in detecting arithmetic error vulnerabilities and reducing false positives. IntFlow successfully detected all real-world vulnerabilities for the tested applications and achieved a reduction of 89% in false positives over standalone static code instrumentation.

Categories and Subject Descriptors

D.4.6 [**Operating Systems**]: Security and Protection; D.2.5 [**Software Engineering**]: Testing and Debugging

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.
ACSAC '14, December 08–12 2014, New Orleans, LA, USA
Copyright 2014 ACM 978-1-4503-3005-3/14/12 . . . $15.00
http://dx.doi.org/10.1145/2664243.2664282 .

General Terms

Security, Reliability

Keywords

Static analysis, information flow tracking, arithmetic errors

1. INTRODUCTION

When developing programs in the C and C++ languages, programming practices that involve undefined arithmetic operations often constitute a well-established status quo. Compilers tolerate the use of undefined behavior, as this enables code optimizations that greatly increase the performance of critical code segments. Consequently, programmers often purposely rely on undefined language constructs due to empirical certainty for certain expected outcomes. This flexibility comes at a cost: arithmetic operations constitute a major source of errors, often leading to serious security breaches when erroneous values directly or indirectly affect sensitive system calls or memory operations.

The root of the problem is fundamentally bound to the differences between the mathematical and machine representations of numbers: although integer and floating point numbers are infinite, their machine representations are restricted by their respective type-specific characteristics (e.g., signedness and bit-length). Furthermore, not all arithmetic operations are well-defined by the language standards, to allow for a number of compiler optimizations. For instance, the standard does not specify what the value of a signed integer that overflows (or underflows) should be.

As it is non-trivial to determine whether a particular arithmetic operation resulting in undefined behavior is benign or not, bugs due to integer errors are prevalent. Integer errors are listed among the 25 most dangerous software bugs [3], and are often the root cause of various vulnerabilities such as buffer overflows [25] and memory disclosures [2]. During the past years, numerous attempts have been made towards their automatic detection and prevention. Such efforts include static [23] and dynamic [7] analysis solutions, tools based on symbolic execution and dynamic test generation [8, 17], as well as compiler extensions [25] that resolve ambiguities at compilation time.

Despite numerous suggested solutions, there is no generic tool that provides effective and complete detection and sanitization of integer bugs. One reason is that tools that focus on coverage typically generate a large amount of false positives (due to the inaccuracy of static analysis) [25, 23, 16],

while those that focus on accuracy provide poor coverage (as they rely on dynamic analysis or dynamic test generation) [10, 17, 19, 20]. Furthermore, existing tools typically focus only on certain integer error classes (mainly overflows and underflows) and as a result they do not provide broad spectrum protection. The most crucial reason, however, is the inherent difficulty of prevention mechanisms to differentiate between critical integer errors that may lead to exploitable vulnerabilities, from *intentional* uses of wrap-around behavior, type castings, bit shifts, and other constructs that serve application-specific purposes.

As a step towards addressing these issues, in this paper we propose an approach that combines static code instrumentation with information flow tracking to improve the accuracy of arithmetic error detection, focusing on reducing the number of false positives, i.e., developer-intended code constructs that violate language standards. Our tool, IntFlow, uses information flow tracking to reason about the severity of arithmetic errors by analyzing the information flows related to them. The main intuition behind this approach is that arithmetic errors are critical when i) they are triggered by or depend on values originating from untrusted locations, or ii) a value affected by an arithmetic error propagates to sensitive locations, such as the arguments to functions like `malloc()` and `strcpy()`.

To demonstrate the effectiveness of our approach, we evaluated IntFlow with real world programs and vulnerabilities and verified that it successfully identifies all the real world vulnerabilities for the applications of our testbed, generating 89% less false positives compared to IOC [10], our reference arithmetic error detection tool.

Our work makes the following contributions:

- We present an accurate arithmetic error detection approach that combines static information flow tracking and dynamic program analysis.
- We present IntFlow, our prototype implementation for this approach, which operates as an LLVM add-on. IntFlow is freely available as an open source project.[1]
- We evaluate IntFlow using real-world programs and vulnerabilities. Our results demonstrate that IntFlow achieves improved detection accuracy compared to previous solutions, as it suppressed more than 89% of the false positives reported by IOC [10].

2. BACKGROUND

To effectively deal with integer errors in real world applications, it is necessary to first define what is considered an error. Doing so is not trivial, as apart from the mere examination of conformance to the language standard, we must also examine whether pieces of seemingly erroneous code—from the perspective of the language specification—are explicitly written in that way due to the developer's intention, typically for performance, convenience, or other reasons.

In this section, we discuss how the C/C++ language standards define correctness for arithmetic operations, and examine why developers often write code that deviates from the language specification. We also present examples of exploitable vulnerabilities caused by integer errors, and demonstrate the importance of good programming practices.

[1] http://nsl.cs.columbia.edu/projects/intflow/

2.1 Integer Errors and Undefined Behavior

Although the C and C++ language standards explicitly define the outcome of most integer operations, a number of corner cases are left undefined. As an example, the C11 standard considers an `unsigned` integer overflow as a well-defined operation, whose result is the minimum value obtained after the wrap-around, while leaving `signed` integer overflows undefined. This choice facilitates compiler implementations to handle them in a way that produces optimized binaries [24]. For instance, signed integer overflows (or underflows) enable compiler developers to implement an optimization that infers invariants from expressions such as `i+1 > i` and replaces them with a constant Boolean value [5].

Table 1 lists special cases of integer operations and their definedness. It should be noted that although more instances of undefined behavior (not necessarily restricted to integer operations) are declared in the language specification, we only consider those relevant to this work.

Arithmetic Operation	Definedness
Unsigned overflow (underflow)	defined
Singed overflow (underflow)	undefined
Signedness conversion	undefined*
Implicit type conversion	undefined*
Oversized/negative shift	undefined
Division by zero	undefined

*if value cannot be represented by the new type

Table 1: Summary of defined and undefined arithmetic operations according to the C/C++ language specification.

As in practice not all cases of undefined behavior necessarily result in actual errors, the difficulty of dealing with these types of bugs lies in distinguishing *critical* integer errors from *developer-intended* violations of the standard. The intention of a developer, however, cannot be formally defined or automatically derived, as the code patterns used in a piece of code are deeply related to the author's knowledge, preference, and programming style.

Although writing code that intentionally relies on undefined operations is generally considered a bad programming practice (as the outcome of those operations can be arbitrary, depending on the architecture and the compiler), there are several cases in which the community has reached consensus on what is the expected behavior of the compiler in terms of the generated code, mainly due to empirical evidence. This explains why idioms that take advantage of undefined behavior are still so prevalent: although *according to the standard* the result of an operation is undefined, developers have an empirically derived expectation that compilers will always handle such cases in a consistent manner.

This expectation creates serious complications whenever developers check the validity of their code with state-of-the-art static analysis tools. These tools evaluate code based on strict conformance to the language specification, and consequently generate a large amount of false positives. Thus, the generated reports are often overlooked by developers who struggle to spot which of the reported bugs are actual errors. Unfortunately, tools based on dynamic code analysis also do not provide strong guarantees in these cases, as they suffer from low code coverage.

```
1  UINT_MAX = (unsigned) −1;
2  INT_MAX = 1 << (INT_WIDTH − 1) − 1;
```

Listing 1: Widely used idioms that according to the standard correspond to undefined behavior.

```
1  /* struct containing image data, 10KB each */
2  img_t *table_ptr;
3  unsigned int num_imgs = get_num_imgs();
4  ...
5  unsigned int alloc_size = sizeof(img_t) * num_imgs;
6  ...
7  table_ptr = (img_t*) malloc(alloc_size);
8  ...
9  for (i = 0; i < num_imgs; i++)
10         { table_ptr[i] = read_img(i); } /* heap overflow */
```

Listing 2: An unsigned integer overflow as a result of a multiplication (line 5), which results in an invalid memory allocation (line 7) and unintended access to the heap (line 10).

To further illustrate the complexity of this issue, in the following we present two characteristic integer error examples and discuss the complications introduced by the use of undefined operations.

2.2 Integer Error Examples

While the task of automatically detecting undefined arithmetic operations is relatively easy, the true difficulty lies in identifying the developer's intention behind the use of constructs that violate the language standard.

As an example, Listing 1 presents two C statements in which developers intentionally rely on undefined behavior, mainly to achieve persistent representation across different system architectures. Both are based on assumptions on the numerical representation used by the underlying system (two's complement). Line 1 shows a case of signedness casting in which the original value cannot be represented by the new type. In Line 2, a shift operation of `INT_WIDTH - 1` is undefined[2] but it conventionally returns the minimum value of the type, while the subtraction operation incurs a signed underflow which is also undefined. Although these cases are violations of the language standard, the desirable operation of an integer overflow checker would be to *not* report them, as they correspond to developer-intended behavior—otherwise, such cases are considered *false positives* [10].

In contrast, in the example of Listing 2, the `unsigned integer` variable (`alloc_size`) might overflow as a result of the multiplication operation at line 5. This behavior is well-defined by the standard, but the overflow may result in the allocation of a memory chunk of invalid (smaller) size, and consequently, to a heap overflow. An effective arithmetic error checker should be able to identify such potentially exploitable vulnerabilities as it is clear that the developer did not intend for this behavior.

3. APPROACH

The security community is still unsuccessful in completely eliminating the problem of integer errors even after years of effort [10, 17, 25, 23]. One of the main reasons is the difficulty in distinguishing *critical* errors, which may lead to reliability issues or security flaws, from uses of undefined constructs stemming from certain programming practices. The latter are regarded as errors by rigorous static checkers like IOC, as they strictly follow the language standard and report all violations found. In this work, we attempt to pinpoint critical, possibly exploitable arithmetic errors among all arithmetic violations, which include numerous less critical and often developer-intended uses of undefined behavior. Although many programming choices deviate from the language specification, IntFlow examines the conditions under which such constructs signify critical bugs, by focusing only on detecting errors that might break the functionality of the program or lead to a security flaw.

Before describing IntFlow's design, we first provide a concrete definition of what we consider critical arithmetic errors.

Definition 1. *An arithmetic error is potentially critical if it satisfies any of the following conditions:*

1. *At least one of the operands in an erroneous arithmetic operation originates from an* ***untrusted source****.*
2. *The result of an erroneous operation propagates to a* ***sensitive program location****.*

As capturing the intention of developers is a hard problem, IntFlow focuses on the detection of arithmetic errors that might constitute exploitable vulnerabilities or cause reliability issues. This is achieved not only by focusing on the identification of violations according to the language standard, which in itself is a tractable problem, but also by considering the information flows that affect the erroneous code. The rationale behind this definition is that *(1)* arithmetic errors influenced by external and potentially untrusted sources, such as sockets, files, and environment variables, may be exploited through carefully crafted inputs, and *(2)* arithmetic errors typically result in severe vulnerabilities when they affect sensitive library and system operations, such as memory allocation and string handling functions.

The two conditions of Definition 1 are reflected in IntFlow by two different modes of operation, *blacklisting* and *sensitive* (in addition to a third *whitelisting* mode), discussed in Section 4.3. Although the existence of either condition is an indication of a critical error, arithmetic violations for which both conditions hold are more severe, as they can potentially allow input from untrusted sources to misuse critical system functions—IntFlow's different modes of operation can be combined to detect such errors.

Figure 1 visualizes the definition with different types of information flows that may involve erroneous arithmetic operations. Critical errors are related to information flows that originate from untrusted inputs, or that eventually reach sensitive operations, such as system calls, through value propagation. In cases where the input of an arithmetic operation is untrusted (Definition 1.1) or a sensitive sink is reached (Definition 1.2), the error is flagged as critical. In contrast, arithmetic errors influenced only by benign inputs are considered less likely to be used in exploitation attempts.

The information flow based approach enables us to handle cases similar to the examples presented in Section 2.2. IntFlow can also silence error reports caused by statements similar to those presented in Listing 1, since it can trace that the

[2]According to the C99 and C11 standards. The C89 and C90 (ANSI C) standards define this behavior.

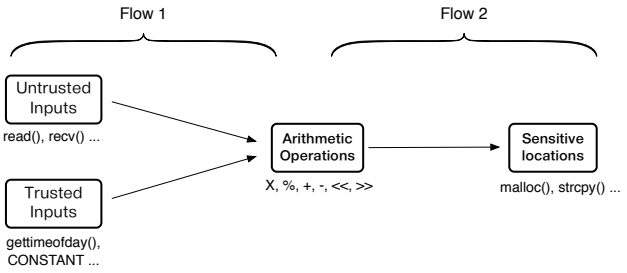


Figure 1: Information flows to and from the location of an arithmetic error.

origin of the value that leads to the undefined behavior is a constant initialization which is de facto developer-intended. The code snippet in Listing 2, contains two different types of flows. The first one connects `get_num_imgs()` (line 3) with the multiplication operation (line 5) while the second one connects the result of the multiplication with a memory allocation function (line 7), which is considered a sensitive program location. The former is a Type 1 flow because the value of the multiplication operand originates from an untrusted input, while the latter is a Type 2 flow since the result of the operation affects a sensitive function call. IntFlow would detect and report arithmetic errors caused by maliciously crafted inputs in both cases.

4. DESIGN AND IMPLEMENTATION

In this section we present the design and implementation of IntFlow, a tool that combines information flow tracking (IFT) [18] with a popular integer error checking tool [10] to improve the accuracy of arithmetic error detection. The main goal of IntFlow is to reduce the number of false positives produced by other static arithmetic error checkers. In this context, the term "false positive" refers to reporting developer-intended violations as critical errors. Although from the perspective of the language standard these correspond to erroneous code, the prevalence of such constructs makes reports of such issues a burden for security analysts, who are interested only in critical errors that may form exploitable vulnerabilities.

4.1 Main Components

IOC operates at the abstract syntax tree (AST) level produced by Clang [1], a C/C++ front-end of LLVM [14]. It instruments all arithmetic operations, as well as most of unary, casting, and type conversion operations. In contrast to previous tools that focus on a subset of integer errors (typically overflows and underflows), IOC provides protection against a broader range of integer errors. Even though it focuses mainly on errors with undefined behavior based on the language standards, it can also protect against errors that do not fall into this category, covering most of the integer error classes presented in Table 1.

IOC instruments *all* arithmetic operations that may lead to an erroneous result, and inserts checks accordingly. Essentially, for each integer operation inside a basic block, additional basic blocks that implement the error-checking logic are added and users are allowed to register callback functions for error handling. Similarly to other integer error detection systems, the fact that IOC instruments blindly all arithmetic operations is a major source of false positives, while IntFlow's active provisioning allows it to reduce false positives by eliminating checks for non-critical violations. IOC is a major component of our architecture, as it provides assurance that all potentially serious arithmetic errors can be checked. It is then up to the information flow analysis to identify and report only the critical ones.

For IntFlow's information tracking mechanism we employ `llvm-deps` [18], an LLVM compiler pass implementing static information flow tracking in a manner similar to classic data flow analysis [5]. It is designed as a context sensitive (inter-procedural) analysis tool that allows forward and backward slicing on source and sink pairs of our choice using the DSA [15] algorithm. DSA performs context-sensitive, unification-based points-to analysis that allows us to track data flows among variables referred by pointer aliases. It is important to note that the analysis scope of `llvm-deps` is limited to a single object file, as it is implemented as a compile-time optimization pass and not as a link-time optimization pass. Finally, due to the use of `llvm-deps`, IntFlow only examines explicit flows during its IFT analysis and ignores possible implicit flows.

4.2 Putting It All Together

Figure 2 illustrates the overall architecture of IntFlow: IOC adds checks to the integer operations that are exposed by Clang in the AST and then `llvm-deps` performs static IFT analysis on the LLVM intermediate representation (IR). To reduce unnecessary checks that may lead to false positives, IntFlow uses `llvm-deps` to examine only certain flows of interest. As discussed in the previous section, IntFlow examines only flows stemming from untrusted sources, or ending to sensitive sinks. Initially, IntFlow performs *forward* slicing: starting from a particular source used in a potentially erroneous arithmetic operation, it examines whether the result of the operation flows into sinks of interest. Once such a source is found, IntFlow performs *backward* slicing, to verify that the sink is actually affected by it. Since the flow tracking mechanism does not offer full code coverage, we employ this two-step process to gain confidence on the accuracy of the flow and verify its validity. Once the source is reached when using backward slicing starting from the sink, the flow is considered established.

After compiling and linking the IR, the resulting binary is exercised to identify critical errors, since the error checking mechanism triggers dynamically. Developers should execute the augmented binary with a broad range of inputs to exercise as many execution paths as possible, and identify whether they cause critical errors that can potentially lead to exploitable vulnerabilities.

4.3 Modes of Operation

As discussed in Section 3, IntFlow uses two different types of information flow to pinpoint errors. The first associates untrusted inputs with integer operations while the second associates the result of integer operations with its use in sensitive system functions. Once IOC inserts checks in all arithmetic operations that may lead to an error, IntFlow eliminates unnecessary checks by operating in one of the following modes:

- In *blacklisting* mode, IntFlow only maintains checks for operations whose operands originate from untrusted sources and removes all other checks.

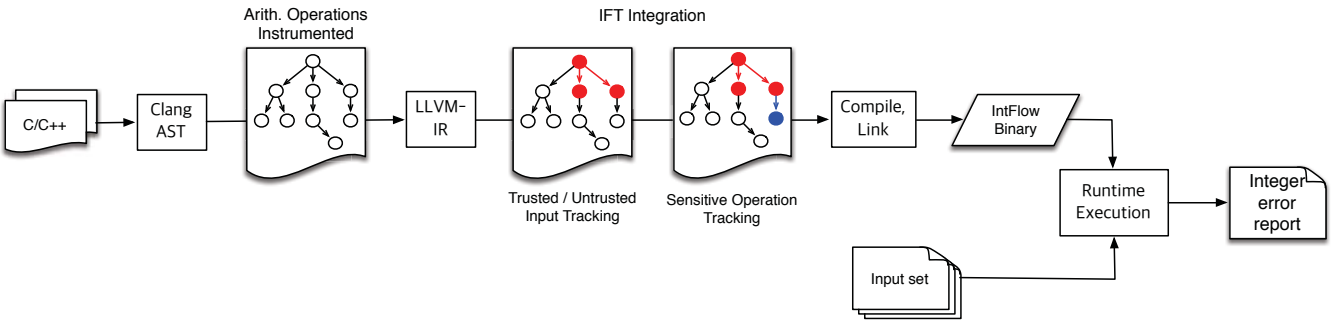


Figure 2: Overall architecture of IntFlow.

- In *sensitive* mode, IntFlow only maintains checks for operations whose results may propagate to sensitive sinks.

- In *whitelisting* mode, all checks for operations whose arguments come from trusted sources are removed.

4.3.1 Trusted and Untrusted Inputs

For each operation that may result in an arithmetic error, IntFlow's IFT analysis determines the origin of the involved operands. IntFlow then classifies the origin as either trusted or untrusted, and handles it accordingly, using one of the following two modes of operation.

Blacklisting: Input sources that can be affected by external sources are considered untrusted, since carefully crafted inputs may lead to successful exploitation. If any of the operands has a value affected by such a source, IntFlow retains the error checking instrumentation. System and library calls that read from untrusted sources, such as `read()` and `recv()`, are examples of this type of sources.

Whitelisting: Erroneous arithmetic operations for which all operands originate from trusted sources are unlikely to be exploitable. Thus, for those cases, IntFlow safely removes the error checks inserted by IOC at the instrumentation phase. Before an operation is verified as safe, IntFlow needs to examine the origin of *all* data flowing to that operation. Values derived from constant assignments or from safe system calls and library functions, e.g., `gettimeofday()` or `uname()`, are typical examples of sources that can be trusted, and thus white-listed.

Following either of the above two approaches, IntFlow selects the unsafe integer operations that will be instrumented with protection checks. These modes of operation can be complemented by IntFlow's third mode, which refines the analysis results for the surviving checks.

4.3.2 Sensitive Operations

In this mode, IntFlow reports flows that originate from integer error locations and propagate to sensitive sinks, such as memory-related functions and system calls. Moreover, in contrast to the previous modes, whenever an integer error occurs, the error is not reported at the time of its occurrence, but only once it propagates as input into one of the sensitive sinks. This is very effective in suppressing false positives, since errors that do not flow to a sensitive operation are not generally exploitable.

To report errors at sensitive sink locations, IntFlow performs the following operations:

- Initially, the tool identifies all integer operations whose results may propagate into a sensitive sink at runtime. Any checks that do not lead to sensitive sinks are not exploitable and thus are eliminated. A global array is created for each sensitive sink, holding one entry per arithmetic operation affecting it.

- Whenever an integer operation generates an erroneous result, its respective entries in the affected global arrays are set to `true`. If the operation is reached again but without generating an erroneous result, before reaching a sensitive location, the respective entry is set to `false`, denoting that the result of the sensitive operation will not be affected by this operation.

- If the execution reaches a sensitive function, the respective global array is examined. Execution is interrupted if one or more entries are set to `true`, as an erroneous value from any previous integer operation may affect its outcome.

Essentially, IntFlow keeps track of all the locations in the code that may introduce errors affecting a sensitive sink at compilation time. Once a sensitive sink is reached during runtime, IntFlow examines whether *any* of the error locations associated with the sink triggered an error in the current execution flow and in that case terminates the program. Although it is better to combine the two modes to establish end-to-end monitoring and detection of suspicious flows, each mode can also be used independently: the first mode to generally reduce the number of false positives, and the second mode to detect exploitable vulnerabilities.

4.4 Implementation

IntFlow is implemented as an LLVM [14] pass written in ~3,000 lines of C++ code. Briefly, it glues together its two main components (IOC and `llvm-deps`) and supports fine-tuning of its core engine through custom configuration files.

IntFlow can be invoked by simply passing the appropriate flags to the compiler, without any further action needed from the side of the developer. Although IOC has been integrated into the LLVM main branch since version 3.3, for the current prototype of IntFlow we used an older branch of IOC that supports a broader set of error classes than the latest one. IntFlow's pass is placed at the earliest stage of the LLVM pass dependency tree to prevent subsequent optimization passes from optimizing away any critical integer operations. During compilation, arithmetic error checks are inserted by IOC, and then selectively filtered by IntFlow. Subsequent compiler optimizations remove the filtered IOC basic blocks.

IntFlow offers developers the option to explicitly specify arithmetic operations or sources that need to be whitelisted or blacklisted. In addition, it can be configured to exclude any specific file from its analysis or ignore specific lines of code. Developers can also specify the mode of operation that IntFlow will use, as well as override or extend the default set of sources and sinks that will be considered during information flow analysis. Finally, they can specify particular callback actions that will be triggered upon the discovery of an error, such as activating runtime logging or exiting with a suitable return value. These features offer great flexibility to developers, enabling them to fine-tune the granularity of the generated reports, and adjust the built-in options of IntFlow to the exact characteristics of their source code.

5. EVALUATION

In this section, we present the results of our experimental evaluation using our prototype implementation of IntFlow. To assess the effectiveness and performance of IntFlow, we look into the following aspects:

- What is the accuracy of IntFlow in detecting and preventing critical arithmetic errors?
- How effective is IntFlow in reducing false positives? That is, how good is it in omitting developer-intended violations from the reported results?
- When used as a protection mechanism, what is the runtime overhead of IntFlow compared to native execution?

Our first set of experiments aims to evaluate the tool's ability to identify and mitigate critical errors. For this purpose, we use two datasets consisting of artificial and real-world vulnerabilities. Artificial vulnerabilities were inserted to a set of real-world applications, corresponding to various types of MITRE's Common Weakness Enumeration (CWE) [4]. This dataset provides a broad test suite that contains instances of many different types of arithmetic errors, which enables us to evaluate IntFlow in a well-controlled environment, knowing exactly how many bugs have been inserted, as well as the nature of each bug. Likewise, our real-world vulnerability dataset consists of applications such as image and document processing tools, instant messaging clients, and web browsers, with known CVEs, allowing us to get some insight on how well IntFlow performs against real-world, exploitable bugs.

In our second round of experiments, we evaluate the effectiveness of IntFlow's information flow tracking analysis in reducing false positives, by running IntFlow on the SPEC CPU2000 benchmark suite and comparing its reported errors with those of IOC. IOC instruments all arithmetic operations, providing the finest possible granularity for checks. Thus, by comparing the reports produced by IntFlow and IOC, we obtain a base case for how many non-critical errors are correctly ignored by the IFT engine.

Finally, to obtain an estimate of the tool's runtime overhead, we run IntFlow over a diverse set of applications of varying complexity, and establish a set of performance bounds for different types of binaries. All experiments were performed on a system with the following characteristics: 2× Intel(R) Xeon(R) X5550 CPU @ 2.67GHz, 2GB RAM, i386 Linux.

Applications	CWEs
Cherokee 1.2.101	CWE-190
Grep 2.14	CWE-191
Nginx 1.2.3	CWE-194
Tcpdump 4.3.0	CWE-195
W3C 5.4.0	CWE-196
Wget 1.14	CWE-197
Zshell 5.0.0	CWE-369
	CWE-682
	CWE-839

Table 2: Summary of the applications and CWEs used in the artificial vulnerabilities evaluation.

5.1 Accuracy

In all experiments for evaluating accuracy, we configured IntFlow to operate in *whitelisting* mode, since this mode produces the greatest number of false positives, as it preserves most of the IOC checks among the three modes. Thus, whitelisting provides us with an estimation of the worst-case performance of IntFlow, since the other two modes perform more fine-tuned instrumentation.

5.1.1 Evaluation Using Artificial Vulnerabilities

To evaluate the effectiveness of IntFlow in detecting critical errors of different types, we used seven popular open-source applications with planted vulnerabilities from nine distinct CWE categories.[3] Table 2 provides a summary of the applications used and the respective CWEs.

Each application is replicated to create a set of test-case binaries. In every test-case binary—essentially an instance of the real-world application—a vulnerability is planted and then the application is compiled with IntFlow. Subsequently, each test-case binary is executed over a set of benign and malicious inputs (inputs that exploit the vulnerability and result in abnormal behavior). A correct execution is observed when the binary executes normally on benign inputs or terminates before it can be exploited on malicious inputs.

Overall, IntFlow was able to correctly identify 79.30% (429 out of 541) of the planted artificial vulnerabilities. The 20.7% missed are due to the accuracy limitations of the IFT mechanism, which impacts the ability of IntFlow to correctly identify flows, and also due to vulnerabilities triggered by implicit information flows (i.e., non-explicit data flows realized by alternate control paths), which our IFT implementation (`llvm-deps`) is not designed to capture. We discuss ways in which accuracy can be further improved in Section 7.

5.1.2 Mitigation of Real-world Vulnerabilities

In our next experiment, we examined the effectiveness of IntFlow in detecting and reporting real-world vulnerabilities. For this purpose, we used four widely-used applications and analyzed whether IntFlow detects known integer-related CVEs included in these programs. Table 3 summarizes our evaluation results. IntFlow successfully detected *all* the exploitable vulnerabilities under examination. From this small-scale experiment, we gain confidence that IntFlow's

[3]The modified applications for this experiment were provided by MITRE for testing the detection of integer error vulnerabilities, as part of the evaluation of a research prototype [6].

Program	CVE Number	Type	Detected?
Dillo	CVE-2009-3481	Integer Overflow	Yes
GIMP	CVE-2012-3481	Integer Overflow	Yes
Swftools	CVE-2010-1516	Integer Overflow	Yes
Pidgin	CVE-2013-6489	Signedness Error	Yes

Table 3: CVEs examined by IntFlow.

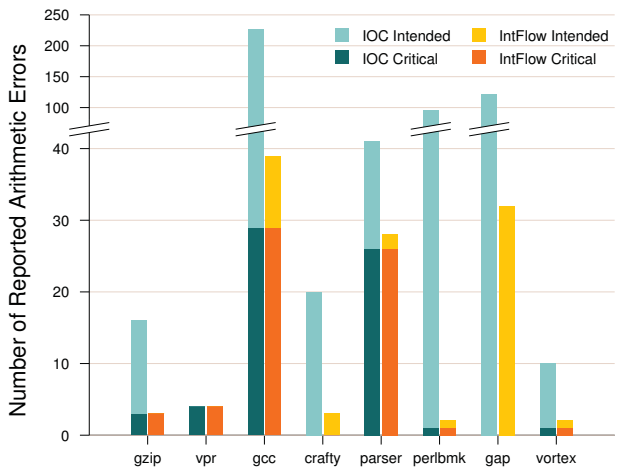


Figure 3: Number of critical and developer-intended arithmetic errors reported by IOC and IntFlow for the SPEC CPU2000 benchmarks. IntFlow identifies the same number of critical errors (dark sub-bars), while it reduces significantly the number of reported developer-intended violations.

characteristics are maintained when the tool is applied to real world programs, and therefore it is suitable as a detection tool for real-world applications.

5.1.3 *False Positives Reduction*

Reducing the number of false positives is a major goal of IntFlow, and this section focuses on quantifying how effective this reduction is. For our first measurement we used SPEC CPU2000, a suite that contains C and C++ programs representative of real-world applications. Since IOC is a core component of IntFlow, we chose to examine the same subset of benchmarks of CPU2000 that was used for the evaluation of IOC to measure the improvements of IntFlow's IFT in comparison to previously reported results [10]. We ran the SPEC benchmarks using the "test" data sets for both IOC and IntFlow, so that we could manually analyze all the reports produced by IOC and classify them as true or false positives, Once all reports were categorized based on Definition 1, we examined the respective results of IntFlow. We report our findings in Figure 3.

IntFlow was able to correctly identify all the critical errors (64 out of 64) triggered during execution, and reduced the reports of developer-intended violations by ∼89%.

5.1.4 *Real-world Applications*

In Section 5.1.2 we demonstrated how IntFlow effectively detected known CVEs for a set of real-world applications. Here, we examine the reduction in false positives achieved when using IntFlow's core engine instead of static instrumentation with IOC alone. To collect error reports, we ran

	Overall	Dillo	Gimp	Pidgin	SWFTools
IOC	330	31	231	0	68
IntFlow	82	26	13	0	43

Table 4: Number of False Positives reported by IOC and IntFlow for the real-world programs of Section 5.1.2.

each application with benign inputs as follows: for Gimp, we scaled the ACSAC logo and exported it as GIF; for SWFTools, we used the `pdf2swf` utility with a popular e-book as input; for Dillo, we visited the ACSAC webpage and downloaded the list of notable items published in 2013; and for Pidgin, we performed various common tasks, such as registering a new account and logging in and out. Table 4 shows the reports generated by IOC and IntFlow, respectively.

Overall, IntFlow was able to suppress 75% of the errors reported by IOC during the execution of the applications on benign inputs. Although this evaluation does not provide full coverage on the number of generated reports (for instance, we did not observe any false positives for pidgin with the tests we performed), it allows us to obtain an estimate of how well IntFlow performs in real world scenarios.

As the last part of our false positive reduction, we exercised vanilla versions of each application used in Section 5.1.1 over sets of benign inputs and examined the output. Since those inputs produce the expected output, we assume that all reported violations are developer-intended either explicitly or implicitly. With this in mind, we compared the error checks of IntFlow with those of IOC to quantify the ability of IntFlow in removing unnecessary checks. Overall, IntFlow eliminated 90% of the false checks (583 out of 647) when tested with the default set of safe inputs. This reduction was achieved due to the successful identification of constant assignments and the whitelisting of secure system calls, as discussed in Section 3.

It should be noted that the effectiveness in the reduction of false positives is highly dependent on the nature of each application, as well as on the level of the execution's source coverage. That is, the more integer operations occur throughout the execution, the greater the expected number of false positives. For instance, Gimp's functionality is tightly bound to performing arithmetic operations for a number of image processing actions, and thus IOC reports many errors, most of which are developer-intended, while Dillo does not share the same characteristics and as a result exhibits a smaller reduction in false positives.

5.2 Runtime Overhead

Although IntFlow was not designed as a runtime detection tool but rather as an offline integer error detection mechanism, one may wonder whether it could be customized to offer runtime detection capabilities. In this section, we seek to examine the performance of IntFlow for various applications, when running them with all the automatically inserted arithmetic error checks. For this purpose, we perform a set of timing measurements on the applications used in Section 5.1.1. For each run, we measured the time that was required to complete a series of tasks for each of IntFlow's modes of operation, and then normalized the running time with respect to the runtime of the native binary. Reported results are mean values over ten repetitions of each experiment, while the reported confidence intervals correspond to

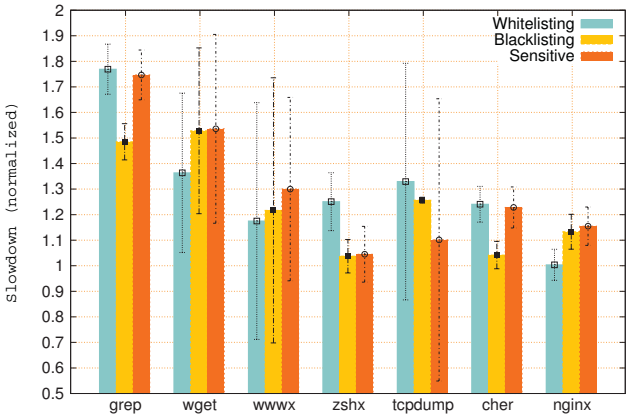


Figure 4: Runtime overhead for the applications of Section 5.1.1 (normalized over native execution).

95%. We ran all binaries natively, and measured user time with the `time` utility.

For `grep`, we search for strings that match a complex regular expression in a 1.2GB file. For `wget`, we download a 1GB file over a 1Gbit/s link from a remote server. For `zshx`, we execute a series of shell commands, and for `tcpdump` we examine packets from a 5.8GB pcap file. For the web servers, Cherokee was configured for IPv4 only, while for Nginx all configurations options were left to their default setting. We measured performance using Apache's `ab` benchmarking utility and static HTML files.

Figure 4 shows the results of our evaluation. IntFlow incurs high overhead in applications that involve a large number of integer operations, such as `grep`.[4] We also notice high performance deviation for `wget`, `wwwx`, and `tcpdump`, as they are I/O bound. Although in such applications the overhead is rather prohibitive, and cancels out the benefits of using a static mechanism, in other cases, such as for the `cher` and `nginx` servers, the overhead is within an acceptable 20%. Thus, it could be the case that IntFlow might be used as a runtime defense for certain types of applications, i.e., I/O-bound. As each of IntFlow's modes of operation targets different flows and can be fine-tuned by developers, customization can result in different overheads, as different flows dominate the execution of different applications. This is the reason we observe different slowdowns per mode: depending on whether the dominating flows involve sensitive calls (e.g., web servers), the sensitive mode will be slower or faster than the other two modes, and so on.

6. RELATED WORK

During the past years, as the protection mechanisms against buffer overflows became more mature, great focus was placed upon efficiently dealing with integer overflows. This section summarizes the main characteristics of the several approaches that have been followed so far for addressing integer overflows and outlines the connection between the current work and existing research on the field.

[4]Based on our experience with the SPEC CPU2000 benchmarks, the overhead on benchmarks with very frequent integer operations, such as `gzip`, is in the range of ∼x10, prohibiting IntFlow from being used as a generic runtime detection mechanism for such applications.

6.1 Static Analysis

Static analysis tools provide good coverage but generally suffer from a high rate of false positives. IntPatch [25] is built on top of LLVM [14] and detects vulnerabilities utilizing the type inference of LLVM IR. Similarly to our tool, IntPatch uses forward & backward analysis to classify sources and sinks as sensitive or benign. Each sensitive variable is located through slicing. If a variable involved in an arithmetic operation has an untrusted source and the respective sink may overflow, IntPatch will insert a check statement after that vulnerable arithmetic operation. If an overflow result is used for sensitive actions such as memory allocations, IntPatch considers it a real vulnerability. Contrary to the current work though, IntPatch does not deal with all types of integer overflows and also does not address programmer-inserted sanitization routines.

KINT [23] is a static tool that generates constraints representing the conditions under which an integer overflow may occur. It operates on LLVM IR and defines untrusted sources and sensitive sinks via user annotations. KINT avoids path explosion by performing constraint solving at the function level and by statically feeding the generated constraints into a solver. After this stage, a single path constraint for all integer operations is generated. Unfortunately, despite the optimization induced by the aforesaid technique, the tool's false positives remain high and there is a need for flagging false positives with manual annotations in order to suppress them. Moreover, contrary to this work, KINT attempts to denote all integer errors in a program and does not make a clean distinction between classic errors and errors that constitute vulnerabilities.

SIFT [16] uses static analysis to generate input filters against integer overflows. If an input passes through such filter, it is guaranteed not to generate an overflow. Initially, the tool creates a set of critical expressions from each memory allocation and block copy site. These expressions contain information on the size of blocks being copied or allocated, and are propagated backwards against the control flow, generating a symbolic condition that captures all the points involved with the evaluation of each expression. The free variables in the generated symbolic conditions represent the values of the input fields and are compared against the tool's input filters. A significant difference of this paper in comparison to SIFT is that the latter nullifies overflow errors but does not detect them nor examines whether they could be exploitable.

IntScope [21] decompiles binary programs into IR and then checks lazily for harmful integer overflow points. To deal with false positives, IntScope relies on a dynamic vulnerability test case generation tool to generate test cases which are likely to cause integer overflows. If no test case generates such error, the respective code fragment is flagged appropriately. This approach varies significantly from the one used in our tool as it relies on the produced test cases to reveal a true positive: if a test case does not generate an overflow, that does not guarantee that no overflow occurs. In addition, IntScope regards all errors as generic, without focusing particularly on errors leading to vulnerabilities.

Finally, RICH [7] is a compiler extension which enables programs to monitor their execution and detect potential attacks exploiting integer vulnerabilities. Although RICH is very lightweight, it does not handle cases of pointer aliasing and produces false positives in cases where developers inten-

tionally abuse the undefined behavior of C/C++ standards, both of which are basic components of IntFlow's design.

6.2 Dynamic & Symbolic Execution

Dynamic Execution tools use runtime checks to prevent unwanted program behavior. IOC [10] uses dynamic analysis to detect the occurrence of overflows in C/C++ programs. The tool performs a compiler-time transformation to add inline numerical error checks and then relies on a runtime handler to prevent any unwanted behavior. The instrumentation transformations operate on the Abstract Syntax Tree (AST) in the Clang front-end, after the parsing, type-checking and implicit type conversion stages. IOC checks for overflows both in shifting and arithmetic operations and makes a clear distinction between well-defined and undefined behaviors, that is, it does not consider all overflows malicious. Our tool adopts this perspective and complements IOC in the sense that it addresses the issue of high false positives by integrating static instruction flow tracking to the performed analysis.

Symbolic Execution tools provide low false positives but can't easily achieve full coverage. They usually use dynamic test generation [11] to detect violations. SmartFuzz [17] generates a set of constraints defining unwanted behavior and determines whether some input could trigger an unwanted execution path. This tool does not require source code and makes use of the Valgrind framework for its symbolic execution and scoring. Coverage and bug-seeking queries are explored in a generational search, whilst queries from the symbolic traces are solved generating a set of new test cases. Thus, a single symbolic execution feeds the constraint solver with many queries, which themselves generate new test cases etc. KLEE [8] uses symbolic execution to automatically generate tests that achieve high coverage for large-scale programs but it is not focused on integer errors, thus it does not achieve as good results against integer overflows as other tools that targeted towards integer operations.

7. DISCUSSION

7.1 Static information flow tracking

A core component of IntFlow is `llvm-deps` [18], which, as an implementation of static information flow tracking, is expected to provide good source code coverage with low runtime overhead. However, we should note that `llvm-deps` suffers from inherent inaccuracy issues, largely due to the limitations of its points-to analysis [15] and due to its data flow analysis mechanism. These limitations are amplified when one wishes to extend the scope of the technique by performing inter-procedural analysis. Fortunately, as our experience revealed, sources and sinks typically reside within a single function. This can be viewed as an instance of the classic trade-off between accuracy and performance: for cases where accuracy has the maximum priority, we may choose to incorporate dynamic IFT [13] and attempt to reduce any increased runtime overhead using techniques that combine static and dynamic analysis [12, 9].

7.2 IntFlow for Runtime Detection

While the primary use case of IntFlow is to help users analyze existing code during the development phase by reducing the amount of false positives reported by previous tools, another use case is to deploy it as a runtime defense against zero-day vulnerabilities. For this purpose, the two main issues that must be addressed are i) the increased runtime overhead due to the inserted checks, and ii) any remaining false positives after IntFlow's analysis.

The main source of runtime overhead, as shown in Section 5.2, can be attributed to IOC's checks, as it replaces each arithmetic operation with at least three basic blocks to perform the checking operation. Given the significance of the problem, there have been many previous proposals for implementing fast and efficient checking operations [22], which IntFlow could adopt to improve performance.

As shown in Section 5, IntFlow was able to identify a large portion of the developer-intended violations in the programs under examination, but still missed some cases. In order to provide as broad coverage of false positives as possible, IntFlow supports manual labeling of false positives. Developers can dedicate a separate off-line phase to apply IntFlow using a trusted input set over their application, and pinpoint the locations in which IntFlow falsely flags a benign operation as malicious. A more suitable solution for this use case would be the incorporation of dynamic IFT, which though would impose high runtime overhead, as already discussed.

7.3 Quality of the Produced Reports

Another advantage of IntFlow's design is the fact that its three different modes of operation offer an estimation of how critical a particular bug is. Errors reported by the sensitive mode have the highest risk, as they involve sensitive operations and are more likely to be exploitable. Likewise, in black-listing mode, IntFlow examines flows originating from untrusted locations and thus the produced reports are of moderate priority. Finally, the whitelisting mode is likely to generate the largest amount of warnings. Thus, if developers wish to examine as few locations as possible, e.g., due to limited available time for performing code auditing, they can first examine the reports generated by IntFlow in the sensitive mode, and if time permits then use the blacklisting mode, and so on.

Throughout our evaluation, we noticed that many of the reports generated by IntFlow follow a particular pattern, mainly due to code reuse from the side of developers. We believe that using simple pattern matching and lexical analysis of the source code, in combination with the reports of IntFlow, could further increase the accuracy with which IntFlow classifies errors as malicious or not—the more the occurrences of a particular error, the more likely for this error to be developer-intended. We will explore this approach as part of our future work.

8. CONCLUSION

We have presented IntFlow, a tool that identifies a broad range of arithmetic errors and differentiates between critical errors and developer-intended constructs that rely on undefined behavior, which do not constitute potential vulnerabilities. IntFlow uses static information flow tracking to associate flows of interest with erroneous statements, and greatly reduces false positives without removing checks that would prevent the detection of critical errors. The results of our evaluation demonstrate the effectiveness of IntFlow in distinguishing between the two types of errors, allowing developers and security analysts to detect and fix critical errors in an efficient manner, without the need to sift through numerous non-critical developer-intended violations. The

significant reduction in false positives that IntFlow achieves over IOC, which has been integrated into Clang since version 3.3, demonstrates the need for effective and accurate automated arithmetic error detection.

Acknowledgments

This work was supported by DARPA and the US Air Force through contracts DARPA-FA8750-10-2-0253 and AFRL-FA8650-10-C-7024, respectively, with additional support from Intel Corp. Any opinions, findings, conclusions, or recommendations expressed herein are those of the authors, and do not necessarily reflect those of the US Government, DARPA, the Air Force, or Intel.

9. REFERENCES

[1] Clang C language family frontend for LLVM. `http://clang.llvm.org/`.

[2] CVE - CVE-2006-3824. `http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2006-3824`.

[3] CWE - 2011 CWE/SANS top 25 most dangerous software errors. `http://cwe.mitre.org/top25/`.

[4] CWE - Common Weakness Enumeration. `http://cwe.mitre.org/`.

[5] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Longman Publishing Co., Inc., 1986.

[6] A. Benameur, N. S. Evans, and M. C. Elder. Minestrone: Testing the soup. In *Proceedings of the 6th Workshop on Cyber Security Experimentation and Test (CSET)*, 2013.

[7] D. Brumley, T. Chiueh, R. Johnson, H. Lin, and D. Song. RICH: Automatically Protecting Against Integer-Based Vulnerabilities. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2007.

[8] C. Cadar, D. Dunbar, and D. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation (OSDI)*, 2008.

[9] W. Chang, B. Streiff, and C. Lin. Efficient and extensible security enforcement using dynamic data flow analysis. *Proceedings of the 15th ACM conference on Computer and Communications Security (CCS)*, 2008.

[10] W. Dietz, P. Li, J. Regehr, and V. Adve. Understanding integer overflow in C/C++. In *Proceedings of the 34th International Conference on Software Engineering (ICSE)*, 2012.

[11] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed automated random testing. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2005.

[12] K. Jee, G. Portokalidis, V. P. Kemerlis, S. Ghosh, D. I. August, and A. D. Keromytis. A general approach for efficiently accelerating software-based dynamic data flow tracking on commodity hardware. In *Proceedings of the 19th Network and Distributed System Security Symposium (NDSS)*, 2012.

[13] V. P. Kemerlis, G. Portokalidis, K. Jee, and A. D. Keromytis. libdft: practical dynamic data flow tracking for commodity systems. In *Proceedings of the 8th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE)*, 2012.

[14] C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the international symposium on Code generation and optimization (CGO)*, 2004.

[15] C. Lattner, A. Lenharth, and V. Adve. Making context-sensitive points-to analysis with heap cloning practical for the real world. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2007.

[16] F. Long, S. Sidiroglou-Douskos, D. Kim, and M. Rinard. Sound input filter generation for integer overflow errors. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 2014.

[17] D. Molnar, X. C. Li, and D. A. Wagner. Dynamic test generation to find integer bugs in x86 binary linux programs. In *Proceedings of the 18th USENIX Security Symposium*, 2009.

[18] S. Moore. thinkmoore/llvm-deps. `https://github.com/thinkmoore/llvm-deps`. (Visited on 06/07/2014).

[19] E. Revfy. Inside the size overflow plugin. `http://forums.grsecurity.net/viewtopic.php?f=7&t=3043`.

[20] T. Wang, C. Song, and W. Lee. Diagnosis and emergency patch generation for integer overflow exploits. In *Proceedings of the 11th Conference on Detection of Intrusions and Malware and Vulnerability Assessment (DIMVA)*, 2014.

[21] T. Wang, T. Wei, Z. Lin, and W. Zou. Intscope: Automatically detecting integer overflow vulnerability in x86 binary using symbolic execution. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2009.

[22] X. Wang. Fast integer overflow detection. `http://kqueue.org/blog/2012/03/16/fast-integer-overflow-detection/`.

[23] X. Wang, H. Chen, Z. Jia, N. Zeldovich, and M. F. Kaashoek. Improving integer security for systems with KINT. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation (OSDI)*, 2012.

[24] X. Wang, N. Zeldovich, M. F. Kaashoek, and A. Solar-Lezama. Towards optimization-safe systems. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*, 2013.

[25] C. Zhang, T. Wang, T. Wei, Y. Chen, and W. Zou. Intpatch: Automatically fix integer-overflow-to-buffer-overflow vulnerability at compile-time. In *Proceedings of the 15th European Symposium on Research in Computer Security (ESORICS)*, 2010.

CPS: Beyond Usability: Applying Value Sensitive Design Based Methods to Investigate Domain Characteristics for Security for Implantable Cardiac Devices

Tamara Denning
University of Utah
tdenning@cs.utah.edu

Batya Friedman
University of Washington
batya@uw.edu

Brian Gill
Seattle Pacific University
bgill@spu.edu

Daniel B. Kramer
Beth Israel Deaconess Medical Center
dkramer@bidmc.harvard.edu

Matthew R. Reynolds
Harvard Clinical Research Institute
matthew.reynolds@hcri.harvard.edu

Tadayoshi Kohno
University of Washington
yoshi@cs.washington.edu

ABSTRACT

Wireless implantable medical devices (IMDs) are cyber-physical systems that deliver life-saving treatments to cardiac patients with dangerous heart conditions. Current access control models for these systems are insufficient; more security is necessary. In response to this problem, the technical security community has investigated new directions for improving security on these resource-constrained devices. Defenses, however, must not only be technically secure; in order to be deployable, defenses must be designed to work within the needs and constraints of their relevant application spaces. Designing for an application space—particularly a specialized one—requires a deep understanding of the stakeholders, their values, and the contexts of technology usage. Grounding our work in value sensitive design (VSD), we collaborated as an interdisciplinary team to conduct three workshops with medical providers for the purpose of gathering their values and perspectives. The structure of our workshop builds on known workshop structures within the human-computer interaction (HCI) community, and the number of participants in our workshops (N=24) is compatible with current practices for inductive, exploratory studies. We present results on: what the participants find important with respect to providing care and performing their jobs; their reactions to potential security system concepts; and their views on what security system properties should be sought or avoided due to side effects within the context of their work practice. We synthesize these results, use the results to articulate design considerations for future technical security systems, and suggest directions for further research. Our research not only provides a contribution to security research for an important class of cyber-physical systems (IMDs); it also provides an example of leveraging techniques from other communities to better explore the landscape of security designs for technologies.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-3005-3/14/12···$15.00
http://dx.doi.org/10.1145/2664243.2664289

Keywords

Cyber-physical systems, human factors, implantable cardioverter-defibrillators, implantable cardiac devices, implantable medical devices, medical, methods, pacemaker, practical security, privacy, security, stakeholders, envisioning workshops, value sensitive design.

1. INTRODUCTION

This work focuses on security for an important class of cyber-physical systems: wireless implantable medical devices. In this work, we bring together the technical computer security research community and value sensitive design—a framework first developed within the human-computer interaction community—in order to gather the kind of domain-specific information that is necessary to design and deploy effective security.

On the Need for Value-Sensitive Investigations. Designing effective security systems that will be appreciated and embraced by users and other stakeholders requires a deep understanding of stakeholders, their values, and the contexts of technology use. This is especially true for systems like cyber-physical systems, which perform critical functions and can require non-standard security and access control solutions. In order for the security community to design effective security that can be deployed in the real world, it is critical that human beings and human-centered methods (e.g., [10], [27]) be foregrounded in the research process.

Prior research has explored users' internal mental models in order to help security researchers design systems that act in accordance with users' expectations; this kind of work also helps security researchers understand the reasons behind unexpected user behaviors (e.g., [13][38]). Other work examines how to design for and evaluate the performance of usable security and access control systems (e.g., [2][4][5][19][33]). While this body of research is valuable, the studies are often focused solely on users, and often give little consideration to other important stakeholders or the larger ecosystem in which a technology and its security system are likely to be deployed.

As part of our research agenda—and drawing on value sensitive design—we have identified four additional elements that we believe are essential to consider in the computer security research and design process:

- **Stakeholders (e.g., [6], [14], [24],).** Who are the broad set of people who have a direct or indirect stake in the system? Indirect stakeholders are not users, but may nonetheless be impacted significantly by the technology.
- **Stakeholder Goals and Values (e.g., [6], [7], [14], [24], [41]).** What are the diverse stakeholder groups trying to accomplish, and what do they value?
- **Implications for Values (e.g., [6], [12], [14], [24], [25]).** How might different security design choices potentially impact stakeholder values—positively or negatively? In what ways might direct and indirect stakeholder groups be differently impacted?
- **Communicating Technical Concepts in Comprehensible Ways (e.g., [6], [7], [26]).** How can security researchers and designers communicate about technical security concepts with stakeholders in ways that stakeholders can understand and appreciate?

Wireless Implantable Medical Devices. Many modern implantable medical devices, including pacemakers and defibrillators, are both computational and wireless. These cyber-physical systems both contain private information and can physically affect patients' bodies. The combination of these properties makes IMDs a critical class of devices to secure.

Our Work. In this study we draw upon value sensitive design (see Section 2.2) to further these agenda elements using the domain of implantable cardiac devices—a domain that has already received significant interest from the community (e.g., [15][37][30][18]).

We conducted security-oriented Envisioning Workshops (see Section 4.2) with a variety of stakeholders involved in the care of patients with implantable cardiac devices, including: nurses, emergency physicians, cardiologists, anesthesiologists, and device manufacturer representatives. We present results on: (1) what participants find important with respect to providing care and performing their jobs; (2) the metaphors participants use to describe implantable cardiac devices and security systems for these devices; (3) participants' evaluations of potential systems that represent different directions in technical security design; and (4) participants' views on what security system properties should be sought or avoided due to domain-relevant side effects. To be clear, the purpose of this research is not to gather participants' feedback on the security performance of these systems—after all, the participants are not security experts—but rather to *gather information about how different access control systems might impact participants' jobs and their ability to care for patients*. This information helps security researchers design solutions that avoid negative side effects and tailors solutions to the needs of the domain. While much of the gathered data might be considered common knowledge by those with experience in the area, we refine and utilize procedures with which to gather this information in a structured manner.

Contributions from this work are as follows:

- **Domain Information for Designing Effective Security.** We offer domain-specific findings for implantable cardiac devices—cyber-physical systems that need improved security. Security experts can utilize the data from this study to inform the design of security systems, with the goals of increasing system adoption, supporting correct usage of security systems, and avoiding negative system side effects.
- **Method.** We adapted the Envisioning Workshop method to the security domain: we foregrounded early-stage security systems in order to gather explicit feedback on potential security directions and to identify value tensions. We provide a case study for the computer security community that demonstrates how researchers can draw upon stakeholder expertise to understand the relevant properties of a new technology domain, including value tensions among stakeholder groups. While we have conducted this research in the context of implantable medical devices, the workshop technique could be used to explore other domains such as automobiles, augmented reality, or 3D printing.

2. RELATED WORK

2.1 Technical Work in IMD Security

Implantable cardiac devices store information such as the patient's name and records of irregular heart rhythms that occurred since the last checkup. Patients visit cardiology clinics periodically so that medical staff can download information about these episodes and adjust settings on the patient's device. Current-generation cardiac devices have the ability to communicate wirelessly with external equipment from distances up to 5 meters away. There are numerous reasons for making IMDs wireless. For example, wireless IMDs can be configured ("programmed") by doctors in the operating room from farther away, which avoids the need to bring programming equipment into the sterile operating area. Wireless technology also allows the IMD to send alerts to a home monitoring station—which can then send a report to the patient's physician for analysis—without causing interruption to the patient's activities. Unfortunately, incorporating a new wireless interface for these devices also increases the communication surface on which they can be attacked, and current security models do not provide access control on a per-device basis. Here we provide an overview of prior work on security for IMDs.

In 2008, it was demonstrated that an implantable cardiac device with centimeters-range wireless communications capabilities can be wirelessly compromised by a (nearby) unauthorized party [18]; in 2011, it was demonstrated that an implantable cardiac device with meters-range wireless capabilities can also be wirelessly compromised [15]. Other work has demonstrated vulnerabilities in wireless insulin pump systems [23][28].

Challenges for securing wireless IMDs and possible directions for improving future security have been outlined [17]. One key challenge is to balance security (*blocking inappropriate access*) while also providing some guarantee that safety can be ensured in an emergency (*facilitating appropriate situational access*). To illustrate this point, consider a security system in which the IMD only grants wireless access to individuals who know a password, such as the patient's regular cardiologist. While such a system does improve security and can prevent unauthorized access by random individuals, this system also directly and negatively impacts safety: emergency personnel will not be able to read or change settings on the device without first contacting the patient's cardiologist, who might be unreachable.

There have been numerous early-stage proposals to help improve security while prioritizing medical access. One proposed direction requires the patient to wear a wristband that protects the security of the IMD when worn, but that can be removed for emergency access [8][15][1]. Another direction requires body modifications, such as RFID implants or tattoos with visible or UV-visible ink [32]. Yet another direction requires the doctor to place something on or near the patient in order to activate longer-range wireless

capabilities (e.g., [3][30][37]), taking advantage of cryptographic distance-bounding, intra-body signaling, or physiologically-derived keys. Drawing from past work, an IMD could also potentially use automated techniques to detect emergency situations and decrease security requirements (e.g., not require a password) if the patient is in a state of medical emergency [16]. Another, more traditional approach might be to issue temporary or permanent access passwords via a centralized entity, such as a manufacturer-maintained database.

Although less related to our work on implantable cardiac devices, there has been significant work focused on security and privacy for personal medical sensors and networked medical devices (e.g., [29]); see [1] for a survey. Many of the efforts in this space also have potential applicability to implantable medical devices. For example, the Amulet system [34]—which requires the user to wear an external device—has many overlapping elements with other defenses for IMDs [8][15][1]. Additionally, any effort to improve key establishment for body-area networks (e.g., [3][37]), can be used to help improve the security of key establishment systems for IMDs.

Recently, a systematization of knowledge paper was published on the topic of computer security for implantable medical devices and body area networks (BANs) [31]. The paper provides a thorough examination of both attack and defense work dealing with these classes of technologies in the computer security community. The authors categorize defense directions as falling into four different trends: Biometric and Physiological Values (e.g., ECG or IPI), Out-of-Band (e.g., tattoos), Distance Bounding (e.g., intrabody signaling or cryptographic), and External Devices (e.g., fail-open wristband). Four additional categories—Wireless Attacks, Software/Malware, Anomaly Detection, and Emerging Threats—are used to classify other research trends in the area.

2.2 Value Sensitive Design

Computer security and access control systems are frequently discussed in the context of values such as security, privacy, and convenience. These systems, however, also affect and are affected by other important human values such as trust, physical welfare, autonomy, or human dignity. In this research we drew on established theory and methods from value sensitive design (e.g., [12][14][24][26]) to frame our study design and our data analyses. First developed in human-computer interaction,, value sensitive design has since been used in civil engineering, information management, human-robotic interaction, and ubiquitous computing. For example, one security-focused series of value sensitive design studies analyzed informed consent for cookies in web browser security [25], leading to recommendations for browser redesign to better support informed consent and a proof-of-concept redesign in the form of a plug-in "cookie watcher" for the Mozilla browser [12]. Another security-focused study investigated users' mental models for web browser security, suggesting that elements in the user interface (e.g., the open or closed padlock) were inadvertently leading some users to construct incorrect mental models for a secure connection [13]. More recently, value sensitive design methods have been applied in research about safety and security for mobile phone parenting technologies for teens [6], home technologies [9], and implantable cardiac devices from patients' perspectives ([7], see Section 3.1).

In this study, we drew explicitly on two value sensitive design methods: direct and indirect stakeholder analyses, and value dams and flows.

Direct and Indirect Stakeholder Analyses. In examining the ecosystem surrounding security for implantable cardiac devices, an important question is what professional roles should be represented among the study participants. Value sensitive design stresses the consideration of both: the direct stakeholders who will interact with the technology (e.g., cardiologists); and indirect stakeholders who—while they do not directly interact with the technology—can affect and be affected by the technology (e.g., venture capitalists who invest in implantable cardiac devices). Section 3.2 provides a brief explanation as to why taking this broader view is valuable.

Value Dams and Flows. Given a wide range of possible technical security, it is not always obvious how to choose which system to pursue. Value dams and flows is a technique for identifying reasonable, value-sensitive design options from among a set of possible designs or technical features (e.g., [6][7][24]). First, options that a threshold percentage of stakeholders strongly object to are removed from the list of viable solutions (value dams); then, from the remaining options, those that many stakeholders favor are selected as good candidates for solutions (value flows). In this research we use the value dams and flows method to help identify viable security designs for implantable cardiac devices.

3. STAKEHOLDERS, SECURITY AND THE MEDICAL ECOSYSTEM FOR IMDS

3.1 Patients: Prior Work

Previous work explored patient views and values regarding their implantable cardiac devices in order to inform the design of security systems for wireless IMDs [7]. The findings in that study—as well as the absence of the medical provider viewpoint—are part of the motivation for this study.

In the patient study, semi-structured interviews were conducted with 13 individuals with implanted cardiac devices. Key questions concerned the evaluation of eight mockups of early-stage IMD security systems. For each system, patients provided judgments as to whether they liked or disliked the system and whether or not they would choose to use it. Qualitatively, patients articulated a number of values that affected their attitudes toward the systems, including: security; safety; privacy; aesthetics; psychological welfare; convenience; cultural and historical associations; self-image and public persona; and autonomy and notification. Importantly, different patients identified different sets of cares and concerns that, in turn, led to different levels of satisfaction with the proposed security designs.

3.2 Medical Providers

A patient's medical care is an ongoing process that is affected by regulation, device manufacturers, federal testing, insurance companies, hospital equipment purchases, primary care staff, specialist nurses and doctors, emergency care staff, operating room staff, and others. While the patient study described above provides some insight into the values and priorities of patients who live with implantable cardiac devices embedded in their bodies, the question of how medical providers might interact with these technical computer security directions was not addressed. Yet, to be effective, security must work for and with all of the key stakeholders. In the case of implantable cardiac devices, this includes not only the patients, but also the medical providers who—in one way or another—ensure that the devices function properly and improve patient health.

The current study with medical providers builds on the patient study by investigating similar security system designs concepts. However, as described in the methods below, the participant pool and study format differ (as appropriate to these stakeholder groups).

4. METHODS

This study is part of an on-going, interdisciplinary collaboration among researchers in computer security, human-computer interaction, and cardiology. The researchers have no special relationships with any particular medical device provider or other conflicting agency.

4.1 Participants

In this study, we sought to investigate in detail the values, priorities, constraints, and themes that emerge in a complex domain. We followed an established approach to work in depth with a smaller number of participants (e.g., [22][36]) rather than seeking to answer specific questions using a larger number of participants. We conducted three workshops with medical providers in the United States: one in a city on the west coast (Group I) and two in a city on the east coast (Groups II and III). Participants were recruited through a snowball method. The research team first sent emails to previous contacts in the medical community requesting suggestions for potential participants and relevant mailing lists; the researchers then followed up on those suggestions with email letters of invitation to participate in the research. Recruitment efforts were initially extremely slow; this was partially because we needed to obtain permission from appropriate authorities (i.e., "gatekeepers") and partially because we needed domain insiders to explain the importance of—and cultivate enthusiasm for—study participation (i.e., "advocates").

We applied for and obtained approval from our institution's human subjects review board. In order to synchronize study protocols across the multiple institutions involved in this study, it was necessary to submit multiple modifications. Participants were compensated $200 for their time; while this amount may seem unusually high, it was deemed appropriate in the context of the particular participant pool (e.g., cardiologists).

A total of 24 medical providers (age: average=39, min=28, max=64) participated in the study. Table 1 breaks down participant gender by workshop. Participants had a broad spectrum of roles in the medical ecosystem: cardiologists and electrophysiologists (n=2), nurses and nurse practitioners in cardiology and electrophysiology (n=5), anesthesiologist (n=1), emergency physician (n=1), other physicians (n=2), physician assistant (n=1), medical residents (n=4), medical device manufacturer representative (n=1), biomedical informatics researcher (n=1), and venture capitalist (n=1).

4.2 Workshop Format

To elicit participant values, priorities, and constraints for the security of implantable cardiac devices, we sought a method that would provide opportunities for open-ended ideation about device security as well as focused reactions to potential early-stage security concepts. We drew inspiration from and adapted Kensing and Madsen's techniques for "generating visions" [20]—which integrates metaphorical design with a Future Workshop (particularly the critique phase)—and Yoo et al.'s Envisioning Workshop, which emphasizes surfacing value tensions among diverse stakeholders [40]. In both instances, we tailored the workshop structure to focus on security aspects of specific system designs. In addition, we sought both to collect individual reflections and to benefit from group discussion; thus, data collection included individual written materials as well as verbal group interactions. We describe the workshop protocol below.

Group	Male	Female	**Total**
I	4	6	**10**
II	4	3	**7**
III	5	2	**7**
TOTAL	**13**	**11**	**24**

Table 1. Number of participants by group and gender.

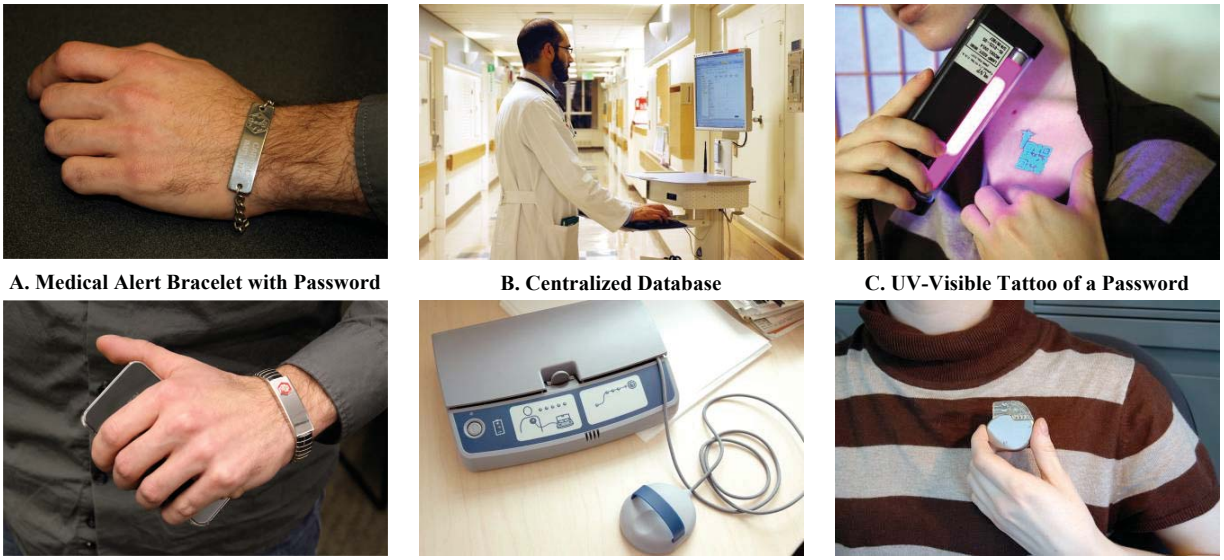
Each session lasted a total of two hours. Audio recordings were made of each session and then later transcribed for analysis.

Implantable Cardiac Device Overview and Initial Perspectives. To ensure that all participants had some shared vocabulary for implantable cardiac devices, a research team member provided a brief overview of implantable cardiac devices and clarified how terms would be used during the workshop. This overview did not include information on security for implantable cardiac devices. Following this overview and to tap into participants' perspectives prior to any influence from the workshop activities, participants were asked to complete a brief paper and pencil worksheet that elicited their initial views on security and access control for implantable cardiac devices. The worksheet contained the following questions: (1) *What properties about implantable cardiac devices or the ecosystem surrounding them do you value most?*; (2) *What things about implantable cardiac devices or the ecosystem surrounding them should not change?;* (3) *What things about implantable cardiac devices or the ecosystem surrounding them most need improvement?*; (4) *What is the most common problem related to implantable cardiac devices that you encounter in your line of work (e.g., lack of access to patient information, inability to access cardiac device, device malfunction)?*; and (5) *What is the problem with the most negative health impact (related to implantable cardiac devices) that you encounter in your line of work?*

Metaphor Generation. To help understand the broad backdrop of participants' perspectives as well as potential mental models, participants were invited as a group to share verbally: (1) metaphors for implantable cardiac devices; and (2) metaphors for security and access control for those devices. A research team member facilitated the contributions and recorded each metaphor in a few concise words on a whiteboard.

Critiques and Concerns. To understand how security and access control systems for implantable cardiac devices could go awry as well as to understand medical providers' hesitations and concerns about this type of technology, participants were invited as a group to share verbally their concerns and fears about security for implantable cardiac devices. Volunteer participants grouped the concerns into clusters based on similarity.

Evaluation of Security and Access Control System Concepts. To understand participants' views on what properties to advocate for and which to avoid in the development of security and access control solutions for implantable cardiac devices, a researcher introduced participants to six potential security and access control systems, one at a time. The researcher indicated that these were early, representative systems designed to elicit feedback. The system concepts are described in Section 4.3 below. Directly after each system presentation, participants completed a paper and pencil worksheet in which they recorded their responses to the following questions: *From your perspective as a professional who*



A. Medical Alert Bracelet with Password **B. Centralized Database** **C. UV-Visible Tattoo of a Password**

D. Fail-Open/Safety Wristband **E. Proximity-Based Equipment** **F. Criticality-Aware Fail-Open IMD**

Figure 1. Photos of the security system concepts presented to medical providers during the workshop.

deals with implantable cardiac devices, what do you like about this concept? What do you dislike about this concept? Why?

Once participants had been introduced to all six system concepts, participants completed a worksheet with the following questions: *(1) Would you say that you like any of the concepts, and if so, which ones?; (2) Would you say that you dislike any of the concepts, and if so, which ones?; (3) If you were to choose one or more of these concepts to recommend for use in the future, which concept or concepts would you choose? Why?; and (4) If you were to choose one or more of these concepts to recommend against use in the future, which concept or concepts would you choose? Why?* Participants had materials for each system concept on hand if they needed to refresh their memories.

Security System Property	System Concepts					
	A	B	C	D	E	F
Requires patient to wear something [8][15][1]	■			■		
Requires modification to the patient's body [32]			■			
Requires patient maintenance [8][15][1]				■		
Visible on the patient [8][15][1]	■		▒	■		
Access depends on centralized infrastructure		■				
Requires specialized provider equipment [3][30][37]			■		■	
Access requires proximity to the patient [3][30][37]	▒		■	▒	■	
Access has a manual override [8][15][1]				▒	■	
Security decisions are automated [16]						■

Table 2. Relevant properties of the security system concepts presented to medical providers (by system concept). Dark cells indicate a property represented by a system. Lighter cells indicate a property represented in some situations or by some interpretations.

Open-ended Discussion. Finally, to ensure that participants had ample opportunity to surface any major issues that might have been missed, participants engaged in an open-ended discussion around security and access control for implantable cardiac devices in which they could respond to and debate each other's ideas. To initiate the discussion, a research team member used the following prompt: *What are the challenges in this space?*

Due to space constraints, we do not present the results of the critiques section or the open-ended discussion. While this data contained some interesting information, by and large the content and issues overlapped with that from the other workshop sections.

4.3 Security System Concepts

As noted above, participants were asked to evaluate six security and access control system concepts (see Figure 1). A researcher presented the systems to participants using verbal explanations and slides. These systems are not complete or perfect from a security (access control: false positive), safety (access control: false negative), or usability standpoint. We presented these systems for feedback because: (a) they are representative of some of the security solutions that have been previously proposed by the security research community; (b) they represent a variety of relevant system properties; and (c) the discussion of a specific system can serve to ground an otherwise abstract discussion. Table 2 provides a summary of some of the systems' relevant properties. The six security concepts are as follows:

A. **Medical Alert Bracelet with Password.** A medical alert bracelet worn by the patient with a password engraved on the inside. The password is required to read information off of or to reprogram the implanted cardiac device via the programmer.

B. **Centralized Database.** A centralized database system that medical centers can access to obtain a temporary password that can then be used to access the implanted cardiac device.

C. **UV-Visible Tattoo of a Password.** The patient is tattooed with a scannable representation of a password (e.g., 2D barcode) using UV-visible ink that is only visible under a UV (black light) light source.

D. **Fail-Open/Safety Wristband.** The patient wears a battery-powered system that, when present, prevents unauthorized programmers from communicating with the implanted cardiac device. When the wristband is removed, the implanted device accepts communications from all programmers. Additionally, the wristband: sounds an alarm when the patient is approaching a strong magnet source (a potentially serious risk to cardiac device patients); and dials 911 if the patient is experiencing a cardiac emergency.

E. **Proximity-Based Equipment.** Equipment that medical professionals can hold in contact with the patient's body in order to communicate with the implantable cardiac device.

F. **Criticality-Aware Fail-Open IMD.** An implantable cardiac device that uses available information—such as the patient's cardiac rhythm, movements, and location—to determine whether or not the patient is likely experiencing a medical emergency. If the system detects a likely emergency, it changes the security settings to allow access from any programmer. The system does not notify anyone about the medical emergency and does not change cardiac therapies.

4.4 Coding and Reliability

Participants' written initial perspectives were coded systematically using the following process: one researcher developed a coding scheme using all of the data; once completed, that researcher used the finalized coding scheme to systematically recode the entire data set. A second coder—not affiliated with the research team or the study—was trained in the coding scheme using data from 4 participants, and then independently performed reliability coding of the data for the remaining 20 participants. This process resulted in an overall kappa of 0.745; Fleiss rates any value of kappa over 0.75 as excellent agreement and between 0.40 and 0.75 as intermediate to good agreement [11], while Landis and Koch rate a kappa of 0.81 to 1.00 as "almost perfect" and between 0.61 and 0.80 as "substantial" agreement [21].

The metaphor data set was smaller and, thus, more appropriately coded by consensus. We used the following process: (1) first, two researchers independently read through all of the data to generate an initial set of coding categories and assign responses to categories; (2) next, using consensus, researchers iteratively synthesized categories and arrived at agreement; and (3) both researchers made another independent pass through all of the data and any lingering disagreements were resolved.

Justifications in the security system concept evaluation data were identified from inspection of the qualitative data and are presented via participant quotes.

5. RESULTS

Given the relatively small number of participants in each workshop, there was no way to draw meaningful comparisons among the workshops' participant demographics (e.g., location, gender, age, professional role). We combined the data from all three workshops into one data set.

5.1 Initial Perspectives

Participants' written responses to the Initial Perspectives questions provide a relatively unbiased (that is, largely uninfluenced by our subsequent workshop activities) view into what participants consider important about implantable cardiac devices and their usage to treat patients. Since our primary interest was to understand broadly the pre-existing issues important to medical providers, we examined providers' responses to the set of five questions as a whole (rather than by individual question).

Thirteen categories of issues emerged from the analysis of participant responses as follows (in alphabetic order): (1) *Access & Sharing;* (2) *Compatibility;* (3) *Correct Usage;* (4) *Device Battery Life;* (5) *Device Compactness / Inertness;* (6) *Device Ecosystem;* (7) *Device Functionality;* (8) *Patient / Patient Health;* (9) *Programming;* (10) *Quality of Data;* (11) *Remote Monitoring;* (12) *Security & Privacy; and* (13) *Surgery & Healing.* Over three-quarters of the participants expressed issues related to Device Functionality (79%) and Patient/Patient Health (75%); more than half mentioned Surgery & Healing (58%). The next most represented categories were mentioned by roughly a quarter of the participants (ranging from 25-29%). That said, given the sample size and exploratory nature of this study, we believe it would be prudent to consider all 13 categories of issues when designing a security system for implantable cardiac devices.

This list of issues provides a window into the values and priorities of medical providers dealing with implantable cardiac devices. Security and human-computer interaction researchers may not have sufficient domain knowledge to make direct judgments as to how a system design might interact with these aspects of medical care; these categories, and other data like them, may serve as a meaningful starting point for dialog with domain experts.

5.2 Metaphors

Metaphors often underlay people's mental models of technological systems, which can affect the ways in which they interact with those systems. The metaphors supplied by participants provide some indication as to how they conceptualize implantable cardiac devices and security systems for those devices. In addition, using metaphor generation as an opening activity was intended to help break the ice; metaphor generation is rapid and appropriate for ideas that might otherwise be considered offbeat or silly.

As a group, participants generated a total of 81 metaphors: 42 for the implantable cardiac devices and 39 for security and access control for those devices. The following 11 categories—given in alphabetic order—emerged from clustering together similar metaphors: (1) *Agency;* (2) *Bio-medical;* (3) *Business;* (4) *Emotion;* (5) *Information;* (6) *Maintenance;* (7) *Personal Identity;* (8) *Privacy;* (9) *Risk;* (10) *Security; and* (11) *Technology.* Participants understood implantable cardiac devices in a broad variety of ways, including bio-medical terms, both positive (e.g., "life savers") and negative ("site of infection"); and emotional terms, though always negatively (e.g., "anxiety producing," "source of hassle"). Participants described the device's security systems in terms of: security, both secure (e.g., "secure site on the Internet") and insecure (e.g., "bank with an unlocked vault"); risk (e.g., "life threatening"); and information,(e.g., "complete control of information," "black box on a plane").

The diversity of metaphors as well the potential for any given metaphor type to convey both positive and negative dimensions points to the need for security researchers to attend carefully to how stakeholders conceptualize—in lay terms—security for such devices, and how they use those conceptualizations to generate positive or negative perspectives on the security system.

5.3 Security System Concepts

To understand participants' feedback on the six specific security system concepts presented, we adapted a values dams and flows approach. We looked at which systems participants found strongly acceptable (flows)—that is, the systems which many participants liked and very few participants disliked—and which systems participants found less acceptable (dams)—that is, the systems which few participants liked and many participants disliked. This analysis helps inform the interpretation of the following section, in which we present some of the reasons that providers expressed for liking or disliking specific systems.

Evaluations. Table 3 provides an overview of the results from participants' evaluations of the security system concepts presented. The fail-open/safety wristband (shown in Table 3 with green cells) was the best received in all categories: the largest percentage of providers liked it (58%) and would recommend its use (46%), and the smallest percentage of providers disliked it (17%) and would recommend against its use (13%). The UV-visible tattoo of a password (shown in Table 3 with the row of red cells) was the least satisfactory in all categories: only 17% of providers liked it; 54% disliked it; 50% recommended against its use; and only 13% recommended its use. Two other systems reach relatively high thresholds on dislike and recommend against: the medical alert bracelet with a password (46% dislike and 33% recommend against) and the criticality-aware fail-open IMD (42% dislike and 38% recommend against); these data might suggest avoiding the use of those three system concepts.

In general, when we examine these evaluation results, we require high satisfaction thresholds (i.e., high "like" and "recommend" percentages, low "dislike" and "recommend against" percentages) in order to describe a system as well-liked. In contrast, less stringent thresholds are necessary to describe a system as problematic, in order to respect the perspectives of stakeholders who may be in the minority. This is in line with the value sensitive design dams and flows analysis guidelines [24].

Justifications. As noted above, the evaluation results on the security system concepts immediately raise the question of why providers like or dislike a system or would recommend for or against its use. Recall that Table 2 provides a breakdown of some of the properties embodied by the various system concepts, such as requiring physical proximity to the patient or having a manual override. Providers are potentially reacting to these properties in their evaluations. Below we report what system properties providers said they liked and disliked about each system; for the systems that were particularly high-ranked or low-ranked, we break out the relevant properties as lists. Note that participants provided the justifications for each system concept directly after it was presented; participant-supplied justifications are potentially influenced by this fact. The quantitative evaluations were completed after all system concepts had been presented.

Providers N=24	Participant Percentage			
	Like	Dislike	Rec.	Rec. Against
A. Medical Alert Bracelet w/ Password	29	46	21	33
B. Centralized Database	38	21	25	25
C. UV-Visible Tattoo of a Password	17	54	13	50
D. Fail-Open/Safety Wristband	58	17	46	13
E. Proximity-Based Equipment	38	25	38	21
F. Criticality-Aware Fail-Open IMD	38	42	33	38

Table 3. Percentage of participants by security system concept who liked, disliked, recommended, or recommended against each system concept. Green indicates high satisfaction with a system concept; red indicates low satisfaction.

Medical Alert Bracelet with Password (System A). The medical alert bracelet was one of the system concepts most disliked (46%) and recommended against (33%). Providers most frequently expressed disliking System A for the following reasons:

- ↓ Access is not guaranteed—the bracelet may be forgotten, lost, stolen, damaged, or the patient may choose not to wear it. (E.g., "In an accident, the bracelet could be damaged/lost and emergency personnel would not be able to access device")
- ↓ The security is insufficient. (E.g., "Way too easy to maliciously steal password")
- ↓ It visibly indicates to the patient and others that the patient has a condition. (E.g., "Identifies pt. as 'having a problem'")

The relatively poor reception of System A suggests that either these properties are particularly disagreeable to participants, or that the advantages are not sufficient incentive to tolerate the disadvantages. When participants expressed liking that the medical bracelet solution they noted that the system did not depend on other equipment or systems, provided "reassurance" to the patient, was cheap, and provided some security.

Centralized Database (System B). The centralized database was neither one of the highest-rated nor one of the lowest-rated systems. Participants expressed concerns about: the availability of the database across regions, across providers and manufacturers, in case of disaster, or in case of other technical difficulties; how to identify patients to look them up in the database; how to secure the database and identify who is authorized to access it; who will administer the database and how they will fund and maintain it; and the fact that a database would require time away from the bedside to access. In contrast, participants appreciated that this system neither required nor depended upon the patient to wear anything, was theoretically universal, and provided more security than System A (the Medical Alert Bracelet with Password).

UV-Visible Tattoo of a Password (System C). The UV-Visible Tattoo was the lowest-ranked system for all evaluation questions. Participants expressed disliking this system for the following reasons:

- ↓ Required equipment may not be working, accessible, or timely to acquire. (E.g., "requires UV light (i.e. working bulb, power source)")
- ↓ Patients may have cultural, social, or personal objections over a tattoo. (E.g., "religious restrictions against tattoos")
- ↓ Access is not guaranteed—the tattoo could be faded, damaged, or distorted. (E.g., "blood or trauma may obscure tattoo")

↓ Password revocation or changes could be complicated. (E.g., "how to change when device is changed out")

Again, this suggests that the disadvantages outweigh the properties about the system which participants liked: its invisibility in the patient's daily life, both for human and security reasons; and the fact that it is (theoretically) always with the patient, but requires no patient effort.

Fail-Open/Safety Wristband (System D). The fail-open/safety wristband was the highest-rated system across all categories. Participants reported liking the system for the following reasons:

↑ The fail-safe mode guarantees access. (E.g., "GREAT failsafe mode (remove bracelet)")

↑ The system provides some safety features. (E.g., "safety features BIG plus")

↑ The system provides some security features. (E.g., "Provides mechanism against snoops…equivalent to locking your door when you leave the house")

↑ The mechanism gives access control power to the patient. (E.g., "pt. feels empowered. pt. is an active participant in their own care")

↑ Provides a visual cue to EMTs. (E.g., "identifies patient as having an ICD")

Following previous lines of reasoning, these advantages must outweigh the dislikes expressed by participants: that there is no security if the wristband is not worn (and it is easily removed); that the wristband requires battery replacements or recharging and requires the patient to wear it, for which there is no incentive; that there may be many false-positive calls to 911; that the patient is visibly identified as having a medical condition; that emergency medical staff would require training to know to remove the wristband; and that the system is potentially expensive to develop and produce.

Proximity-Based Equipment (System E). The proximity-based equipment was neither one of the highest-rated nor one of the lowest-rated systems; this system most closely resembles the status quo of access control for implantable cardiac devices. Participants expressed liking a variety of system properties: that it provides some security from wireless tampering; that it does not require the patient to wear or do anything (and therefore does not provide a visual indication of the patient's condition); that it does not depend upon other equipment or systems; that it is similar to the current model; that it is easy, and allows bedside access; that it would be a (theoretically) universal access system; and that it gives the patient some control over who may access their device. Conversely, participants reported disliking: that patients are still susceptible to in-person security breaches; that such a system would require new equipment, which is expensive; that such a system would potentially be manufacturer-specific; and that such a system would require all medical centers to have the equipment on-hand and readily accessible for emergencies.

Criticality-Aware Fail-Open IMD (System F). The criticality-aware IMD was one of the systems most disliked (42%) and recommended against (38%). Providers most frequently expressed disliking System F for the following reasons:

↓ The IMD may not correctly identify a medical emergency (false negative—closed access). (E.g., "this assumes the device can properly recognize emergencies → current devices can't even recognize some arrhythmias correctly")

↓ The IMD may incorrectly identify a medical emergency (false positive—open access). (E.g., "possibility of misidentifying a 'medical emergency'")

↓ There may be non-emergency situations where the IMD needs to be accessed. (E.g., "what happens if the patient moves and has a new cardiologist?")

↓ This system could change IMD size or shape, consume battery life, or cost more. (E.g., "will certainly add expense to cost of device such that CMS may veto payment/reimbursement")

As previously reasoned, these disliked properties apparently overpower the properties that participants liked: that it (theoretically) allows access in an emergency; that it provides some security; that it depends upon no extra equipment; and that it does not require the patient to do or wear anything.

6. PROVIDER VS. PATIENT RESULTS

Recall that this medical provider study was conducted in part to complement the prior study on patients' views and values regarding early-stage security solutions for implantable cardiac devices [7]. The set of systems presented to medical providers was the same as those presented to patients with the following exceptions: three somewhat redundant system concepts were removed (i.e., visible tattoo of a password, fail-open wristband, and fail-open/patient-specified-functionality wristband) and one new security concept was included: a centralized database. Additionally, the presentation format was tailored to the participants. For the medical providers, the system concepts were presented in a group setting via a verbal description and supporting slides. For the patients—many of whom were older or ill—the system concepts were presented to each individual with a verbal description and a physical mockup as a prop; patients provided verbal feedback as part of their semi-structured (individual) interviews. The questions asked, while similar in substance, were also slightly different as appropriate to the individual's role (as a medical provider or a patient).

We now consider the quantitative results from both studies together, as a way to explore security concepts that might be successful for both sets of stakeholder groups. For the proximity-based equipment approach (System E): in the patient study this security concept was least disliked (0% dislike), and hence might be the most logical system to choose; however, 25% of the medical providers disliked the proximity-based equipment approach and 17% would recommend against its use, making it a less desirable choice overall. In a similar vein, the criticality-aware fail-open IMD (System F) was more liked (27%) than disliked (18%) by patients; however it was more disliked (42%) than liked (38%) by providers. In the case of the criticality-aware fail-open system, we suspect that this difference is primarily due to providers' higher concern regarding the lack of a manual override if the system fails to recognize a medical emergency. The provider results suggest that this approach may not be a suitable solution for securing IMDs.

The patient study recommended a set of three solution choices that, if offered together, might satisfy the desires of most patients: a proximity-based system (System E), a fail-open/safety wristband (System D), and a UV-visible tattoo of a password (System C). Given the strong opposition to UV-visible tattoos among providers (50% would recommend against), however, our new results caution against their use in practice).

In terms of similarity of perspective, the fail-open/safety wristband approach (System D) was the security concept that was least disliked (17% dislike, 13% recommend against) and the most liked by medical providers (58% like, 46% recommend). This concept was also the most liked by the patients (45% like, 27% would choose to use it if it were available).

7. SECURITY DESIGN CONSIDERATIONS

Drawing on a synthesis of the results from the study data, we now provide concrete design considerations for security researchers working in the implantable cardiac device domain.

7.1 Access, Access, Access

Access—and the related issue of compatibility—show up in both the initial perspectives and the security system concept data sets, and are particularly emphasized in the latter. Participants repeatedly indicated the importance of (different kinds of) access: (a) Providers must always be able to access the implanted device, and security systems should either fail to an open state or offer some kind of override; (b) "Unplanned" access does not only occur in emergencies, since patients may travel or change cardiologists, and records are not always transferred smoothly; (c) Access should not rely upon a centralized system, which could be unavailable (due to technical, geographic, or other reasons) and which merely defers the security problem; (d) Access cannot rely upon a conscious or compliant patient; (e) Access should avoid relying on additional equipment, which can delay or block patient care or remove providers from the bedside; (f) Access should be timely, and should therefore require few steps. Perhaps, above all, in the words of one of our participants: "Please, please, please keep it SIMPLE."

7.2 Mechanics and Logistics

Various aspects of IMD mechanics and logistics are raised in both the initial perspectives and the system concept data sets. Any security system should avoid disturbing the status quo in terms of: (a) cost, which can also affect insurance approval; (b) required training, particularly for non-cardiology staff; (c) implant battery usage; (d) implant size; or (e) any other aspect that might impact the surgical or healing processes.

7.3 Safety Features and Incentives

Participants showed interest in the possibility of incorporating safety features into a security system for IMDs. The exact nature of such features and how they might be tuned should be further investigated; for example, many participants expressed concern that a 911 feature would result in many false-positive emergency notifications. However, if well designed, safety features might function as incentives for patients to comply with using a security system. This is particularly relevant in the case of a system like the wristband (System D), which incorporates an external battery which can be used for safety features, and with which the patient only receives security if they choose to wear the band.

7.4 Empowering Without Burdening

Ideally, patients should be given some implicit or explicit role in the access control process, whether via overt action or by allowing someone extended skin contact. Generally speaking, such a role might give patients a feeling of empowerment, but more practically speaking, patients could provide human reasoning as to whether or not their device should be accessed in a given situation. Conversely, patients should not be unduly mentally or physically burdened by a security system. As one example of this, anything that visually indicates the patient's condition should be opt-in; visual indicators such as medical alert bracelets are useful to emergency staff, but patients should be able to weigh the advantages and disadvantages and choose whether or not to participate. Moreover, this consideration raises a host of ethical, legal, and philosophical questions: Should a security design hinge upon patients being able to choose whether or not they wish to comply? How many patients would actually comply? Should a security design strive to equally protect all patients from potential harm, or is that attitude paternalistic? What are the repercussions, legally or in terms of reputation, if a company's IMD is attacked, and security was optional? The domain is full of interesting questions that are ripe for consideration.

8. CONCLUSION

The work reported here makes two important contributions. First, we offer domain-specific findings for security for implantable cardiac devices—cyber-physical systems that perform critical operations within patients' bodies. The purpose of these findings is to facilitate designs with increased adoption, correct usage, and few negative side effects for patients and medical providers.

Our second contribution concerns method. Specifically, we adapted established methods from value sensitive design—the Envisioning Workshop and values dams and flows—to the security domain: we gathered feedback on classes of security systems via concrete descriptions of systems that embody particular properties (security and non-security). This work demonstrates how security researchers can draw upon direct and indirect stakeholders to understand the relevant properties of a new technology domain; the techniques we used were in the context of implantable cardiac devices, but could be used to explore other emerging technology domains.

Bridging the gap between technological innovation and the lives of stakeholders who will be impacted by that technology is not easy; however, it is critical to do so. Toward that end, the work reported here provides one specific study to suggest how such work could be done, and methods for making progress toward incorporating human values into technical security design.

9. ACKNOWLEDGMENTS

This work was supported in part by the US National Science Foundation (NSF) under awards CNS-0846065, CNS-0905118, and CNS-0905384, by the US Department of Health & Human Services (HHS) under award HHS-2010-03958-09, and by an Alfred P. Sloan Research Fellowship. Any opinions, findings, and conclusions or recommendations expressed in this publication do not necessarily reflect the views of the funding agencies. We thank the medical providers who participated in this study. We thank M. Enev, D. Halperin, K. Koscher, B. Ransford, and A. Takakuwa for contributing to the data analysis process. This work was done while Tamara Denning was at the University of Washington.

REFERENCES

[1] S. Avancha, A. Baxi, and D. Kotz. Privacy in Mobile Technology for Personal Healthcare. ACM Computing Surveys, 45(1), 2013.

[2] D. Balfanz. Usable Access Control for the World Wide Web. ACSAC 2003.

[3] S. Cherukuri, K. Venkatasubramanian, and S. Gupta. BioSec: A Biometric Based Approach for Securing Communication in Wireless Networks of Biosensors Implanted in the Human Body. ICPP Workshops 2003.

[4] S. Clark, T. Goodspeed, P. Metzger, Z. Wasserman, K. Xu, and M. Blaze. Why (Special Agent) Johnny (Still) Can't Encrypt: A Security Analysis of the APCO Project 25 Two-Way Radio System. USENIX 2011.

[5] L. Cranor and S. Garfinkel. Security and Usability. O'Reilly Media, Inc., 2005.

[6] A. Czeskis, I. Dermendjieva, H. Yapit, A. Borning, B. Friedman, B.T. Gill, T. and T. Kohno. Parenting From the Pocket: Value Tensions and Technical Directions for Secure and Private Parent-Teen Mobile Safety. SOUPS 2010.

[7] T. Denning, A. Borning, B. Friedman, B. T. Gill, T. Kohno, and W. H. Maisel. Patients, Pacemakers, and Implantable Defibrillators: Human Values and Security for Wireless Implantable Medical Devices. CHI 2010.

[8] T. Denning, K. Fu, and T. Kohno. Absence Makes the Heart Grow Fonder: New Directions for Implantable Medical Device Security. HotSec 2008.

[9] T. Denning, T. Kohno, H. M. and Levy. Computer Security and the Modern Home. Communications of the ACM, 56(1), 94-103, 2013.

[10] T. Dimkov, A. van Cleeff, W. Pieters, and P. Hartel. Two Methodologies For Physical Penetration Testing Using Social Engineering. ACSAC 2010.

[11] J. L. Fleiss, B. Levin, and M. C. Paik. Statistical Methods for Rates and Proportions (3rd ed.). New York: John Wiley & Sons. 2003.

[12] B. Friedman, D. Howe, and E. Felten. Informed Consent in the Mozilla Browser: Implementing Value Sensitive Design. HICSS 2002.

[13] B. Friedman, D. Hurley, D. C. Howe, E. Felten, and H. Nissenbaum. Users' Conceptions of Web Security: A Comparative Study. CHI 2002 (Extended Abstracts).

[14] B. Friedman, P. H. Kahn Jr., and A. Borning. Value Sensitive Design and Information Systems: Three Case Studies. In P. Zhang and D. Galletta, editors, Human-Computer Interaction and Management Information Systems: Foundations. 2006.

[15] S. Gollakota, H. Hassanieh, B. Ransford, D. Katabi and K. Fu. They Can Hear Your Heartbeats: Non-Invasive Security for Implanted Medical Devices. ACM SIGCOMM 2011.

[16] S. K. S. Gupta, T. Mukherjee, and K. Venkatasubramanian. Criticality Aware Access Control Model for Pervasive Applications. IEEE PERCOM 2006.

[17] D. Halperin, T. S. Heydt-Benjamin, K. Fu, T. Kohno, and W. H. Maisel. Security and Privacy for Implantable Medical Devices. IEEE Pervasive Computing, 7, 2008.

[18] D. Halperin, T. S. Heydt-Benjamin, B. Ransford, S. S. Clark, B. Defend, W. Morgan, K. Fu, T. Kohno, and W. H. Maisel. Pacemakers and Implantable Cardiac Defibrillators: Software Radio Attacks and Zero-Power Defenses. IEEE S&P Symposium 2008.

[19] A. Jøsang, B. AlFayyadh, T. Grandison, M. AlZomai, and J. McNamara. Security Usability Principles for Vulnerability Analysis and Risk Assessment. ACSAC 2007.

[20] F. Kensing and K.H. Madsen. Generating Visions: Future Workshops and Metaphorical Design. In J. Greenbaum and M. Kyng, editors, Design at Work: Cooperative Design of Computer Systems. Lawrence Erlbaum, 1991.

[21] J. Landis and G. Koch. The Measurement of Observer Agreement for Categorical Data. Biometrics, 33, 1977.

[22] C. A. Le Dantec and W. K. Edwards. Designs on Dignity: Perceptions of Technology Among the Homeless. CHI 2008.

[23] C. Li, A. Raghunathan, and N. K. Jha. Hijacking an Insulin Pump: Security Attacks and Defenses for a Diabetes Therapy System. Healthcom 2011.

[24] J. K. Miller, B. Friedman, G. Jancke, and B. Gill. Value Tensions in Design: The Value Sensitive Design, Development, and Appropriation of a Corporation's Groupware System. GROUP 2007.

[25] L. I. Millett, B. Friedman, and E. Felten. Cookies and Web Browser Design: Toward Realizing Informed Consent Online. CHI 2001.

[26] L. P. Nathan, B. Friedman, P. V. Klasjna, S. K. Kane, and J. K. Miller. Envisioning Systemic Effects on Persons and Society Throughout Interactive System Design. DIS 2008.

[27] X. Ou. Ethnographic Fieldwork at a University IT Security Office. ACSAC 2013.

[28] N. Paul and T. Kohno. Security Risks, Low-tech User Interfaces, and Implantable Medical Devices: A Case Study with Insulin Pump Infusion Systems. HealthSec 2012.

[29] A. Raij, A. Ghosh, S. Kumar, M. Srivastava. Privacy Risks Emerging from the Adoption of Innocuous Wearable Sensors in the Mobile Environment. CHI 2011.

[30] K. B. Rasmussen, C. Castelluccia, T. S. Heydt-Benjamin, and S. Capkun. Proximity-Based Access Control for Implantable Medical Devices. ACM CCS 2009.

[31] M. Rushanan, C. Swanson, D. F. Kune, and A. D. Rubin. SoK: Security and Privacy in Implantable Medical Devices and Body Area Networks. IEEE S&P Symposium 2014.

[32] S. Schechter. Security That Is Meant To Be Skin Deep: Using Ultraviolet Micropigmentation To Store Emergency-Access Keys for Implantable Medical Devices. HealthSec 2010.

[33] S. Schechter, R. Dhamija, A. Ozment, and I. Fischer. The Emperor's New Security Indicators. IEEE S&P Symposium 2007.

[34] J. Sorber, M Shin, R. Peterson, C. Cornelius, S. Mare, A. Prasad, Z. Marois, E. Smithayer, and D. Kotz. An Amulet for Trustworthy Wearable mHealth. HotMobile 2012.

[35] Symposium on Usable Privacy and Security. http://cups.cs.cmu.edu/soups/.

[36] E. Troshynski, C. Lee, and P. Dourish. Accountabilities of Presence: Reframing Location-Based Systems. CHI 2008.

[37] K. K. Venkatasubramanian and S. K. S. Gupta. PSKA: Usable and Secure Key Agreement Scheme for Body Area Networks. IEEE Transactions on Information Technology in Biomedicine, 14(1), 2010.

[38] R. Wash. Folk Models of Home Computer Security. SOUPS 2010.

[39] A. Whitten and J. D. Tygar. Why Johnny Can't Encrypt: A Usability Evaluation of PGP 5.0. USENIX 1999.

[40] F. Xu, Z. Qin, C. C. Tan, B. Wang, and Q. Li. IMDGuard: Securing Implantable Medical Devices with the External Wearable Guardian. IEEE INFOCOM 2011.

[41] D. Yoo, M. Lake, T. Nilsen, M. E. Utter, R. Alsdorf, T. Bizimana, L. P. Nathan, M. Ring, E. J. Utter, R. F. Utter, and B. Friedman. Envisioning Across Generations: A Multi-Lifespan Information System for International Justice In Rwanda. CHI 2013.

A Security Evaluation of AIS Automated Identification System

Marco Balduzzi
Trend Micro Research

Alessandro Pasta
Independent Researcher

Kyle Wilhoit
Trend Micro Research

ABSTRACT

AIS, Automatic Identification System, is an application of cyber-physical systems (CPS) to smart transportation at sea. Being primarily used for collision avoidance and traffic monitoring by ship captains and maritime authorities, AIS is a mandatory installation for over 300,000 vessels worldwide since 2002. Other promoted benefits are accident investigation, aids to navigation and search and rescue (SAR) operations. In this paper, we present a unique security evaluation of AIS, by introducing threats affecting both the implementation in online providers and the protocol specification. Using a novel software-based AIS transmitter that we designed, we show that our findings affect all transponders deployed globally on vessels and other maritime stations like lighthouses, buoys, AIS gateways, vessel traffic services and aircraft involved in SAR operations. Our concerns have been acknowledged by online providers and international standards organizations, and we are currently and actively working together to improve the overall security.

1. INTRODUCTION

AIS is an Automatic Identification System introduced to enhance the safety of vessels traffic by automatically exchanging up-to-date information, as well as tracking and monitoring of ships. Since 2002, AIS is a mandatory installation for international voyaging ships having a gross tonnage of at least 300[1], and all passenger ships regardless of their size. Found to be beneficial to the maritime industry, nowadays pleasure crafts and fishing boats are often equipped with an automatic identification system. With an estimated number of over 300,000 installations according to a popular online AIS provider[2], AIS is currently an important and widely used technology and solution in smart transportation. Some known benefits are traffic monitoring, collision avoidance, search and rescue operations, accidents investigations and aids to navigation. Clearly, the number of AIS-equipped vessels might be higher because vessels are not required to register with online providers.

AIS works by acquiring GPS coordinates and exchanging via radio transmissions current and up-to-date information between ships and maritime authorities – i.e. vessel traffic services located onshore. AIS information includes, but is not limited to, ships' position, name and cargo and aids to navigation, frequently used by port authorities to assist a ship's navigation or warn about hazards, low tides, rocky outcroppings and shoals commonly found at sea. In open sea, AIS-enabled distress beacons are used to signal and locate a man overboard. AIS data is collected and exchanged between AIS providers operating over the Internet, which offer AIS data visualization, monitoring and reporting in free or commercial forms.

Given its primary importance and prevalence in maritime traffic safety, we conducted a comprehensive security evaluation of AIS, by tackling it from both a software and a hardware (i.e., radio-frequency) perspective. Overall, we identified threats affecting AIS globally, either at its implementation level or in the protocol specification. They allow, for example, disabling AIS communications (i.e., DoS), tampering with existing AIS data (i.e., to modify the information broadcast by a ship), triggering search and rescue alerts in order to lure a victim ship into navigating to a hostile and attacker-controlled sea space, or spoofing a collision in order to possibly bring a ship off course. Interestingly, according to Bloomberg [3], AIS was found, in the past, to be polluted with counterfeit information, i.e. with Iranian ships having switched their flagged countries to Zanzibar as US and Europe tighten sanctions over their nuclear programs.

In summary, our contributions are the following:

- We conducted a security evaluation of AIS – A cyber-physical system introduced to enhance vessels tracking and provide extra safety to maritime traffic, on top of conventional radar installations;
- We designed and implemented a novel software-based AIS transmitter, that we called *AISTX*;
- We identified and verified several threats affecting both the current implementation and the protocol specification of AIS;
- We did responsible disclosure and collaborated actively with the affected providers, international standards organizations and CERTs to improve the overall situation.

The remainder of the paper is structured as follows. Sec-

[1] Unitless index related to a ship's overall internal volume.
[2] http://www.vesselfinder.com/vessels

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.

ACSAC '14 December 08 - 12 2014, New Orleans, LA, USA
Copyright 2014 ACM 978-1-4503-3005-3/14/12 ...$15.00.
http://dx.doi.org/10.1145/2664243.2664257

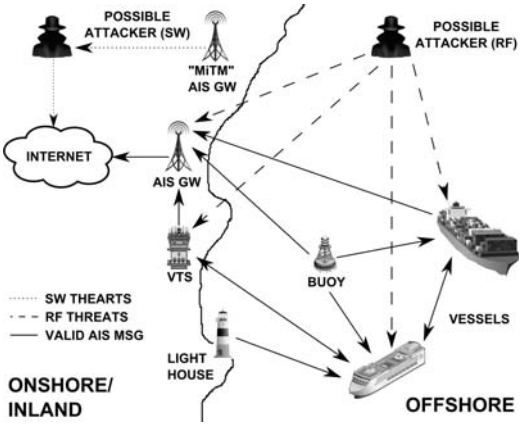


Figure 1: The Automatic Identification System (AIS).

tion 2 introduces AIS and explains how it works. Section 3 gives a general overview of the issues that we identified. Section 4 describes the software-based threats. In Section 5 we introduce and detail *AISTX*, the AIS transmitter that we designed and implemented to conduct the radio-frequency evaluation described in Section 6. Section 7 discusses ethical implications, mitigation strategies and collaboration with the affected parties. Finally, Sections 8 and 9 look over related work and briefly conclude.

2. AUTOMATIC IDENTIFICATION SYSTEM

Since 2002 IMO's SOLAS convention[3] the Automatic Identification System (*AIS*) is required for international voyaging ships having a gross tonnage of 300 or more, and all passenger vessels regardless of their size. Found to be heavily used and widely beneficial to the maritime industry, a second generation of AIS devices, termed *class-B* transponders, were introduced in 2006. When compared with their predecessors, i.e. *class-A*, class-B are typically smaller, lower cost, and simpler to operate. These are typically used in smaller vessels, pleasure vessels, or any fishing vessel less than 300 tons. Since 2010, AIS-related regulations have been constantly adjusted, making it easier to implement and deploy AIS installations. As of 2014, AIS is estimated to be running on at least 300,000 vessels, and in the near future, there is expected to be near a million installations. Nowadays, AIS is a major technology and solution in traffic monitoring and vessels assistance. Shipowners and maritime authorities rely on AIS to supplement traditional radars for collision avoidance and location tracking, in addition to complementary systems like visual observations, audio exchanges and LRIT (long-range identification and tracking).

AIS has many promoted benefits. Any ship transmitting an AIS signal can be received by nearby ships and help make the captains aware of the other shipsâĂŹ location. In addition, AIS provides navigational aids. The aids-to-navigation (AtoN) standard was developed with the ability to broadcast the positions and names of objects other than vessels, such as navigational aid and marker positions and dynamic data reflecting the markerâĂŹs environment (e.g., currents and climatic conditions). These aids can be located on shore, such as in a lighthouse, or on water platforms, such as buoys. Examples of AtoN installations are provided by online AIS providers, such as Marine Traffic[4]. Finally, an example of the importance of the utilization of AIS revolves around accident investigation. Since AIS provides GPS coordinates, course, ground speed, and additional information, it proves more valuable in accident investigation versus the less accurate radar technology used widely today. For the same reason, AIS is largely used in search-and-rescue transponders. AIS-SARTs are self-contained, waterproof devices intended for emergencies, mainly to help the detection and location of vessels and people in distress, i.e. a man overboard.

AIS, as we depict in Figure 1, works by acquiring GPS coordinates and exchanging regional information with nearby stations via VHF, i.e. two radio channels operating at 161.975 and 162.025 MHz, and with AIS providers operating over the Internet. These providers collect data primarily through AIS gateways deployed geographically, i.e. along coast lines and in vessel traffic services (VTS) operated by port authorities. A VTS is a ship traffic monitoring system established by maritime authorities, similar to air traffic control systems for aviation. In addition, single individuals, such as a ship's captain, can upstream AIS data by using a mobile app and a special forwarding software, which duplicates and sends the data to the desired providers as soon as the data is made available. AIS information is broadcasted, collected and exchanged on a regular basis – varying from a couple of seconds to minutes, depending on the type of information and condition of the station. For example, a class-B equipped ship navigating faster than 23 knots is supposed to broadcast its position every 5 seconds. Conversely, an aid to navigation, for example a lighthouse or a buoy notifying an hazard, is sent every 3 minutes.

By regulation, each communicating station, such as a ship, is required to register and obtain valid AIS identifiers, namely MMSI and call-sign, which are issued by official maritime authorities like the US Coast Guard or the Italian Ministry of Economic Development. The maritime mobile service identity (MMSI) consists of a nine-digits number uniquely identifying a station. The first three digits of the MMSI, the maritime identification digits (MID), specifies the country – e.g. 247 for Italy and 338 for United States of America according to ITU-R[5]. The call-signs are a designation for radios, or AIS stations willing to communicate, and are widely used in marine, aviation, military, spacecraft, and by radio amateurs. Finally, AIS information is rendered via chartplotters or providers, e.g. Vessel Finder[6], which, in near real-time, visualize the status of other vessels in the region, navigation aids and other useful maritime information, such as weather forecasts or critical situations. These systems allow worldwide access to AIS statistics, vessels, lighthouses, buoys locations, and corresponding details on a simple, easy to navigate display or website. In addition, AIS information collected from providers is useful in identifying, for example, ships spilling oil in open sea [1] and predicting the financial gain given by marine trading [17].

[3]International Maritime Organization, http://www.imo.org/ourwork/facilitation/documents/solasvonsafetyofnavigation.pdf

[4]http://www.marinetraffic.com/ais/it/datasheet.aspx?datasource=LIGHTHOUSE

[5]http://www.itu.int/online/mms/glad/cga_mids.sh

[6]http://www.vesselfinder.com/

Category	Threat	SW	RF
Spoofing	Ships	✓	✓
	AtoNs	✓	✓
	SARs	✓	✓
	Collisions (CPA)		✓
	Distress Beacons		✓
	Weather Forecasting		✓
Hijacking	Hijacking	✓	✓
Availability Disruption	Slot Starvation		✓
	Frequency Hopping		✓
	Timing Attack		✓

Table 1: Summary of the Identified Threats

3. THREATS OVERVIEW

In this Section, we give a general overview of the threats that we identified in our research. As reported in Table 1, we grouped them in three macro categories, namely spoofing, hijacking, and availability disruption. For each threat, we detail whether it can be performed via software (SW), radio-frequency (RF), or both. Note that Figure 1 includes the information on where attackers fit in the AIS infrastructure. We describe in more detail the software and RF-based attacks later in the paper, respectively in Sections 4 and 6.

Ship Spoofing [SW/RF].

This first threat consists of crafting (i.e., spoofing) a valid non-existent ship. This process involves assigning to the fictitious ship static information, such as the vessel's name, identifiers (i.e., MMSI and call sign), flag, type of ship, cargo type, manufacturer and dimension, and dynamic information like the ship's status (e.g., under way or anchored), position, speed, course and destination. On top of ships, aircraft involved in search and rescue (SAR) operations can be spoofed as well. In fact, SAR aircraft are equipped with AIS class-B transponders as per regulation.

As the reader can imagine, this threat gives an attacker a wide surface of malicious scenarios, such as spoofing a vessel into the jurisdiction of an adversarial nation or making a nuke-carrying cargo sailing the waters of a nuclear-free nation. In addition, vessels spoofing represent an issue for automated systems doing data identification and inference on collected AIS information, for example to detect ships spilling oil in open sea or to predict marine trading. An attacker can counterfeit this information to blame someone else's vessel, for example.

AtoN Spoofing [SW/RF].

Navigational aids, also known as aids-to-navigation (AtoNs), are commonly used to assist vessel traffic, for example along a channel or a harbor, or to warn about hazards, low tides, rocky outcroppings and shoals commonly found in open sea. Spoofing AtoNs consists of crafting fake information to lure a targeted ship into conducting wrong maneuvers. Some examples consist of placing one or more buoys at the entrance of a harbor to tamper with existing traffic, or installing a fake buoy that maliciously instructs a ship into navigating in low water. Given the number of different aids-to-navigations, there are multiple attack scenarios such as for the spoofing of ships.

Collision Spoofing (CPA) [RF].

Collision avoidance is a primary application of AIS, which has been effectively introduced as a system to reduce the risk of collisions among vessels, especially in open sea where no port authority monitoring is in place. The AIS system, in fact, allows for automatic response upon detection and expectation of a collision. This feature is called CPA (closest point of approach) and works by computing the minimal distance between two ships in which at least one is in motion. Using CPA, a ship can be configured to trigger an alert, both visually on the captain's console or acoustically via a siren, and change course in order to avoid a collision. The threat consists of spoofing a ship navigating on a collision course with a targeted vessel. This triggers a collision alert in the CPA system on the victim ship, and could lead the vessel off course into a rock or running it aground during low tide.

AIS-SART Spoofing [RF].

Apart from collision avoidance, AIS is largely used for search and rescue operations. Search and rescue transponders (SARTs) are self-contained, waterproof devices intended for emergency, mainly to help the detection and location of vessels and people in distress, i.e. a man overboard. An AIS-SART activates automatically when in contact with water, and sends a distress radio beacon followed by the GPS position to help aid in locating the survivor. The threat we identified consists of generating a false distress beacon for a man overboard at coordinates chosen by the attacker. By protocol specification, AIS transponders are required to generate an alert when such a message is received. In this scenario, the attacker (i.e., a pirate) triggers a SART alert to lure its victim into navigating to a hostile and attacker-controlled sea space. Note that by law a vessel is required to join a rescue operation upon receiving a search and rescue message.

Weather Forecasting [RF].

One application of AIS is the communication of dynamic data reflecting the changing environment like currents and climate conditions. A special type of messages, namely binary, is used to convey such information. This threat consists of announcing false weather forecasts, for example a sunny day when a squall is expected.

AIS Hijacking [SW/RF].

AIS hijacking consists of altering any information about existing AIS stations, e.g. about the cargo, speed, location and flag of country of a real vessel. As another example, the attacker can maliciously modify the information provided by aids-to-navigation installed at port by authorities for vessels assistance and monitoring. In the software variant of the attack, the attacker eavesdrops (i.e., MiTM) on the communication and replaces AIS information arbitrarily; in the radio-frequency version, the attacker overrides the original AIS message with a higher powered fake signal. In both cases, the recipient receives an attacker-modified version of the victim's original AIS message.

Availability Disruption Threats [RF].

We identified three attacks that go under the same category, namely availability disruption. Since these attacks can be performed only in radio-frequency, we describe their practical implementation later in detail in Section 6. Here,

we summarize them:

- *Slot Starvation*: This attack consists of impersonating the maritime authority to reserve the entire AIS transmission "address space", in order to prevent all stations within coverage in communicating; this includes ships and aids-to-navigation, as well as AIS gateways used in traffic monitoring. As a result, the attacker can disable AIS systems on a large scale;
- *Frequency Hopping*: In a frequency hopping attack, the attacker impersonates the maritime authority to instruct one or more AIS transponders to change their frequencies of operation. By protocol specification, the receiving station is required to maintain the information, which makes the attack persistent even if the system is rebooted. In addition, this operation can be bound to a geographical region, i.e. an attacker can "program" a targeted ship into switching the frequency upon reaching a region chosen by the attacker; this makes AIS useless. Note that for class-B devices, the AIS standard prevents a manual reset of the transponder, and *not* notify the user of the frequency change;
- *Timing Attack*: In this attack, the malicious user instructs the AIS transponder(s) to delay its transmission time – the attacker, by simply renewing the command, can prevent the transponder(s) from further communicating its position. This makes a vessel disappear from the AIS-enabled radars, for example. Inversely, the attacker can overload (i.e., floods) the marine traffic, including ships and vessel traffic services, by requesting the existing stations into sending AIS information and updates at a very-high rate.

4. SOFTWARE EVALUATION

In this Section, we discuss the software-based threats that we identified. We evaluated three popular online AIS providers, namely Marine Traffic[7], AisHUB[8] and Vessel Finder[9], and showed that they are affected by the same threats.

When referencing AIS, we have to address AIVDM, the application layer protocol used by AIS to exchange data sentences, i.e. from vessels' AIS transponder broadcasting their position, or from vessel traffic services (VTS) monitoring the ships at port. AIVDM specifies 27 message types; each one having a corresponding purpose and value that designates its purpose. The full list is given in [10]. For example, message type 1 is used in communications between ships and ship-to-VTS to exchange updated position reports, and message 24 describes the type of ship, its cargo, dimensions and name. For our experiments, we implemented an encoding tool written in Python, named *AIVDM Encoder*, that we used to generate arbitrary AIVDM sentences and conduct both the software and the radio-frequency evaluation of AIS[10].

While AIS installations on ships involve hardware, software is used to upstream AIS data to online providers. While these services are very useful for tracking and navigation, there are security issues with their implementations. Because of the loosely implemented nature of AIS receivers, online providers are often required to accept any data they receive, since they represent a consortium of users and enthusiasts sharing data. This however, introduces several security issues.

AIS providers allow multiple ways of collecting AIS data like pre-formatted emails, mobile apps[11] and forwarding software such as `AIS Dispatcher`[12]. When an AIS message is generated, the forwarding software duplicates and sends the message to the desired providers – e.g., over `UDP/5321` for Marine Traffic. The interval of forwarding these messages can be established, thus forwarding near real-time statistics to AIS providers. The same software can be used to upstream AIS messages received from an AIS gateway, i.e. a local VHF receiver one may have in their home. Gateways are often located along coast lines and present in vessel traffic services operated by port authorities.

Throughout our analysis, we identified security issues with all of the aforementioned online providers. These providers, for example, lack source vetting. They do not check to ensure the message originating for a vessel comes from the same region as the vessel purportedly sending the message. Likewise, there is no authentication present to ensure the vessel sending the AIVDM sentence is the proper sender. As shown in Table 1, the identified problems allow an attacker to carry out both spoofing and MiTM-style attacks against the affected providers as we discuss in the following.

Spoofing consists of crafting valid AIS information remotely, e.g. a non-existent ship or aid to navigation, from nowhere near a body of water or a real AIS station. To verify this threat we first used our *AIVDM Encoder* to generate an innocuous AIVDM sentence indicating low tide in a closed lake nearby. We upload it to the providers by using a generic networking client like `netcat`. Note that message 21 is reserved for AtoNs reports and type 13 is used for buoys. In addition, AtoNs have a MMSI in the form of `99MIDXXXX` as per specification. An example is given in the following Listing:

```
$ ./AIVDM_Encoder.py -type=21 -aid_type=13
                     -aid_name=LOWTIDE
                     -mmsi=993381001
                     -long=9.9400 -lat=45.7821
  | nc -q0 -u 5.9.207.224 5322
```

Listing 1: UDP spoofing example for Marine Traffic.

Second, we generated a pre-formatted email report for a moored vessel, i.e. of Listing 2, and sent it to the receiving address of the targeted provider:

```
To: report@marinetraffic.com

MMSI=247320161
LAT=44.3522
LON=8.5665
SPEED=0
COURSE=243
TIMESTAMP=2013-11-11 13:11
```

Listing 2: Email spoofing example Marine Traffic.

[7] http://www.marinetraffic.com/
[8] http://www.aishub.net
[9] http://www.vesselfinder.com/
[10] The tool will be made publicly available on the Security Intelligence Blog of Trend Micro [24] and on the personal page of one of the authors [2].
[11] http://www.marinetraffic.com/ais/iphone.aspx
[12] http://www.marinetraffic.com/ais/downloads/aisdispatcher.zip

Figure 2: Example of spoofed ship following a pr grammed path.

Finally, we implemented an automated script that us Google Earth's KMZ files to make a spoofed AIS station follow a path over time, e.g. a fictional generic ship spelli the word `PWNED` in the Mediterranean Sea, as depicted Figure 2. Overall, all of the experiments were successf and allowed us to spoof and upstream valid AIS messag to the evaluated providers.

Man-in-the-middle involves the modification or injecti of erroneous data in the AIS communication of a stati transmitting valid AIVDM sentences. We first physically tercepted valid AIVDM sentences transmitted via air fron nearby station (i.e. our AIS transponder[13]), by deploying AIS gateway we controlled. We configured it with `AIS Dispatcher` and an USB dongle AIS receiver[14]. We then used a proxy server to intercept, modify and upstream the AIS messages to the online providers, which accepted any tampered message unhindered. In a second experiment, we picked an existing ship being rendered on online providers[15] and we spoofed via software – as described previously – modified information for the vessel. We were able to make the providers rendering the ship in a different location from where initially located.

5. AIS TRANSMITTER

In the previous Section, we described the software-based threats that we identified, i.e. how to upstream arbitrary AIVDM sentences to AIS providers and how to tamper with existing AIS information sent by real ships to the providers. In the following, we focus on radio-frequency-based threats. We describe the system that we designed and implemented for generating and transmitting arbitrary AIS messages over the air. Later in Section 6, we present the experiments that we conducted to show that our concerns are real and affect all AIS transponders installed on vessels worldwide.

5.1 Architecture

Our AIS transmitter that we called *AISTX*, is designed and implemented as a software defined radio (SDR)[16].

[13]Note that for ethical implications (ref. Section 7), we used our own AIS transponder for this and the following experiment.

[14]http://www.radargadgets.com

[15]We upstreamed the real position broadcast from our transponder.

[16]The transmitter and the *AIS Frame Builder* block we describe later will be made publicly available on the Security Intelligence Blog of Trend Micro [24] and on the personal page of one of the authors [2].

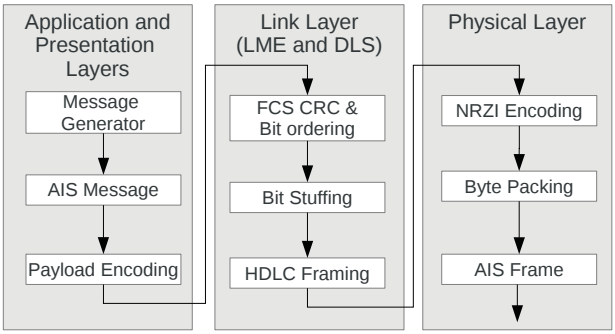


Figure 4: Detail of the *AIS Frame Builder* block.

A SDR consists of a software application, which implements the signal elaboration chain, and a hardware peripheral, which converts binary data to radio-frequency signals for over-the-air transmission. The growth of personal computers' computational capabilities and the drop of hardware acquisition costs have made SDR peripherals available at affordable costs. One example is the universal software radio peripheral (USRP), being one of the most used SDR peripherals. Our *AISTX* is built on top of GnuRadio[17], an open source framework widely used to design and implement efficient software defined radios. While it has been extensively used to conduct research in security, for example in evaluation of the ADS-B aviation protocol [7], GSM security [26] and RFID used in Taipei Metro's EasyCard [5], we believe we are the first to adopt it for building an AIS transmitter.

Figure 3 shows the general architecture of *AISTX*. We generate the AIS frames with a block called *AIS Frame Builder*, thoroughly described later in Section 5.2. We then realize a GMSK modulation of each frame, and over the two AIS channels. GMSK (Gaussian Minimum Shifting Keying) is a form of digital modulation widely used in mobile communication, e.g. in GSM and DECT transmission. The GMSK modulator is configured with parameters as per AIS specification [11]: A bandwidth-time product (BT) of 0.4, a bit rate (R) of 9,600 bps, and a samples/symbol as rounded ratio of sampling and bit rate.

The modulated frames, generated at baseband, are then transposed to the default operating frequencies that the AIS specification dictates. We first modulate the baseband signal around the frequencies of $\pm$ 25 KHz, by multiplying it by two sinusoids of said frequency. To prevent signal clipping and linearity distortion in the digital-to-analog converter (DAC) of the SDR peripheral, the amplitude range of the signal is reduced from $\pm$ 1 to $\pm$ 0.9. Finally, we shift the resulting signal over the carrier frequency of 162 MHz, resulting in two signals over the AIS standard frequencies of 161.975 MHz and 162.025 MHz. The *UHD: USRP Sink* block acts as a driver for the SDR peripheral.

5.2 Building an AIS Frame

Although GnuRadio comes with a wide range of commonly-used pre-defined blocks – e.g. filters, signal generators and converters – for the purposes of our work, we extended the

[17]http://gnuradio.org

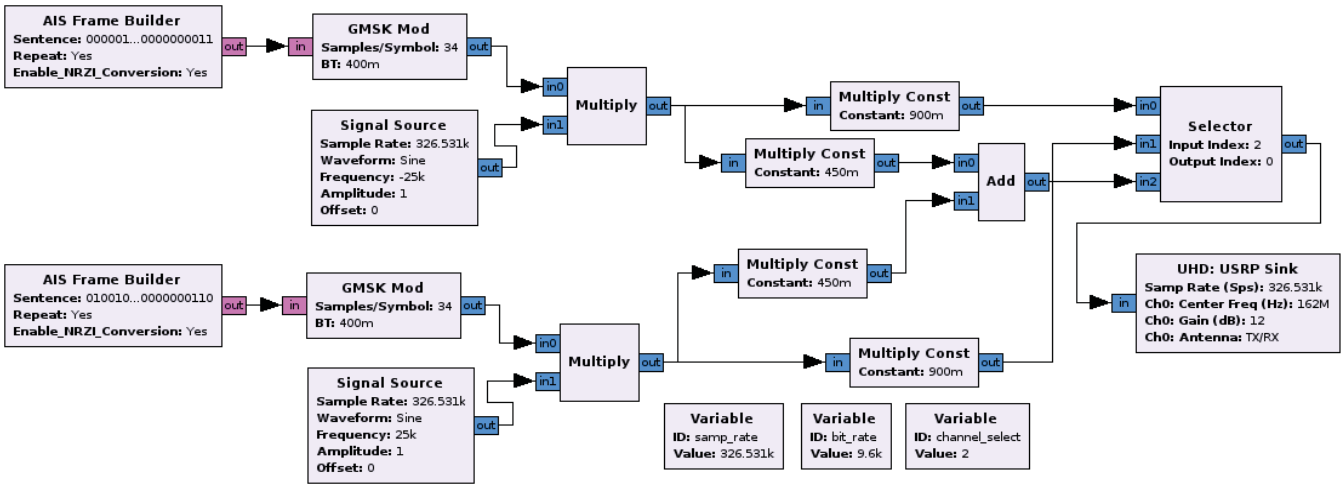


Figure 3: *AISTX* architecture on GnuRadio.

suite with a custom block, named *AIS Frame Builder*, which serves as generator of AIS frames. As Figure 4 shows, our block implements the full AIS stack, and it is composed of three main components covering respectively the application/presentation layers, the link layer and the physical layer, as defined in the protocol specification for AIS. Our block works as following. It takes as input a AIS message, formatted using the AIVDM format. At first, it encodes the message using a 6-bit ASCII alphabet. Only capital letters are supported, i.e. lower-cases are substituted with upper-cases. Numbers are in decimal notation and negatives are expressed using a 2's complement. A padding with zeros (0's) to a multiple of 8 bits is done for further processing. The reason behind the use of such a short alphabet is to reduce the average message length transmitted over the air.

In the link layer module, we compute a frame check sequence of the previous AIS-encoded messaged (i.e., the payload), by using a 16-bit polynomial cyclic redundancy check. This is used by the receiver to validate the integrity of our AIS message. We also reorder the bits of the payload from big-endian to little-endian as per specification. Next, application of bit stuffing occurs – a technique consisting of inserting a zero (0) if five consecutive ones (1's) are found in the bit stream. Bit stuffing reduces the chance of errors in communication and makes sure that HDLC control information is always in the same position.

This control information is added during HDLC framing and consists of a training sequence and of a start/ending flag. The training sequence are twenty-four bits of alternating 0's and 1's (010101010...) and is used to synchronize the receiver to the data stream; the start/end flag consists of a 8-bit pattern 01111110 (0x7E) and is used to delimit the payload portion. Although this flag consists of 6 bits of consecutive ones (1's), it is not subjected to bit stuffing because the flag is meant to act as a delimiter. HDLC is used to synchronize the sender and receiver, and permits synchronous, code-transparent data transmission. This concludes the operations carried out in the link layer module.

Finally, the physical layer prepares the frame for the GMSK modulation, which takes place following in the AIS transmission chain, as we described in Section 5.1 and in Figure 3. The data is encoded using the NRZI mapping, which stands for non-return-to-zero inverted mapping. NRZI is a method for mapping binary digits to a physical waveform and it is characterized by two levels (high and low). The mapping works by having a signal transition on the clock boundary whenever a logical 0 has to be represented, and by keeping the signal at level whenever a logical 1 is transmitted. Note that AIS, similarly to USB, adopts a reverse transition convention with respect to the common use of NRZI.

To conclude, the frame is then packed, i.e. transformed from a bit to a byte representation in order to fit the modulation requirements of the following GMSK block.

6. RADIO-FREQUENCY EVALUATION

Our radio-frequency evaluation consists of two experiments. In the first, we conducted an in-lab experiment in which we used three AIS receivers to verify that our transmitter works well in practice and can be used to run the attacks we identified. Later, we conducted a coverage assessment, in which we verified that a malicious actor can perform such attacks in an open air environment like onshore or open sea, targeting real vessels and authorities.

Our experimental setup consisted of a transmitter acting as an attacker and of a receiver acting as victim (e.g., a vessel or a vessel traffic service). For the transmitter, we used a standard Linux machine running our *AISTX* transmitter together with a USRP device, i.e. a commonly used and well-supported SDR peripheral for transmitting signals over the air. Our USRP consisted of an Ettus USRP B100 completed with a WBX-model daughter-board[18]. This device was well suited for our requirements because it supports VHF maritime frequencies. From the receiver standpoint, we evaluated our ability to generate AIS signals over three distinct AIS receivers: A commercial and standard class-B transponder called Weatherdock EasyTRX2[19], a hardware receiver called AIS em-track R100[20], and a hybrid receiver (hardware and software). The latest is based on a software-

[18] https://www.ettus.com/product/details/UB100-KIT
[19] http://www.easyais.de/en/product_page.php?prodid=33
[20] http://www.em-trak.com/PRODUCTS/Receiver/Receiver.aspx

based AIS receiver[21] and a standard YAESU VHF radio, which we modified by adding an additional output port in order to interface the radio with a computer's audio port and bypass the final-stage audio filter. Note that we connected receivers and transmitter by physical cable to prevent any signal from propagating over the air[22].

As said, we used our *AISTX* transmitter to generate the appropriate AIS messages and perform the attacks, and the receivers to validate the threats. We analyzed the receivers' behavior in three ways: At the hardware layer, at the presentation layer and at the application layer. From a hardware standpoint, we equipped the AIS receivers with alarming lights, in order to observe the behavior of a transponder installation when an alarm is triggered. At the presentation layer, we used the serial port provided by the AIS receivers to monitor the AIS messages received. In fact, via the serial port, we could see all AIS-demodulated messages, including those that are not handled by the software at the application layer (e.g., message type 22, which is used to control the operating frequency of the transponder). Finally, we used a standard chart-plotter software, i.e. OpenCPN[23], to evaluate the receivers' behavior at the application layer. OpenCPN implements a fully-functional chart plotter for personal computers and supports all AIS message types, including SART and CPA alerting. We used OpenCPN to visually render the results of the received malicious messages, i.e. impersonating the on-board computer normally installed on vessels. Together with OpenCPN, for the spoofing attacks we also used the official monitoring tool offered by one of the transponders, i.e. EasyTRX2. This enabled us to understand how and in which way the target was impacted by our misleading transmissions.

We started our evaluation by using the *AIVDM Encoder* introduced in Section 4 to provide *AISTX* with valid AIVDM sentences for the *spoofing-related attacks*. The Listing 3 gives an example for the generation of an AIS radio message on channel A (161.975 MHz) for a spoofed vessel called FOO, having Italian nationality, i.e. the MMSI prefix is 247, and navigating at 100 knots East out of the coast of Sardinia, at the coordinates 43.01N,8.46E.

```
$ ./AIVDM_Encoder.py -type=24 -mmsi=247320160
                     -vname=FOO -csign=FOO
H3co>H0Htt0000000000000000
$ ./AiS_TX.py -payload=H3co>H0Htt0000000000000000
              -channel=A

$ ./AIVDM_Encoder.py -type=1 -mmsi=247320160
                     -speed=100 -course=83
                     -long=8.46 -lat=43.01
13co>HgP?`0VfQ0HW4d3?gw<0000
$ ./AiS_TX.py -payload=13co>HgP?`0VfQ0HW4d3?gw<0000
              -channel=A
```

Listing 3: Ship spoofing in radio-frequency (static and dynamic reports).

Figure 5 shows that the receiver correctly interpreted the spoofed message as a valid vessel and reported it on the monitoring tool. The same methodology of attack is confirmed to work well for generating different types of ships (e.g., law enforcement, military, search and rescue, carrying dangerous goods, etc...), aids-to-navigation like buoys and SAR aircraft, giving the ability to inject malicious and bogus information in AIS traffic, targeting both vessels and authorities.

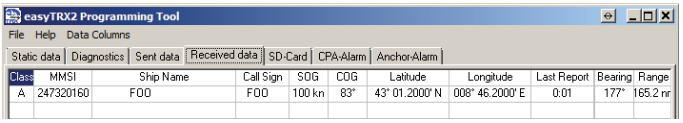


Figure 5: The EasyTRX2 monitoring tool correctly interpreted our spoofed vessel.

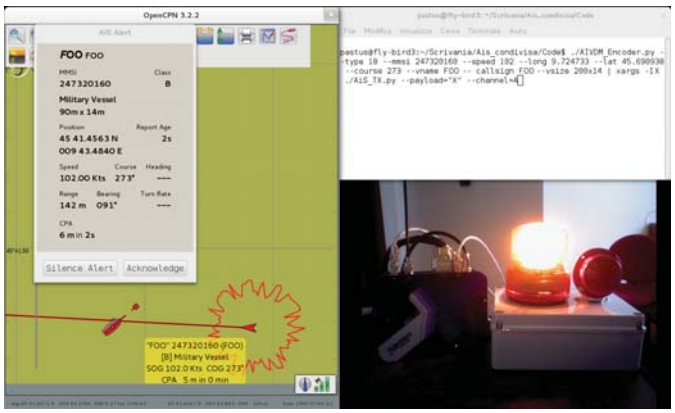


Figure 6: Collision alert being triggered via signaling light and software (OpenCPN).

We then focused on *collision spoofing* (ref. Section 3). For this attack to work, we spoofed a vessel colliding with our AIS transponder. We used the coordinates and course of the targeted transponder to fake a ship navigating within the CPA threshold-alarm configured in the equipment. Figure 6 shows the result. On the top-right window, the attacker sent the spoofed vessel's position report to the target, shown in the other two windows. On the bottom-right, we see the signaling light being triggered because a ship is expected to collide. The monitoring console of OpenCPN (on the left) confirmed the alert by informing that the collision is expected in 2 seconds and 6 meters away. Depending on the final configuration, this threat could lead the targeted vessel off course.

In a similar way, we confirmed that malicious *weather forecasting* is achievable. We used AIS+[24], a chart-plotting software supporting AIS binary messages normally employed to broadcast weather forecasts by port authorities (i.e., type 8). This verified that weather-specific AIVDM sentences spoofed via RF, such as `!AIVDO,1,1,5,A,8>jR06@0Gwli:QQUP3en?wvlFR06EuOwgwl?wnSwe7wvlOwwsAwwnSGmwvhO,0*51`, were correctly interpreted by the AIS receivers and reported by the tool.

A malicious user can generate search and rescue messages to trigger a *SART alert*, for example to lure the victim into navigating to the coordinates chosen by the attacker. In fact, by protocol specification and legislation, AIS stations are required to trigger an alert when a distress beacon is received, i.e. informing the captain that a rescue operation is needed for a man overboard. In our experiments, we exper-

[21] https://www.cgran.org/wiki/AIS
[22] A 90 dB in-line attenuator was installed to reduce the transmitter's power according to the receivers' specification
[23] http://www.opencpn.org
[24] http://www.aisplus.vtt.fi/

Figure 7: AIS Hijacking in radio-frequency.

imentally verified that we could appropriately spoof such messages and trigger the alert of our three AIS receivers, both visually and acoustically. The strategy was to emulate an AIS-SART transmitter by generating over both AIS channels a class-A position report (i.e. a message type 1) having MMSI in the form of `970YYXXXX`: The prefix (970) is reserved for AIS-SARTs, opposed to other MMSI prefixes that specify the country. The Ys are assigned by CIRM[25] and refer to the SART manufacturer, while the Xs are sequential digits assigned by the manufacturer identifying the SART. The command reported in Listing 4 was used.

```
$ ./AIVDM_Encoder.py -type=1 -mmsi=970010000
                     -lat=45.6910 -long=9.7235
  | xargs -I X ./AiS_TX.py -payload=X -channel=A,B
```

Listing 4: Distress beacon (SART) spoofing in radio-frequency.

We have impersonated fictional ships, buoys and other AIS stations, showing that spoofing threats are real and affect standard AIS transponders used worldwide in maritime traffic.

Next, we describe *AIS hijacking* (ref. Fig 7). Bob, the victim, communicates its AIS information to Alice, an AIS receiver installed onshore from authorities or another ship navigating nearby. The attacker, Mallory, is situated within the radio-frequency coverage of Bob and Alice's AIS stations. She generates a modified AIS message for Alice, pretending to be Bob, and overrides the legitimate communication of Bob by transmitting with higher power. Using a physical cable, we connected Bob's AIS transponder and Mallory's SDR-transmitter to Alice's receiver's ports. We simulated Bob's lower output power (when compared with Mallory), by installing a 120 dB attenuator on its connection with Alice (i.e., 30 dB attenuation more than Mallory). By monitoring Alice's receiver, we proved that Mallory is able to override Bob's signal, i.e. to tamper with valid AIS information sent via radio-frequency.

Following, we verified the *availability disruption* threats by using a particular class of AIS messages, called control messages, which are reserved to port authorities and *not* supported in transmission by transponders, i.e. only in reception. Control messages are used by port authorities to control the maritime traffic and have higher priority with respect to normal AIS traffic, e.g. position reports.

For the *frequency hopping* attack, we verified that by broadcasting the message shown in Listing 5 we immediately switched the receivers to non-standard channel frequencies (i.e., we lowered their operating frequencies of 4.950 MHz). This made the devices unavailable to receive, i.e. to know the position of nearby ships, and to transmit, i.e. to broadcast their positions. Note that this operation was applied immediately because we specified a geographical region including their current position. As an alternative form of attack, an attacker can "program" the devices to disappear from AIS monitoring upon entering a region of interest for the attacker, for example the sea quadrant of Somalia[26].

```
$ ./AIVDM_Encoder.py -type=22
                     -channel_a=2080 -channel_b=2081
                      -ne_lat=45.8 -ne_lon=9.9
                      -sw_lat=45.5 -sw_lon=9.5
F3co>HR22240;VQcF0FA3EB20000
$ ./AiS_TX.py -payload=F3co>HR22240;VQcF0FA3EB20000
              -chan=A
```

Listing 5: Example of availability disruption by frequency hopping.

In a similar attack, that we call *timing* attack, the attacker inhibits the transmission capabilities of one or more AIS stations. For example, in the targeted attack of Listing 6, she uses VTS-reserved `assignment command` messages to instruct a victim to delay the transmission for 15 minutes; by iterating through the command, she accomplishes a denial-of-service attack. Inversely, the attacker can overload (i.e., floods) the marine traffic by requesting the existing stations into sending AIS updates at a very-high rate.

```
$ while true; do
  ./AIVDM_Encoder.py -type=23
                     -quiet=15 -target=246100200
  | xargs -I X ./AiS_TX.py -payload=X -channel=A,B;
 sleep 15; done
```

Listing 6: Example of availability disruption by timing attack.

Finally, for the *slot starvation* attack, we used the message types 4 and 20 to simultaneously fake a base station installed at VTS and allocate the AIS transmission "address space" entirely (i.e., the TDMA slots), in order to consume all slots and prevent all nearby stations to further operate (both in transmission and reception).

6.1 Coverage Experiment

In the following, we performed a coverage experiment to simulate the operational conditions of an attacker, e.g. a pirate targeting a ship in navigation, and we showed that our concerns are real. Our coverage experiment consisted of installing our AIS transmitter at a fixed and defined position and a receiving station on a moving car[27]. Our evaluation consisted of generating a harmless testing message with *AISTX* and verifying if, and at which distance, the receiver was able to correctly receive and decode the testing message. Note that as we extensively discuss in Section 7, we took appropriate precautions to conduct the experiment in a safe manner.

We used an amplifier to raise the 50 mW power output of the SDR peripheral to values being in the same order of commercial AIS transponders[28]. This was accomplished through a hardware modification of a traditional VHF transceiver – KENWOOD TK-762G. Note that this device is affordable (i.e., less than 100$ on eBay) and gives attackers an easy

[25] The International Association for Marine Electronics Companies

[26] http://upload.wikimedia.org/wikipedia/commons/7/7e/Somalian_Piracy_Threat_Map_2010.png

[27] As a receiver, we used the Weatherdock EasyTRX2 hardware transponder

[28] 12.5 W for class-A and 2 W for class-B devices as per specification [11]

access to the hardware needed to conduct malicious activities. The hardware modification entailed disconnecting the pre-amplifier's output, i.e. the circuitry from microphone to hybrid module's input, and soldering an external coaxial cable to connect the SDR.

We then built two AIS antennas to simulate a more accurate attack scenario. For the receiver, we used an omnidirectional antenna consisting of a 5 element collinear structure, i.e. a standard installation for ships and VTS. For the attacker, we used a Moxon directional antenna in order to sustain the amplified signal and improve the attacker's coverage and precision. The power gain of our antennas was respectively 6 and 10 dBi.

As we already mentioned, our coverage experiment consisted of transmitting a testing message from a fixed station and of using a movable receiver to verify the coverage experimentally. By using different configurations, as we report in Table 2, we showed that an attacker is able to reach a victim station and convey AIS messages from a distance of approximately 16.5 kilometers, at the least.

For example, by substituting the transmitter's default antenna ($\lambda/4$[29]) with the directional one that we built, the coverage doubled, i.e. from 0.8 to 1.5 kilometers. Further improvements are confirmed when the amplifier is used. Note that our amplifier comes with two selectable output power levels of 5 W and 15 W. These values are in the same order of standard AIS transponders, i.e. class-B and class-A respectively. In these condition, we scored a coverage of 8 and 16.5 kilometers. Note that this value is a lower estimation because our testing site was located near mountains that attenuated the signal we transmitted.

7. RESPONSIBLE DISCLOSURE AND MITIGATION STRATEGIES

Clearly, generating and tampering with AIS information may raise ethical concerns. As researchers, we are fully aware of that and we believe that realistic experiments, like the ones conducted by Jakobsson et al. in [12, 13], are the only way to reliably evaluate attacks in a real-world.

During our experiments, we only used harmless and testing messages and we did *not* interfere with existing systems. We also physically connected our equipment, i.e. *AISTX* and receivers, in order to *not* propagate RF signals over the air. The coverage experiment was conducted on land (at the coordinates 45.69N,9.72E) and we verified that no AIS receiving installations were present by using information made public by online providers; the closest open water (Mediterranean Sea) was 200 kilometers away and all nearby waters were *not* navigable.

As part of responsible disclosure, we reached out upfront on September 2013 the affected providers[30] and involved standards organizations[31]. At the time of writing, we received positive feedback from Marine Traffic and Vessel Finder, and we are actively collaborating with IMO and ITU-R – The latest informed us that they will consider enhancements to AIS, in terms of security, at the World Radiocommunication Conference in 2015. We also shared our concerns with selected CERTs and coastguards, which are interested in discussing the problems with us, standards organizations and vendors.

In addition, we propose and share the following possible mitigation strategies.

Anomaly Detection This strategy consists of applying anomaly detection techniques to AIS data collected, e.g. by online providers and vessel traffic services, in order to detect suspicious activities like unexpected changes in vessels' route or static information. In addition, AIS data can be correlated with satellite information to find incongruities, for example the dimensions of a vessel. Although anomaly detection may be valuable in data collection systems, it seems not a solution for transponders installations on vessels, which remain vulnerable to RF-specific threats like availability disruption and SART spoofing.

X.509 PKI A complementary form of mitigation consists in adopting a public-key infrastructure schema in the AIS protocol used in the RF communications. We suggest X.509 [9], a well-known PKI standard, in which the digital certificates are issued by official national maritime authorities acting as certification authorities[32] and configured in a transponder concurrently with the other station's identifiers, i.e. MMSI and call-sign. X.509 provides authentication on messages exchanged among stations, e.g. between ships and with port authorities. The certificates are handled in two forms: certificates belonging to noteworthy stations like VTSs are preloaded via onshore installations, e.g. when a ship enters a port; generic certificates and certificates previously unknown to a station are exchanged with nearby stations (i.e., vessels in navigation) on demand, during the acquaintance phase of two vessels[33]. Vessels with satellite Internet access can additionally retrieve the certificates from online services.

8. RELATED WORK

A large body of literature focuses on correlating and analyzing ships information collected from vessel traffic services and online providers. Xianbiao et al. in [14] use online analytical processing (OLAP) to store, process and correlate information for collision avoidance and investigation. The same authors discuss in [19] different techniques to organize AIS-collected data in an efficient manner. Carthel et al. in [4] research on multi-sensor networks for data surveillance. They suggest to extend DMHT tracking, an algorithm originally designed for undersea surveillance networks, to AIS. Similarly, authors in [23] suggest ways to increase the maritime domain awareness (MDA) by collecting and using AIS data. Other applications consist of correlating oil slick shape and tracking data to identify ships illegally spilling oil in the sea [1] and of predicting the financial gain given by commercial trading [17]. With respect to navigational safety, most of the literature focuses on collision-avoidance systems and prevention, for example as reported in [15] and [16].

Despite the large body of AIS research, and to the best of our knowledge, we are the first to conduct a security evaluation of the automatic identification system. We use software-defined-radio to build a novel AIS *transmitter*, namely *AISTX*, and to show that our concerns are real. With respect to existing AIS receivers, we report [8], [18], [21], and [25]. Both

[29]For AIS, a $\lambda/4$ antenna measures 46 cm of length. Lambda corresponds to the wavelength (i.e., $c/162$ MHz.)

[30]Marine Traffic, AisHUB and Vessel Finder

[31]IMO, IALA and ITU-R

[32]The same organizations issuing MMSI and call-sign identifiers for AIS stations, e.g. the US Coast Guard or the Italian Ministry of Economic Development

[33]There is no need to exchange the certificates over trusted channels because counterfeiting will invalidate them

TX Antenna	RX Antenna	Amplifier	Output Power [W]	Coverage [Km]
Default ($\lambda/4$)	Omni		0.05	0.8
Directional	Omni		0.05	1.5
Directional	Omni	✓	5	8
Directional	Omni	✓	15	16.5

Table 2: Measured coverage.

gr-ais [8] and ais-tool [25] are software-based receivers built on top of GnuRadio. In particular, we used gr-ais in the hybrid receiver introduced in our radio-frequency evaluation. When compared with Guarnieri [6], our work looks at faults in the AIS's implementation and protocol specification, while his work focuses on data leaked from AIS data collected in Internet Census 2012[34].

Similarly to our research in smart transportation, Costin et al. [7] perform a security evaluation of ADS-B[35], a radio-frequency protocol used in aviation for data communication and monitoring. Using software-defined-radio, they show that ADS-B is vulnerable to eavesdropping, message jamming and replaying of injection. In a similar work, Teso [22] shows how to use ACARS[36] to upload malicious flight management system (FMS) plans to aircraft. More recently, Humphreys et al. in [20] introduce a software-based GPS transmitter to fool GPS communication and demonstrate how to hijack valid GPS signals to bring a ship off course.

9. CONCLUSIONS

AIS is a cyber-physical system commonly used in the marine industry for vessels traffic monitoring and assistance. Given its importance in collision detection, search and rescue operations and piracy prevention, we conducted a unique security evaluation of AIS. Using a software-based transmitter that we introduced, we discovered and experimentally proved that both the AIS's implementation and the protocol specification are affected by several threats, offering malicious actors many attack possibilities. We did responsible disclosure and notified upfront with mitigation strategies the affected providers and involved international organizations, with which we are currently and actively collaborating in order to improve the overall security. We hope that our research will help in this direction.

Acknowledgements The authors would like to thank Germano Valbusa – call sign IW2DCK – for contributing to the development of the amplifier (ref. Section 6.1). A special thanks goes to the Forward-Looking Research team and Trend Micro who supported the research in different forms. Finally, we thank the organizers of both Black Hat and Hack In The Box conferences for hosting our talk on AIS[37].

10. REFERENCES

[1] C. Ambjorn. Seatrack web forecasts and backtracking of oil spills. efficient tool to find illegal spills using ais.
[2] M. Balduzzi. Personal Page. http://iseclab.org/people/embyte/.
[3] Bloomberg. Iran Oil Tankers Said by Zanzibar to Signal Wrong Flag. http://www.bloomberg.com/news/2012-10-19/iranian-oil-tankers-said-by-zanzibar-to-be-signaling-wrong-flag.html.
[4] C. Carthel, S. Coraluppi, and P. Grignan. Multisensor tracking and fusion for maritime surveillance.
[5] C.-C. Chen, I.-T. Chen, C.-M. Cheng, M.-Y. Chih, and J.-R. Shih. A practical experience with rfid security.
[6] Claudio, Guarnieri. Spying on the Seven Seas with AIS.
[7] A. Costin and A. Francillon. Ghost in the Air(Traffic): On insecurity of ADS-B protocol and practical attacks on ADS-B devices. In *BLACKHAT 2012*.
[8] N. Foster. GnuRadio AIS Receiver. https://www.cgran.org/wiki/AIS.
[9] R. Housley, W. Ford, W. Polk, and D. Solo. Rfc 5280: Internet X. 509 Public Key Infrastructure Certificate and CRL profile, 2008.
[10] R. S. I.-R. International Telecommunication Union. AIS AIVDM Message Types. http://www.navcen.uscg.gov/?pageName=AISMessages.
[11] ITU-R. Technical characteristics for an automatic identification system using time-division multiple access in the VHF maritime mobile band. http://www.itu.int/dms_pubrec/itu-r/rec/m/R-REC-M.1371-4-201004-I!!PDF-E.pdf.
[12] M. Jakobsson, P. Finn, and N. Johnson. Why and how to perform fraud experiments. *Security Privacy, IEEE*.
[13] M. Jakobsson and J. Ratkiewicz. Designing ethical phishing experiments: a study of (rot13) ronl query features. In *Proceedings of WWW 2006*.
[14] X. Ji, Z. Shao, J. Pan, and C. Tang. *A New AIS-Based Way to Conduct OLAP of Maritime Traffic Flow*.
[15] L.-n. LI, S.-h. YANG, B.-g. CAO, and Z.-f. LI. A summary of studies on the automation of ship collision avoidance intelligence. *Journal of Jimei University*.
[16] L. Li-na. Determination of the factors about safe distance of approach and etc on the research of ship automatic avoidance collision.
[17] B. L.P. Bloomberg Commodities. http://www.bloomberg.com/professional/markets/commodities/.
[18] K. F. Mathapo. A software-defined radio implementation of maritime AIS. https://scholar.sun.ac.za/handle/10019.1/2215.
[19] Z. Shao, C. Tang, J. Pan, and X. Ji. The application of database techniques in the integrated vessel information service system.
[20] D. P. Shepard, T. E. Humphreys, and A. A. Fansler. Evaluation of the vulnerability of phasor measurement units to gps spoofing attacks.
[21] O.-S. Software. Gnu AIS. http://gnuais.sourceforge.net/.
[22] H. Teso. Aircraft Hacking - Practical Aero Series.
[23] B. Tetreault. Use of the automatic identification system (ais) for maritime domain awareness.
[24] TrendMicro. Security Intelligence Blog. http://blog.trendmicro.com/trendlabs-security-intelligence/category/internet-of-everything/.
[25] R. Undheim. Ais-Tools. http://www.funwithelectronics.com/?id=9.
[26] F. van den Broek. Eavesdropping on gsm: state-of-affairs. 2011.

[34] http://internetcensus2012.bitbucket.org/paper.html
[35] Automatic Dependent Surveillance-Broadcast
[36] Aircraft Communications Addressing and Reporting System
[37] The slides are available on the personal page of one of the authors [2].

NodeSentry: Least-privilege Library Integration for Server-Side JavaScript

Willem De Groef
iMinds–DistriNet
KU Leuven, Belgium
Willem.DeGroef@cs.kuleuven.be

Fabio Massacci
University of Trento
Fabio.Massacci@unitn.it

Frank Piessens
iMinds–DistriNet
KU Leuven, Belgium
Frank.Piessens@cs.kuleuven.be

ABSTRACT

Node.js is a popular JavaScript server-side framework with an efficient runtime for cloud-based event-driven architectures. Its strength is the presence of thousands of third-party libraries which allow developers to quickly build and deploy applications. These very libraries are a source of security threats as a vulnerability in one library can (and in some cases did) compromise one's entire server.

In order to support the least-privilege integration of libraries, we developed NodeSentry, the first security architecture for server-side JavaScript. Our policy enforcement infrastructure supports an easy deployment of web-hardening techniques and access control policies on interactions between libraries and their environment, including any dependent library.

We discuss the implementation of NodeSentry, and present its practical evaluation. For hundreds of concurrent clients, NodeSentry has the same capacity and throughput as plain Node.js. Only on a large scale, when Node.js itself yields to a heavy load, NodeSentry shows a limited overhead.

Categories and Subject Descriptors

K.6.5 [**Management of Computing and Information Systems**]: Security and Protection; H.3.5 [**Information Storage and Retrieval**]: Web-based services

Keywords

Web security, JavaScript

1. INTRODUCTION

Services offered on the web have a standard conceptual architecture: a client (or tenant) accesses a web application which talks to one or more databases [8]. In order to serve multiple clients, the traditional approach (represented by e.g., Apache and IIS) has been to duplicate the entire path for each client at the process level. In order to cope with increasing demands, modern services (e.g., Salesforce, SAP-ByDesign) have evolved to multi-tenancy event-driven architecture: different tenants access the same pipe which takes care of the different events by an event-driven program [41].

The major reason behind the success of event-driven programs is that they offer developers a much finer control (and therefore better performance) than switching between application processes [41, 17]. Among the various event-driven programming languages, Node.js is a widely successful platform that combines the popular JavaScript language with an efficient runtime tailored for a cloud-based event architecture [34].

JavaScript has many advantages for web development [14]. It is the de facto dominant language for client-side applications and it offers the flexibility of dynamic languages. In particular it allows the easy combination or mash-up of content and libraries from disparate third parties. Such flexibility comes at a price of significant security problems [26, 33], and researchers have proposed a number of solutions to contain them: from sandboxing (e.g., Google's Caja or [35, 2]) and information flow control [9, 10] to instrumenting the client with a number of policies [30], or trying to guarantee control-flow integrity at a web-firewall level [6]. Bielova presents a good recent survey on JavaScript security policies and their enforcement mechanism within a web browser [4]. These proposals are appropriate for client-side JavaScript but cannot be lifted to server-side code. At first, they assume that the client is not running with high-privileges; second they command a significant overhead acceptable at client side but not at server side. For example, Meyerovich's et al., [30] report some of the best micro-benchmarks for client side JavaScript and still report an overhead between 24% to 300% of the raw time.

Security problems are magnified at server side: applications run without sandboxing and serve a large number of clients simultaneously; server processes must handle load without interruptions for extended periods of time. Any corruption of the global state, whether unintentional or induced by an attacker, can be disastrous. Unfortunately, JavaScript features make it easy to slip and introduce security vulnerabilities which may allow a diversion of the control flow or even complete server poisoning. Hence, developers should be cautious when developing server applications in JavaScript, yet the current trend is to build up one's application by loading (dynamically) a large number of third-party libraries. Figure 2 shows the libraries integrated in one of the most popular web application servers based on Node.js. Verifying such a massive amount of third-party code, especially in a

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.
ACSAC '14 December 08 - 12 2014, New Orleans, LA, USA
ACM 978-1-4503-3005-3/14/12 ...$15.00.
http://dx.doi.org/10.1145/2664243.2664276

language as dynamic and flexible as JavaScript, is close to impossible [43, §6].

How do we combine the flexibility of loading third-party libraries from a vibrant ecosystem with strong security guarantees at an acceptable performance price? There is essentially no academic work addressing the problem of server-side JavaScript security. Our paper targets this gap.

1.1 Contributions

This paper proposes a solution to the problem of least-privilege integration of libraries with the following contributions:

1. NodeSentry, a novel server-side JavaScript security architecture;
2. Policy infrastructure that allows to subsume and combine common web-hardening techniques and measures, common and custom access control policies on interactions between libraries and their environment, including any dependent library;
3. Description of the key features of NodeSentry's implementation and its policy infrastructure in Node.js;
4. Practical evaluation of the performance of our solution.

In summary we show that for hundreds of concurrent clients NodeSentry is essentially close to its theoretical optimum, between 250-500 concurrent clients NodeSentry exhibits an increasing drop in capacity and after 500 move in synch with Node.js's own drop in performance reaching 50% of the theoretical optimum (while Node.js is at 60%).

The rest of this paper is structured as follows. Section 2 sketches the necessary background on Node.js and the security problems of its ecosystem of third-party libraries. Section 3 describes the exact threat model and gives a general overview of our solution, called NodeSentry. Section 4 discusses how NodeSentry can be used in practice and how it protects against real-life attacks. In Section 5, we exemplify several real-life policies and Section 6 gives insight into the implementation. Section 7 discusses the quantitative evaluation of the performance. Finally, Section 8 discusses related work, and Section 9 summarizes the contributions.

2. BACKGROUND ON Node.js LIBRARIES

Node.js by itself only provides core system functionality like e.g., accessing the file system or network communication. Developers that want to build applications must therefor often rely on third-party libraries. They are distributed as packages, structured according to the CommonJS package format and installable via the de facto standard `"npm"` package manager (by itself a JavaScript package). The official package registry contains more than 70 thousand packages and has more then 290 million downloads each month. Such libraries are statically or dynamically loaded in order to provide the corresponding services.

Node.js module loading system is very easy to use. Via the built-in `require` function, modules living within the base system, in a separate file or directory, can be included in the application. The loading works by reading the JavaScript code (from memory or from disk), executing that code in its own name space and returning an `exports` object, which acts as the public interface for external code. On line 2 of

```
1 var mime = require('mime')
2 var path = require('path')
3 var fs;
4 try { fs = require("graceful-fs") }
5 catch (e) { fs = require('fs') }
```

Figure 1: Code excerpt that shows how different system functionalities are exposed within the Node.js environment by requiring specific libraries.

Figure 1, the variable `path` will be an object with several properties including `path.sep` that represents the separator character or the function `path.dirname` that returns the directory name of a given file path.

Libraries can also be dynamically loaded at any place in a program. For example on line 4, the program first tries to load the `"graceful-fs"` library. If this load fails, e.g., because it is not installed, the program falls back into loading the original system library `"fs"` (line 5). In this example constant string are provided to the `require` function but this is not necessary. A developer can define a variable `var lib='fs'` and later on just call a `require(lib)` function where `lib` is dynamically evaluated.

The resulting ecosystem is such that almost all applications are composed of a large numbers of libraries which recursively call other libraries. The most popular packages can include hundreds of libraries: `"jade"`, `"grunt"` and `"mongoose"` make up for more than 200 included libraries each (directly or recursively); `"express"`, a popular web package includes 138, whereas `"socket.io"` can be unrolled to 160 libraries.

Figure 2 shows a bird's eye view of the library used by the `"npm-www"` JavaScript package maintainer. One of the single nodes of this package tree, is the sub library `"st"` (the fourth node from the left) which is developed specifically to manage static files hosting for the backend of the web site [1]. As you can see, the `"st"` library further relies on access to the `"http"` and `"url"` package to process URLs and on the `"fs"` package to access the file system.

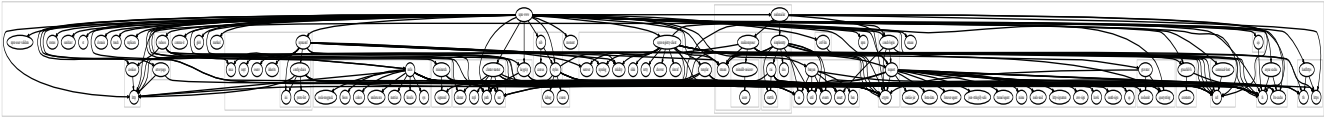
The quote below from a blog post of a Node.js developer clearly explains the sharing principles of the Node.js ecosystem[2]:

> I'm working on my own project, and was looking for a good static serving library. I found the best one, but sadly it was melded tightly to the npm-www project... glad to see it extracted and modularized!

Unfortunately, the resulting `"st"` turned out to be vulnerable to a directory traversal bug[3] which allowed it to serve essentially all files on the server, and thus leading to a potential massive compromise of all activities.

How can one check libraries for potential vulnerabilities? Server-side JavaScript code is not subject to changes as client-side code, so one may hope that static analysis might work.

[1] http://blog.npmjs.org/post/80277229932/newly-paranoid-maintainers
[2] https://github.com/isaacs/st/issues/3
[3] https://nodesecurity.io/advisories/st_directory_traversal & http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-3744



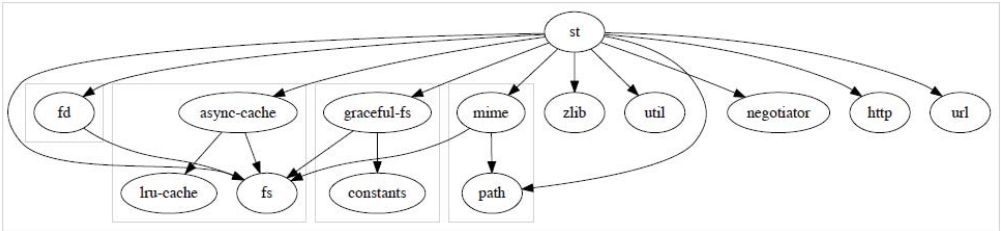


Figure 2: The code that runs http://npmjs.org, which is a Node.js package itself (top image), loads a large number of third parties libraries (which may use further libraries). The fourth node from left is the `"st"` library which further uses additional libraries (bottom image). Static verification is close to impossible.

Unfortunately, the dynamic functionalities and the usage of exceptions alone make static analysis of JavaScript packages extremely difficult: only a handful of frameworks for static analysis can deal with exceptions and dynamic calls [18, 16]. Further, the large quantity of libraries to be considered (and modeled) is another major hurdle. For example JAM requires modeling such dependencies in Prolog [15]. Runtime monitoring seems the only alternative *if* it can scale up to hundreds or thousands of concurrent requests. For client-side JavaScript, for *one* client, an effective implementation like ConScript already tallies a minimum 25% up to 300% overhead.

3. THREAT MODEL AND SOLUTION

The server-side scenario, discussed earlier, assumes that libraries are actually executed on the server with server privileges. Hence, we assume *non-malicious libraries, although potentially vulnerable and exploitable (semi-trusted)*, as for example the `"st"` library. They might end up using malicious objects or doing something they were not intended to do.

The purpose of our security model is to shield the potential untrusted libraries from *some* of the other libraries loaded in the package which may offer a functionality that we consider core. For example we may want to filter access by the semi-trusted library to the trusted library offering access to the file system.

We consider outright malicious libraries out of scope from our threat model, albeit one could use NodeSentry equally well to fully isolate a malicious library. We believe that the effort to write the policies for *all* other possible libraries to be isolated from the malicious one by far outweigh the effort of writing the alleged benign functionalities of the malicious library from scratch.

Given the fact that NodeSentry has a programmatic policy, and that policy code can effectively modify how the enforcement mechanism functions, it could be possible to introduce new vulnerabilities into the system via a badly written policy. However, we consider the production of safe and secure policy code an interesting but orthogonal – and thus out-of-scope – issue, for which care must be taken by the policy writer to prevent mistakes/misuse.

The key idea of our proposal is to use a variant of an inline reference monitor [38, 13] as modified for the Security-by-Contract approach for Java and .NET [11] in order to make it more flexible. Namely, we do not embed the monitor into the code as suggest by most approaches for inline reference monitors but inline only the hooks in a few key places, while the monitor is an external component. In our case this has the added advantage of potentially improving performance (a key requirement for server-side code) as the monitor can now run in a separate thread and threads which do not call security relevant actions are unaffected.

Further, and maybe most importantly, we do not limit ourselves to purely raising security exceptions and stopping the execution but support policies that specify how to "fix" the execution [12, 5, 10]. This is another essential requirement for server side applications which must keep going.

In order to maintain control over all references acquired by the library, e.g., via a call to `"require"`, NodeSentry applies the *membrane* pattern, originally proposed by Miller [31, §9] and further refined in [44]. The goal of a membrane is to fully isolate two object graphs [31, 44]. This is particularly important for dynamic languages in which object pointers may be passed along and an object may not be aware of who still has access to its internal components.

Intuitively, a membrane creates a shadow object that is a "clone" of the target object that it wishes to protect. Only the references to the shadow object are passed further to callers. Any access to the shadowed object is then intercepted and either served directly or eventually reflected on the target object through handlers. In this way, when a membrane revokes a reference, essentially by destroying the shadow object [44], it instantly achieves the goal of transitively revoking all references as advocated by Miller [31].

The NodeSentry-handler intercepts the object references received by the semi-trusted library and can check them for compliance with the policy. Our policy decision point can be seen as a simple automaton: if the handler receives a request for an action and can make the transition then the object proxied by the membrane is called and the (proxied) result is returned; if the automaton cannot make a transition on the input (i.e., the policy is violated), then a security countermeasure can be implemented by NodeSentry or, in the worst case scenario, a security exception will be automatically raised.

We have identified *two* possible points where the policy

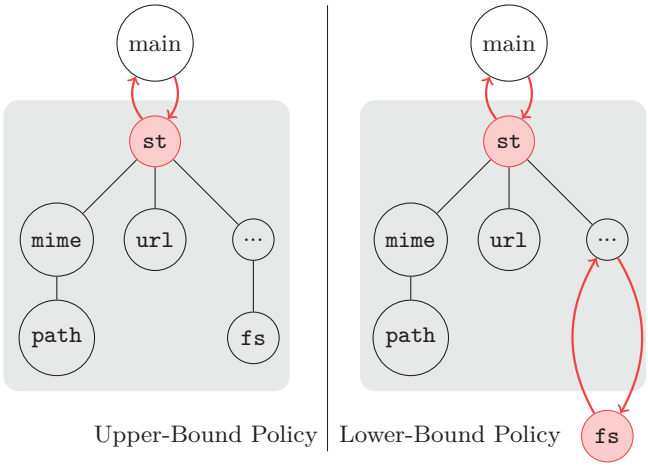


Figure 3: NodeSentry allows policies to be installed on the public interface of the secure library (*Upper-Bound policies*) and/or on the public interface of any depending library (*Lower-Bound policies*).

hooks can be placed that fall together with two distinct types of policies: on the *public interface of the library itself* with the outer world, on the *public interface of any depending library* (both built-in, core libraries and other third-party libraries), or in both places. The choice of the location determines two type of policies:

Upper-Bound policies are set on each member of the public interface of a library itself with the outer world. Those interfaces are used by the rest of the application to interact with it. It is the ideal location to do all kinds of security checks when specific library functionality is executed, or right after the library returns control.

For example, these checks can be used (i) to implement web application firewalls and prevent malformed or maliciously crafted URLs from entering the library or (ii) to add extra security headers to the server response towards a client. Another example of a useful policy would be to block specific clients from accessing specific files via the web server.

Lower-Bound policies can be installed on the public interface of any depending library, both built-in core libraries (like e.g., `"fs"`) or any other third-party library.

Such a policy could be used to enforce e.g., an application-wide *chroot jail* or to allow fine-grained access control such as restricting reading to several files or preventing all write actions to the file system.

Figure 3 depicts interactions with these two types of policies with the red arrows and highlights the isolated context or membrane with a grey box. The amount of available policy points is thus a trade-off between performance (less points mean less checks) and security (more points mean a more fine-grained policy).

A developer wishing to use NodeSentry only needs to replace the `require` call to the semi-trusted library with a `safe_require`. This approach makes it possible to implement

```
// code snippet from st.js
// get a path from a url
Mount.prototype.getPath = function (u) {
    u = path.normalize(url.parse(u).pathname
            .replace(/^[\/\\]?/, '/'))
            .replace(/\\/g, '/')
// ...
```

Figure 4: The `"st"` library has a potential security issue because it does not check the file path for potential directory traversal.

a number of security checks used for web-hardening , like e.g., enabling the HTTP Strict-Transport-Security header [20], set the Secure and/or HttpOnly Cookies flags [3] or configure a Content Security Policy (CSP) [39], in quite a modular way without affecting the work of rank-and-file JavaScript developers. This is described in the next section whereas we illustrate some policy examples more in detail in Section 5.

4. USAGE MODEL

Here we describe the usage model [24] of the NodeSentry library. The developer (such as the one whose blog entry we have cited) has found an appropriate library for her application.

She may now use the library to serve files to clients. As mentioned, the library has a potential potential directory traversal issue, as shown in Figure 4. By itself, this may *not* be a vulnerability: if a library provides a functionality to manage files, it should provide a file from any point of the file system, possibly also using '`..`' substrings, as far as this is a correct string for directory. However, when used to provide files to clients of a web server based on URLs, the code snippet below becomes a serious security vulnerability.

An HTTP request for `/%2e%2e/%2e%2e/etc/passwd`, sent by an attacker towards a server using the `"st"` library to serve files, could expose unintended files.

It is of course possible to modify the original code to fix the bug but this patch would be lost when a new update to `"st"` is done by the original developers of the library. Getting involved in the community maintenance of the library so that the fix is inserted into the main branch may be too time demanding, or the developer may just not be sufficiently skilled to go fix it without breaking other dependent libraries, or just have other priorities altogether.

In all these scenarios, which are the majority of the cases, the application of NodeSentry is the envisaged solution. The `"st"` library is considered semi-trusted and a number of default web-hardening policies are available in the NodeSentry policy toolkit.

The only adjustment is to load the NodeSentry framework and to make sure that `"st"` is safely required so that the policy becomes active, as shown in Figure 5.

The policy rules in Figure 6 can then be activated in the policy section and all URLs passed to `"st"` would be correctly filtered. The policy states that if a library wants to access the URL of the incoming HTTP request (via `IncomingMessage.url`), we first test it on the presence of a (encoded) dot character. If so, we return a new URL that e.g., points to a file that contains a warning message.

```
require("nodesentry");
var http = require("http");
var st= safe_require("st");
var handler = st(process.cwd());
http.createServer(handler).listen(1337);
```

Figure 5: After loading the NodeSentry framework, policies can be (recursively) enforced on libraries by loading them via the newly introduced `safe_require` function.

```
if (method === "IncomingMessage.url") {
    var regex = new RegExp(/%2e/ig);
    if (regex.test(origValue))
        return "/your_attack_is_detected.html";
    else
        return origValue;
}
```

Figure 6: If application code requests the URL of the incoming request, a pointer to a different page is returned whenever malicious characters are detected.

5. POLICY EXAMPLES

In defining the policies, we have tried to be as modular as possible: real system security policies are best given as collections of simpler policies, a single large monolithic policy being difficult to comprehend. The system's security policy is then the result of composing the simpler policies in the collection by taking their conjunction. This is particularly appropriate considering our scenario of filtering library actions.

If the library may not be trusted to provide access to the file system it may be enough to implements OWASP's check on file system management (e.g., escaping, file traversal etc.). If a library is used for processing HTTP requests to a database, it could be controlled for URL sanitization. Each of those two libraries could then be wrapped by using only the relevant policy components and thus avoid paying an unnecessary performance price.

As a simple example for the potential of NodeSentry we describe how we implemented the checks behind the 'helmet' library[4], a middleware used for web hardening and implementing various security headers for the popular `"express"` framework.

It is used to, e.g., enable the HTTP Strict Transport Security (HSTS) protocol [20] in an `"express"`-based web application by requiring each application to actually use the library when crafting HTTP requests. Figure 7 shows a NodeSentry policy that adds the HSTS header before sending the outgoing server response.

The developer does not need to modify the original application code to exhibit this behaviour. They only need to `safe_require` the library whose HTTPS calls they want to restrict. This can be done once and for all at the beginning of the library itself, as customary in many Node.js packages.

The example in Figure 8 shows a possible policy to prevent

[4]https://github.com/evilpacket/helmet

```
if (method === "ServerResponse.write")  {
    var h = "Strict-Transport-Security";
    var v = "max-age=3600; includeSubDomains";
    response.setHeader(h, v);
    // move on with the real
    // ServerResponse.write call
}
```

Figure 7: Before a server response is sent towards a client, the policy first adds the HSTS header, effectively mimicking the behaviour of `helmet.hsts()`.

```
if (method === "fs.writeFile" ||
    method === "fs.write" ||
    method === "fs.writeFileSync" ||
    method === "fs.writeSync" ||
    method === "fs.appendFile" ||
    method === "fs.appendFileSync")  {
    // simply return
    return
}
```

Figure 8: A possible policy that wants to prevent a library from writing to the file system must cover all availalbe write operations of the `"fs"` library.

a library from writing to the file system without raising an error or an exception. Whenever a possible write operation via the `"fs"` library gets called, the policy will silently `return` from the execution so that the real method call never gets executed, and thus effectively prevent writing to the file system. It is possible to change this behavior by e.g., throwing an exception or *chrooting* to a specific directory.

6. IMPLEMENTATION DETAILS

This section reports on our development of a mature NodeSentry prototype, which is designed to work with the latest Node.js versions and relies on the upcoming ES Harmony JavaScript standard. Membranes require this standard, in order to implement fully transparent wrappers, and also build on WeakMaps, to preserve object identity between the shadow object and the real object (1) across the membrane and (2) on either side of the membrane. The main goal of wrapping a library's public API with a membrane, is to be sure that each time an API is accessed, our enforcement mechanism is invoked in a secure and transparent manner.

We rely on the ES Harmony reflection module shim by Van Cutsem[5] and its implementation of a generic membrane abstraction, which is used as a building block of our implementation and is shown in Figure 9. The current prototype runs seamlessly on Node.js v0.10.

Our first stepping stone is to introduce the `safe_require` function (see Figure 10) that virtualizes the `require` function so that any additional library, called within the membrane, can be intercepted by the framework.

This operation does not normally cost any additional overhead since it is only done at system start-up and is therefore completely immaterial during server operations. If

[5]https://github.com/tvcutsem/harmony-reflect

```
function newMembrane (ifaceObj, policyObj) {
    return require("membrane")
        .makeGenericMembrane(ifaceObj, policyObj)
        .target;
}
```

Figure 9: We rely on a generic implementation, available via the "membrane" library, to wrap a membrane around a given ifaceObj with the given handler code in policyObj.

```
1  function safe_require (libName) {
2      var loadLib = function () {
3          var mod = new Module(libName);
4          var customRequire = membranedRequire(ctxt);
5
6          mod.require = function (libName) {
7              return customRequire(libName); };
8
9          return mod.loadLibrary();
10     };
11     return newMembrane(loadLib().exports, policy);
12 }
```

Figure 10: While loading a library with safe_require, the original require function is replaced with one that wraps the public interface object with a membrane and a given (Upper-Bound) policy.

```
1  function membranedRequire (lib) {
2      return function (lib) {
3          var libexports;
4
5          // [...] load the requested library
6          // and assign to libexports
7
8          if (lowerBoundPolicyNeeds(lib)) {
9              return newMembrane(libexports, policy);
10         } else {
11             return libexports;
12         }
13     }
14 }
```

Figure 11: In order to enforce a (Lower-Bound) policy between a library and a depending library, its interface object must be wrapped within a membrane.

```
// set to false for plain Node.js
var enable_nodesentry = true;

var st;
var http = require("http");

if (enable_nodesentry) {
        require("nodesentry");
        st = safe_require("st");
} else { st = require("st"); }

var handler = st(process.cwd());
http.createServer(handler).listen(1337);
```

Figure 12: The streamlined benchmark application implements a bare static file hosting server, by using the popular "st" and "http" libraries.

require is called dynamically we can still catch it. Either way, each time the function is called we can now test whether a library we want to protect has been invoked.

Line 11 of Figure 10 shows how the public interface object (exports), gets membraned with a given policy. This line makes it possible to enforce Upper-Bound policies.

Lower-Bound policies can be enforced because of the custom require function that is given to the context in which a library gets loaded (line 4 in Figure 10). Because we provide the context with our own require function, we can intercept all its calls from any depending library. At interception time, we can decide if it is necessary to membrane the public interface object of that depending library (line 8 of Figure 11). If decided so, all interactions between the library and its depending library are effectively subject to the Lower-Bound policy. If not, the original interface objects get returned (line 11 of Figure 11).

7. EVALUATION

Performance is king for server-side JavaScript and the main goal of our benchmarking experiment is to verify the impact of introducing NodeSentry on the two major performance drivers. We define performance as *throughput*, i.e., the amount of tasks or total requests handled by our server, or as *capacity*, i.e., the total amount of concurrent users/requests handled by our server. These are standard measures for high performance concurrent servers [19].

In order to streamline the benchmark and eliminate all possible confounding factors, we have written a stripped file hosting server that uses the "st" library to serve files requests. The *entire* code of the server, besides the libraries "http" and "st", is shown in Figure 12. The only conditional instruction present in the code makes it possible for us to run the benchmark test suite at first for pure Node.js and then with NodeSentry enabled.

Each experiment (for plain Node.js and for Node.js with NodeSentry enabled) consists of multiple runs. Each run measures the ability of the web server to *concurrently serve files to N clients*, for an increasingly large N, as illustrated in Figure 13. Each client continuously sends requests for files to the server throughout the duration of each experiment. At first only few clients are present (warm-up phase), after few seconds the number of clients step up and quickly reaches the total number N (ramp-up phase). The number of clients then remains constant until the end of the experiment (peak phase) with N clients continuously sending concurrent requests for files.

The experimental setup consists of two identical machines[6] interconnected in a switched gigabit Ethernet network. One machine is responsible for generating HTTP requests by spawning multiple threads, representing individual users. The second machine runs Node.js v0.10.28 and acts as the server.

[6]Each machine has 32 Intel© Xeon™ CPUs ES-2650 and 64GB RAM, running Ubuntu 12.04.4 LTS.

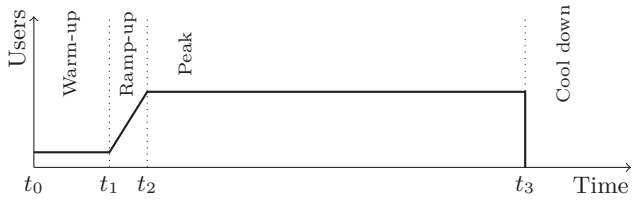


Figure 13: In our experimental set-up, the load profile of the experiment varies between a minimum (the warm-up phase) and a maximum (the peak phase) of concurrent users. This is repeated for $N = 1..1000$ concurrent users sending requests to our server.

The results of the experiment are summarized in Figure 14. The left graphics reports the throughput: how many requests the system is able to concurrently serve as the number of clients increases. This value is represented on the y-axis while the number of clients is represented on the x-axis. The diagonal black line plots the theoretical maximum: all requests by all clients are served in the given time horizon. Each blue square represent the summary of the performance of pure Node.js for the corresponding number of clients. The red circles denote the performance of NodeSentry for the same number of clients. The solid lines shows the interpolation curve with the *glm* method in R with a polynomial of grade 2. The gray shaded area represent the 95%-confidence interval computed by the function.

The right graphics reports the *capacity*: the number of concurrent requests handled at each time instance. The coding of lines and data follows the same criteria as for throughput: the blue squares and the blue line represents Node.js data points and interpolated values whereas the red line and the red circles represent the data points for NodeSentry.

For the first 200-250 all systems are able to serve requests at essentially the theoretical maximum capacity of the local benchmarking system. The system can comfortably host the intended amount of threads/concurrent users without slow-down. The results in Figure 14 indicate that NodeSentry's loss in capacity starts from around 200-250 concurrent users whereas the capacity of a plain Node.js instance starts to degrade at around 500 concurrent users.

NodeSentry gradually loses capacity until it stabilizes at approx 40% loss over the plain Node.js capacity and then moves in synchrony with NodeSentry after 500 users. It starts gaining again after approx 800-900 users and reduces the gap to 10%. Therefore, we can conclude that after 500 the losses of capacity are no longer due to NodeSentry but are directly consequence of the loss of capacity of Node.js. The sprint-up at 1000 clients can be easily explained: the main Node.js system is strained to keep up with performance, it has lost already 40% of its capacity over the theoretical maximum. In such stressful conditions, the additional constraints posed by NodeSentry's policy monitor are a drop in the sea.

We do not report data beyond 1000 users (albeit we tested it) because the behavior of *plain* Node.js started to exhibit significant jitters. It showed that largely beyond 1000 the actual capacity of our system set-up was dominated by other factors (OS process swaps, network processes, caches, etc. etc.). Setting up a benchmarking system that can smoothly process 10.000 users and beyond is an interesting direction for future work.

We've also measured the impact on the capacity of a server between using only policy hook (`"fs"` inside the membrane) and two policy hooks (`"fs"` outside the membrane).The results shown in Figure 15 indicate that there is no significant loss of capacity by bounding the semi-trusted library at the different policy points and thus tightening the policy rules.

At first, we stress again that *up to 200 clients there is no difference in performance*, which brings us almost at the same level of performance for an industrial security events monitoring system, suitable for deployment at a small business [23]. This strikingly compares with traditional approaches for JavaScript client side security in which even for *one* client there can be a performance penalty up to 300%.

For a larger number of clients there is a trade-off between performance and security. Such trade-off is still limited (less than 50% overhead) and decreases when other conditions stretch the performance of the system. Just as in normal program code, developers must take care to write efficient policy code. However, since policy code is written in plain JavaScript, it can benefit from efficiency measures in the underlying JavaScript engine, like e.g., a JIT compiler. Further, we believe that there are at least three ways to optimize the performance. The overhead is mostly due to the peculiarities of membranes: the overhead cost of the actual invariant enforcement mechanism, e.g., its use of a *shadow object*, the run-time post-condition assertions of the trap functions of the membrane handler, and the reliance on a self-hosted implementation of *Direct Proxies* in JavaScript [44, §5&6]. These would require a significant engineering effort that would not be justified for a research implementation.

8. RELATED WORK

There is a large body of work on JavaScript security, but the main focus has been overwhelmingly on client-side security. A very comprehensive survey of many of the recent works has been provided by Bielova [4] who describes a variety of JavaScript security policies and their enforcement mechanism within a web browser context. Therefore we refer to her work for additional details and only focus here on the few works that are closest to our own contribution.

JavaScript security in general.

Restricting third-party components within a web browser or web application by mediating access to specific security-sensitive operations, has seen a lot of attention since its rise the last decade.

BrowserShield [36] is a server-side rewriting system that rewrites certain JavaScript functions to use safe equivalents. These safe equivalents are injected into each web page via the BrowserShield JavaScript libraries. BrowserShield makes use of proxies for injecting its code into a web page. Self-protecting JavaScript [28, 35] is a client-side wrapping technique that applies an advice around JavaScript functions. The wrapping code with its advices are provided by the server and executed first, to make sure to operate in a clean, non-tampered environment. Browser-Enforced Embedded Policies (BEEP) [22] is a server system that injects a policy in a web page. The browser will call this policy script before loading any another script, giving the policy the opportunity

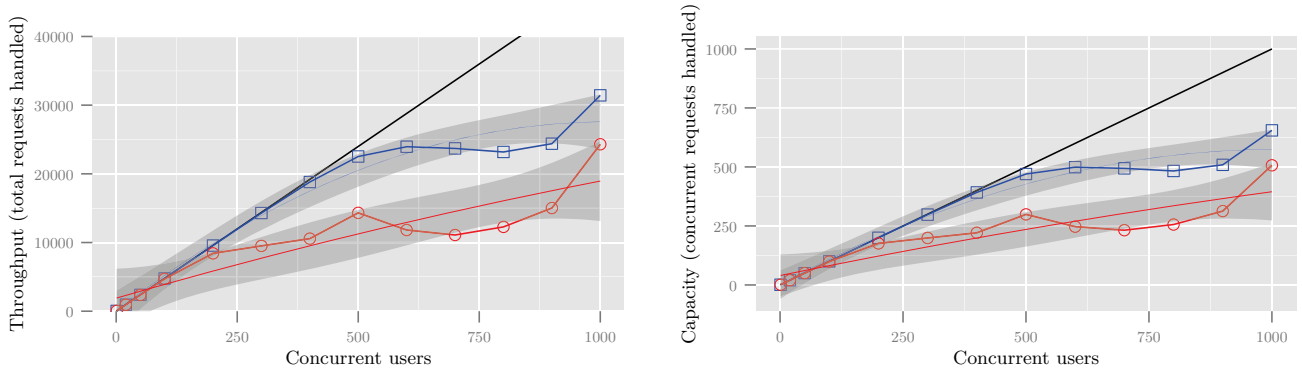


Figure 14: The solid black line is the theoretical performance of concurrent requests served in the fixed time horizon. The red circles represent the actual performance of plain Node.js with NodeSentry; the blue squares the performance of pure Node.js. Up to 200 clients the performance is optimal. Between 500-1000 we have a slight drop that is anyhow below 50%.

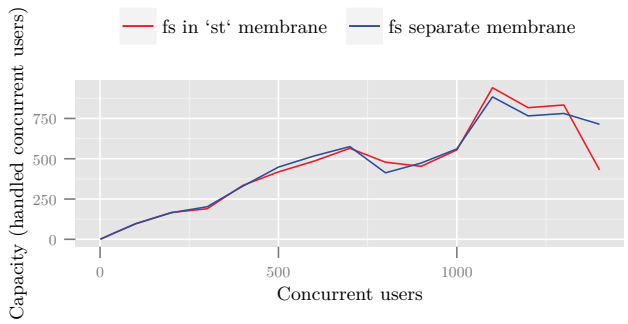


Figure 15: Tightening security by adding both an Upper-Bound policy and a Lower-Bound policy does not affect capacity, as demonstrated with the comparison of `"fs"` inside or outside the `"st"` membrane.

to vet the script about to be loaded. The loading process will only continue after approval of the policy. ConScript [29] allows the enforcement of fine-grained security policies for JavaScript in the browser. The approach is similar to self-protecting JavaScript [28, 35], except that ConScript uses deep advice, thus protects all access paths to a function. The price for using deep advice is the need for client-side support in the JavaScript engine. WebJail [42] offers the integrator the possibility to define a policy, in a specific, non-JavaScript way, that restricts the behavior of a third-party component in an isolated way. Agten et al., [2] present JSand, a server-driven sandboxing framework to enforce server-specified security policies in a client's browser. Richards et al. [37] present a security infrastructure for dealing with the gadget attacker threat model, by allowing the specification of access control policies on parts of a JavaScript program via leveraging the concept of delimited histories with revocation. Fredrikson et al., [15] have developed an off-line mechanism for the analysis of JavaScript applications that identify the place where policy hooks can be implemented by ConScript, thereby relying heavily on model checking technologies. Stefan et al. [40] introduce COWL, a JavaScript confinement system based on a new label-based mandatory access control API within web browsers that can be used by developers to indicate how to restrict the communication between compartments and external servers.

Security platforms for managed code.

Livshits [27] provides a taxonomy of runtime taint tracking approaches, in order to preventing web application vulnerabilities such as cross-site scripting and SQL injection attacks.

Wei et al., [45] propose a new architecture that decomposes a web service into two parts, executing in a separate protection domain. Only the trusted part can handle security-sensitive data.

Burket et al., [7] developed GuardRails, a source-to-source tool for building secure Ruby on Rails web applications, by attaching security policies, via annotations, to the data model itself. GuardRails produces a modified application that automatically enforces the specified policies.

Hosek et al., [21] developed a Ruby-based middleware that (1) associates security labels with data and (2) performs transparent label tracking, across a multi-tier web architecture in order to prevent harmful data disclosure.

Nguyen-Tuong et al., [32] propose a fully automated approach to harden PHP-based web applications via precise taint tracking of data and checking specifically for dangerous content only in parts of commands and output that came from untrustworthy sources.

Web application firewalls (WAF).

Krueger et al., [25] describe a technique, based on anomaly detectors, that replace suspicious parts in HTTP requests by benign data.

ModSecurity [1] is a firewall that detects malicious behavior by pattern matching HTTP requests with an existent rule base. A similar proxy-based approach has been proposed by Braun et al., [6] who implemented a policy enforcement mechanism to guarantee the control flow integrity of web applications.

9. CONCLUSIONS

Among the various server-side frameworks, Node.js has emerged as one of the most popular. Its strengths are the use of JavaScript, an efficient runtime tailored for cloud-based event parallelism, and thousands of third-party libraries.

Yet, these very libraries are also a source of potential security threats. Since the server runs with full privileges, a vulnerability in one library can compromise one's entire server. This is indeed what recently happened with the `"st"` library used by the popular web server libraries to serve static files.

In order to address the problem of least privilege integration of third party libraries we have developed NodeSentry, a novel server-side JavaScript security architecture that supports such least-privilege integration of libraries.

We have illustrated how our enforcement infrastructure can support a simple and uniform implementation of security rules, starting from traditional web-hardening techniques to custom security policies on interactions between libraries and their environment, including any dependent library. We have described the key features of the implementation of NodeSentry which builds on the implementation of membranes by Miller and Van Cutsem as a stepping stone for building trustworthy object proxies [44].

In order to show the practical effectiveness of NodeSentry we have evaluated its performance in an experiment where a server must be able to provide files concurrently to an increasing number of clients up to thousands of clients and tens of thousands of file requests. Our evaluation shows that for up to 250 clients NodeSentry has the same server capacity and throughput of plain Node.js, and that such capacity is essentially the theoretical optimum. At 1000 concurrent clients in a handful of seconds, when plain Node.js's already dropped capacity barely above 60% of the theoretical optimum, NodeSentry is able to attests itself at 50%.

Our complete prototype implementation (including the full source code, test suites, code documentation, installation/usage instructions, and the `"st"` example) is available at https://distrinet.cs.kuleuven.be/software/NodeSentry/ or directly installable via `npm install nodesentry`.

Acknowledgments

We thank our shepherd Joshua Schiffman and the anonymous reviewers for their valuable feedback. This research is partially funded by the Research Fund KU Leuven, the EU-funded FP7 projects NESSoS and WebSand, the IWT-SBO project SPION and the Italian Project PRIN-MIUR-TENACE. With the financial support from the Prevention of and Fight against Crime Programme of the European Union European Commission – Directorate-General Home Affairs. This publication reflects the views only of the authors, and the funders cannot be held responsible for any use which may be made of the information contained therein.

10. REFERENCES

[1] Modsecurity – the open source web application firewall. https://www.modsecurity.org/.

[2] P. Agten, S. Van Acker, Y. Brondsema, P. H. Phung, L. Desmet, and F. Piessens. JSand: Complete Client-Side Sandboxing of Third-Party JavaScript without Browser Modifications. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, pages 1–10, 2012.

[3] A. Barth. RFC 6265: HTTP State Management Mechanism. http://tools.ietf.org/html/rfc6265, 2011.

[4] N. Bielova. Survey on JavaScript Security Policies and their Enforcement Mechanisms in a Web Browser. *Journal of Logic and Algebraic Programming*, 2012.

[5] N. Bielova, D. Devriese, F. Massacci, and F. Piessens. Reactive non-interference for a browser model. In *Proceedings of the International Conference on Network and System Security (NSS)*, pages 97–104, 2011.

[6] B. Braun, P. Gemein, H. P. Reiser, and J. Posegga. Control-flow integrity in web applications. In *Engineering Secure Software and Systems (ESSOS'13)*, pages 1–16. Springer, 2013.

[7] J. Burket, P. Mutchler, M. Weaver, M. Zaveri, and D. Evans. GuardRails: A Data-Centric Web Application Security Framework. In *Proceedings of the USENIX Conference on Web Application Development (WebApps)*, 2011.

[8] F. Chong and G. Carraro. Architecture strategies for catching the long tail. Technical report, Microsoft Corporation, April 2006. Available on the web at http://msdn.microsoft.com/en-us/library/aa479069.asp.

[9] W. De Groef, D. Devriese, N. Nikiforakis, and F. Piessens. FlowFox: a Web Browser with Flexible and Precise Information Flow Control. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, pages 748–759, 2012.

[10] W. De Groef, D. Devriese, N. Nikiforakis, and F. Piessens. Secure Multi-Execution of Web Scripts: Theory and Practice. *Journal of Computer Security*, 22(4):469–509, 2014.

[11] L. Desmet, W. Joosen, F. Massacci, P. Philippaerts, F. Piessens, I. Siahaan, and D. Vanoverberghe. Security-by-contract on the. net platform. *Information Security Technical Report*, 13(1):25–32, 2008.

[12] D. Devriese and F. Piessens. Noninterference Through Secure Multi-Execution. In *Proceedings of the IEEE Symposium on Security and Privacy (SP)*, pages 109–124, 2010.

[13] U. Erlingsson. *The inlined reference monitor approach to security policy enforcement*. PhD thesis, Cornell University, 2003.

[14] D. Flanagan. *JavaScript: the definitive guide*. " O'Reilly Media, Inc.", 2002.

[15] M. Fredrikson, R. Joiner, S. Jha, T. Reps, S. Hassen, and V. Yegneswaran. Efficient Runtime Policy Enforcement Using Counterexample-Guided Abstraction Refinement. In *Proceedings of the International Conference on Computer Aided Verification (CAV)*, 2012.

[16] P. Gardner, S. Maffeis, and G. Smith. Towards A Program Logic for JavaScript. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, January 2012.

[17] L. Griffin, B. Butler, E. de Leastar, B. Jennings, and D. Botvich. On the Performance of Access Control Policy Evaluation. In *Proceedings of the IEEE International Symposium on Policies for Distributed Systems and Networks (POLICY)*, pages 25–32, 2012.

[18] A. Guha, C. Saftoiu, and S. Krishnamurthi. The Essence of JavaScript. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, pages 126–150, 2010.

[19] N. J. Gunther. *Guerrilla capacity planning – a tactical approach to planning for highly scalable applications and services.* Springer, 2007.

[20] J. Hodges, C. Jackson, and A. Barth. Rfc 6797: Http strict transport security (hsts). http://tools.ietf.org/html/rfc6797, 2012.

[21] P. Hosek, M. Migliavacca, I. Papagiannis, D. M. Eyers, D. Evans, B. Shand, J. Bacon, and P. Pietzuch. SafeWeb: A Middleware for Securing Ruby-based Web Applications. In *Proceedings of the International Middleware Conference*, pages 480–499, 2011.

[22] T. Jim, N. Swamy, and M. Hicks. Defeating script injection attacks with browser-enforced embedded policies. In *Proceedings of the International World Wide Web Conference (WWW)*, pages 601–610, May 2007.

[23] K. M. Kavanagh, M. Nicolett, and O. Rochford. Magic Quadrant for Security Information and Event Management. http://www.gartner.com/technology/reprints.do?id=1-1W1N1U4&ct=140627, June 2014.

[24] P. B. Kruchten. Architectural Blueprints – The "4+1" View Model of Software Architecture. *Journal of IEEE Software*, 12(6):42–50, 1995.

[25] T. Krueger, C. Gehl, K. Rieck, and P. Laskov. TokDoc: A Self-Healing Web Application Firewall. In *Proceedings of the 2010 ACM Symposium on Applied Computing (SAC)*, pages 1846–1853, 2010.

[26] S. Lekies, B. Stock, and M. Johns. 25 Million Flows Later – Large-scale Detection of DOM-based XSS. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2013.

[27] B. Livshits. Dynamic Taint Tracking in Managed Runtimes. Technical Report MSR-TR-2012-114, Microsoft Research, 2012.

[28] J. Magazinius, A. Askarov, and A. Sabelfeld. A Lattice-based Approach to Mashup Security. In *Proceedings of the ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, pages 15–23, 2010.

[29] L. Meyerovich, A. Felt, and M. Miller. Object views: Fine-grained sharing in browsers. In *Proceedings of the 19th international conference on World wide web*, pages 721–730. ACM, 2010.

[30] L. A. Meyerovich and B. Livshits. ConScript: Specifying and Enforcing Fine-Grained Security Policies for JavaScript in the Browser. In *Proceedings of the IEEE Symposium on Security and Privacy (SP)*, pages 481–496, 2010.

[31] M. S. Miller. *Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control.* PhD thesis, Johns Hopkins University, Baltimore, Maryland, USA, May 2006.

[32] A. Nguyen-Tuong, S. Guarnieri, D. Greene, J. Shirley, and D. Evans. Automatically hardening web applications using precise tainting. In *Proceedings of the IFIP International Information Security Conference*, pages 372–382, 2005.

[33] N. Nikiforakis, L. Invernizzi, A. Kapravelos, S. Van Acker, W. Joosen, C. Kruegel, F. Piessens, and G. Vigna. You Are What You Include: Large-scale Evaluation of Remote JavaScript Inclusions. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, pages 736–747, 2012.

[34] A. Ojamaa and K. Düüna. Assessing the Security of Node.js Platform. In *Proceedings of the International Conference for Internet Technology and Secured Transactions (ICITST)*, pages 348–355, 2012.

[35] P. H. Phung, D. Sands, and A. Chudnov. Lightweight Self-Protecting JavaScript. In *Proceedings of the ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, pages 47–60, 2009.

[36] C. Reis, J. Dunagan, H. J. Wang, O. Dubrovsky, and S. Esmeir. BrowserShield: Vulnerability-driven filtering of dynamic HTML. *ACM Transactions on the Web (TWEB)*, 1(11), September 2007.

[37] G. Richards, C. Hammer, F. Z. Nardelli, S. Jagannathan, and J. Vitek. Flexible Access Control for JavaScript. In *Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages & Applications (OOPSLA)*, 2013.

[38] F. B. Schneider. Enforceable Security Policies. *ACM Transactions on Information and System Security (TISSEC)*, 3(1):30–50, 2000.

[39] S. Stamm, S. Brandon, and G. Markham. Reining in the Web with Content Security Policy. In *Proceedings of the International Conference on World Wide Web (WWW)*, 2010.

[40] D. Stefan, E. Z. Yang, P. Marchenko, A. Russo, H. Dave, K. Brad, and D. Mazieres. Protecting Users by Confining JavaScript with COWL. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, 2014.

[41] S. Tilkov and S. Vinoski. Node.js: Using JavaScript to Build High-Performance Network Programs. *IEEE Internet Computing*, 14(6):80–83, 2010.

[42] S. Van Acker, P. De Ryck, L. Desmet, F. Piessens, and W. Joosen. WebJail: Least-privilege Integration of Third-party Components in Web Mashups. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, 2011.

[43] S. Van Acker, N. Nikiforakis, L. Desmet, F. Piessens, and W. Joosen. Monkey-in-the-browser: Malware and vulnerabilities in augmented browsing script markets. In *Proceedings of the ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, 2014.

[44] T. Van Cutsem and M. S. Miller. Trustworthy Proxies: Virtualizing Objects with Invariants. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, pages 154–178, 2013.

[45] J. Wei, L. Singaravelu, and C. Pu. A Secure Information Flow Architecture for Web Service Platforms. *IEEE Transactions on Services Computing*, 1(2):75–87, 2008.

TrueClick: Automatically Distinguishing Trick Banners from Genuine Download Links

Sevtap Duman
Northeastern University
Boston, MA
sevtap@ccs.neu.edu

Kaan Onarlioglu
Northeastern University
Boston, MA
onarliog@ccs.neu.edu

Ali Osman Ulusoy
Brown University
Providence, RI
ali_ulusoy@alumni.brown.edu

William Robertson
Northeastern University
Boston, MA
wkr@ccs.neu.edu

Engin Kirda
Northeastern University
Boston, MA
ek@ccs.neu.edu

Abstract

The ubiquity of Internet advertising has made it a popular target for attackers. One well-known instance of these attacks is the widespread use of *trick banners* that use social engineering techniques to lure victims into clicking on deceptive fake links, potentially leading to a malicious domain or malware. A recent and pervasive trend by attackers is to imitate the "download" or "play" buttons in popular file sharing sites (e.g., one-click hosters, video-streaming sites, bittorrent sites) in an attempt to trick users into clicking on these fake banners instead of the genuine link.

In this paper, we explore the problem of automatically assisting Internet users in detecting malicious trick banners and helping them identify the correct link. We present a set of features to characterize trick banners based on their visual properties such as image size, color, placement on the enclosing webpage, whether they contain animation effects, and whether they consistently appear with the same visual properties on consecutive loads of the same webpage. We have implemented a tool called TrueClick, which uses image processing and machine learning techniques to build a classifier based on these features to automatically detect the trick banners on a webpage. Our approach automatically classifies trick banners, and requires no manual effort to compile blacklists as current approaches do. Our experiments show that TrueClick results in a 3.55 factor improvement in correct link selection in the absence of other ad blocking software, and that it can detect trick banners missed by a popular ad detection tool, Adblock Plus.

1. INTRODUCTION

Internet advertising, both in traditional forms such as web banners and email campaigns, and more recent social media and mobile marketing schemes, is a rapidly growing, lucrative business [27]. Publishers and application developers find it increasingly easy to integrate advertising into their content and, consequently, attackers have leveraged this channel as an efficient mechanism for distributing malware. Naturally, the computer security community has shown an increasing interest in this field as well. Researchers have produced a large body of work to address critical security issues surrounding Internet advertising, such as sandboxing of advertisements from the actual content, resolving user privacy concerns, and preventing ad fraud.

While the security community has primarily focused on finding solutions to the technical side of the problems around Internet advertising so far, the *human factor* in Internet advertising and the class of attacks that attempt to exploit an Internet user's perception have largely been left unexplored. One well-known instance of such an attack is the widespread use of *trick banners* [32]. Trick banners are advertisement banners that are crafted to deceive and mislead users into clicking on them, potentially linking to a malicious domain or a malware executable. While trick banners have traditionally come in the form of colorful and animated messages, or as pop-ups imitating application messages, a more recent and pervasive trend is to imitate the "download" or "play" buttons in popular file sharing sites (e.g., one-click hosters, video-streaming sites, bittorrent sites) in an attempt to trick users into clicking on these fake banners instead of the genuine download link.

Previous work has explored user behavior and security awareness when browsing the Internet, and shown that trick banners found in file sharing sites are effective at tricking even technically sophisticated users who had previous familiarity with file sharing sites used in the study [37]. This study concluded that trick banners pose a significant security risk to ordinary Internet users, and even to those with a heightened security awareness. Indeed, abuse of advertisement banners in this way (a practice that is also referred to as "malvertising") has been recognized as a current and effective attack vector [42, 45]. It has also been shown that rather than using complex exploitation techniques, simply buying ad space is an easy and effective way for attackers to spread malware and quickly victimize a large number of Internet users [18]. Numerous attacks on high-profile websites and ad networks in recent years demonstrate that the problem is real and is actively being exploited [36, 50].

In this paper, we explore the problem of automatically

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.
ACSAC '14, December 08 – 12 2014, New Orleans, LA, USA
Copyright 2014 ACM 978-1-4503-3005-3/14/12 ...$15.00
http://dx.doi.org/10.1145/2664243.2664279

assisting Internet users in successfully detecting malicious trick banners, focusing on distinguishing fake download buttons ubiquitously found in popular file sharing websites from genuine download links. We first identify a set of features to characterize trick banners based on their visual properties such as banner size, color, placement on the webpage, whether they contain animation effects, and whether they consistently appear with the same visual properties on multiple copies of the same webpage obtained with separate requests. We then leverage these features in an approach combining image processing and machine learning to automatically detect trick banners on a webpage.

We implement our system in a prototype Firefox browser extension called TrueClick, evaluate its effectiveness on a data set of 259 banners collected from 88 file sharing websites, and demonstrate that TrueClick achieves a 96.97% true positive rate given a false positive rate of 3.03%. Note that unlike state-of-the-art ad blocking, our approach does not require a priori blacklisting of advertising domains, or any other manual classification of known banners. After an initial training phase, it operates in a completely automated manner by analyzing the visual properties of banners. In other words, the approach we propose is complementary to existing blacklisting approaches, and can support them in identifying previously unknown trick banners. Moreover, TrueClick does not rely on examination of the source code of webpages or the structure of the DOM tree. Instead, it utilizes image processing techniques to capture and analyze webpages *as the user sees them*, and therefore is not affected by attempts to thwart detection through dynamic modifications to the page.

In summary, we make the following contributions in this work.

- We present five visual features that can be used to characterize trick banners and experimentally demonstrate that these features can be used in practice to distinguish trick banners from genuine download links.
- We present a novel methodology for automatically and reliably distinguishing trick banners from genuine download links by using a combination of image processing and machine learning.
- We describe a prototype implementation of our solution as a Firefox browser extension called TrueClick that can effectively guide users toward finding and clicking on the genuine download link in a file sharing website littered with trick banners.
- We evaluate the usability of TrueClick with a user-study, and demonstrate a 3.55 factor improvement in selection of benign links in the presence of trick banners. We also show that TrueClick is able to detect trick banners missed by a popular ad detection tool, Adblock Plus.

2. PROBLEM STATEMENT

A trick banner is generally defined as any Internet advertising banner with a deceptive visual appearance, crafted to lure users into clicking on them [32]. They often do not contain any indication of the identity of the advertiser or the advertised service or product. Trick banners are known to integrate well with the look and feel of the website they appear on, and often imitate popular applications, operating system windows, and pop-up messages.

In this work, we focus on a specific pervasive class of trick banners: fake download buttons found on file sharing websites, which have recently been shown to be effective at tricking even users with security expertise [37]. Figure 1 shows various examples obtained from popular file sharing services. Note that the techniques we present in this paper are sufficiently generic to be applied to other kinds of trick banners as well. We believe that fake download buttons represent an up-to-date manifestation of the more general trick banner problem, and pose a challenging research task because of their tight integration with the file sharing sites they are displayed on. To illustrate, one of the sample trick banners taken from The Pirate Bay (Figure 1, lower left corner) displays identical replicas of the correct download links (i.e., the links "GET THIS TORRENT" and "ANONYMOUS DOWNLOAD"), making it an especially difficult trick banner to spot for an unsuspecting user.

Also note that banner design (and user interface design in general) for attracting users and maximizing click-through rates has been extensively studied in computer science and other, non-technical fields [5, 8, 10]. Our use of the term *trick banner* in this paper could refer to images with either benign or malicious intent. In other words, trick banners we examine could be crafted with malicious intent, such as directing a user to an attack site or downloading malware to her computer. Or, they could simply be used as a device for artificially inflating the number of visitors to a destination website, without explicit malicious intent. In this work, we do not aim to verify whether the trick banners lead to malicious destinations (other recent work has explored this aspect of the problem [30]), nor do we visit the sites or download the files they link to. Instead, our goal is to detect trick banners regardless of their purpose, and distinguish them from legitimate, genuine download links.

We divide the threat model we consider for this work into two scenarios. In the first scenario, an Internet user visits a file sharing website with the intention to download a specific file. We do not make any assumptions about the security expertise or awareness of the user. The pages on the website can contain any number of trick banners and regular advertisements, and one or more correct download buttons defined as the link to the content the user intends and expects to download. In cases where the website requires the user to step through a number of different pages to complete the download, we take the links that lead the user closer toward the final download link as the correct one.

The second scenario is identical to the first except that the website in question does not actually contain any correct download links. Such sites could be found on the Internet, for example, as part of a well-known scam scheme in which a scammer creates webpages that contain detailed descriptions of various content despite not including any actual download links, possibly in an attempt to trick search engines and steal clicks from users for ad revenue.

In both of these cases, regardless of whether a correct link exists or not, our system analyzes the webpage and determines the page regions that contain trick banners. If trick banners are detected, they could be marked with warning cues to alert the user or blocked entirely, depending on implementation choices or user preferences.

Finally, we would like to point out that there exist other types of attacks involving malicious modifications to websites and browser user interfaces that aim to make users in-

Figure 1: Four trick banner examples taken from popular file sharing sites The Pirate Bay, Rapidgator, and FilesTube that imitate the look and feel of genuine download buttons to deceive users.

advertently click on incorrect or malicious links. For example, various forms of clickjacking attacks compromise the visual and temporal integrity of the pointer cursor or browser's display to this end [24]. The problem we aim to address in this work is separate from those attacks in that, in our threat model, the system's integrity is not compromised; instead, trick banners exploit weaknesses in human perception to trick users. In clickjacking attacks, the user's system is technically crippled, making her unable to click on the correct link even if she can identify it. However, in the attacks we aim to address, the user *willingly* clicks on a trick banner, thinking that it is a genuine download link. While defenses against both types of attacks are necessary for secure browsing, we only explore the latter problem in this work.

3. BACKGROUND & RELATED WORK

Before we explain our approach to trick banner detection, we present related work and the state-of-the-art in this field in order to highlight the differences between our work and current advertisement detection systems.

Internet Advertising and Trick Banners. There is a large body of prior work on Internet advertising and the technologies around it, both in computer science and other non-technical fields. The effectiveness of various types of ad banners and ways to influence user click-through behavior have been studied widely in marketing, finance, psychology, and related fields [8, 12]. In a recent work, Onarlioglu et al. [37] presented a study that investigates the computer security implications of trick banners, and showed that trick banners can mislead even technically sophisticated Internet users and expose them to attacks.

Advertisement Blocking and Filtering. A simple method of blocking trick banners, and advertisements in general, is to disable prerequisites for displaying such content in web browsers. This can involve disabling image loading, blocking Flash Player and similar browser plugins, or blocking JavaScript code used by advertising networks through widely available browser security extensions [25]. Similarly, Web proxies could be deployed to filter out trick banners before they could be displayed in the browser [3, 4]. While these solutions are effective at blocking trick banners, they also significantly impair the user's browsing experience, or even render many websites nonfunctional, as the World Wide Web today makes extensive use of multimedia and dynamic content. Moreover, deploying HTTP proxies might not be accessible to the average Internet user, or might not be a viable option on more restricted mobile devices.

An alternative approach to the problem is using specialized ad filtering software that is often included as part of commercial antivirus suites, or designed as open-source web browser extensions such as the popular Adblock Plus [1]. On the one hand, these solutions offer the ability to selectively block offending content and therefore provide improved usability over the previously discussed solutions. On the other hand, they typically detect links to advertisements by consulting various blacklists and whitelists that must be continuously maintained and updated, which often involves significant manual labor. TrueClick instead performs trick banner detection based on the visual characteristics of such content in a completely automated manner and, thus, is able to detect trick banners that have not previously been classified into blacklists or whitelists.

Security Uses of Visual Features. Another body of work applies image comparison techniques and visual similarity metrics to different Internet security problems. For instance, various groups have investigated phishing site detection techniques based on visually comparing suspected phishing pages to their legitimate counterparts [9, 33, 34, 49], and Gargiulo and Sansone [13] use image processing to extract visual and text features from spam emails. In contrast, we identify visual features specifically tailored to detect trick banners and implement them in TrueClick.

Doppelganger [43] explores the results of different cookie policies on websites by transparently mirroring the user's web session to create two conceptual browser windows, one with cookies enabled and one without. They then compare the two to investigate the impact of accepting cookies from a given website on the content displayed. Likewise, TrueClick utilizes a detection feature based on comparing multiple views of a webpage, as we explain in Section 5.1.

Securely Isolating Ads and Applications. A large number of studies aim to protect sensitive web content from potentially malicious third-party ads by sandboxing the ads displayed on the page. Several projects use language containment and static policy enforcement to restrict JavaScript features used by ad networks [2, 11, 19, 23, 31], while others perform dynamic policy enforcement [15, 35, 46]. Likewise, researchers have investigated the ad ecosystem on mobile devices and proposed approaches that aim to isolate third-party ad libraries from mobile applications [16, 29, 38, 44]. Other approaches to create general secure mashups of advertisements and applications include secure browser architectures and browser-based operating systems [17, 39, 48].

In another set of studies, researchers explore the privacy issues around Internet advertising, and propose techniques to deliver targeted ads while limiting the exposure of privacy-sensitive user information to ad networks [20, 21, 28, 40, 47].

These studies have the common goal of protecting applica-

Figure 2: The image extraction process illustrated on a real trick banner. Step 1 detects four superfluous sub-regions on the banner, Step 2 then corrects this error, and matches the fragments to the actual image.

tions against uncontrolled and potentially malicious ad code. While the isolation they provide is clearly essential for secure deployment of online ads, the measures they employ do not provide protection in a scenario in which Internet users are deceived and misled to click on a trick banner that links to a malicious destination. In contrast, TrueClick addresses this problem by analyzing the ad banners on a web page and guiding users to identify the genuine download link among a set of trick banners.

4. IMAGE EXTRACTION

Before we can compute any features on potential trick banner images, we first need to identify and extract all of the image regions on a webpage, which are subsequently fed to our trick banner classification system. Note that simply searching for HTML image tags in the page source is not sufficient to perform this task correctly, because some of the banners may be loaded dynamically by JavaScript, or they may come in non-image formats like Flash files or regular links stylized to look like buttons. In this section, we briefly explain the details of this process.

The image extraction technique we propose in this work is a two-step process. Initially, we leverage well-known image processing techniques designed for this task, which follow the common pipeline of edge detection, region filling and connected component analysis [14], and banner region identification on the webpage. However, the enormous variability in webpage content often does not allow a single generic image processing pipeline to perform perfectly in all cases, and our early experiments indicate that extracting image regions solely through image processing usually falls short. For example, when faced with a trick banner that displays several button lookalikes in a single image file, the aforementioned image processing pipeline yields multiple detection results for each disconnected component in the image. (See the banner presented in Figure 2 for a real example.) Similarly, more sophisticated visual designs could result in a large number of superfluous detections of small sub-regions on a single image (or, conversely, missed portions of banners), which would later result in unnecessary detection feature computations or inaccurate results. Although image processing methods that are customized for each specific website can be devised to improve extraction accuracy and performance, such an approach would be time and effort-intensive, without any clear indication as to how the extraction scheme could be adapted to the future banner designs.

Instead, based on the observation that while the discussed image processing techniques are imperfect, they rarely completely miss entire banner regions, we employ a second step to correct for partially detected and fragmented banners. This involves collecting and temporarily caching all of the actual image files requested from the web server for a given webpage to form a small image database. Next, the possibly fragmented banners extracted using the initial image processing step are matched to the images in the database. This matching is performed using the SURF (Speeded-Up Robust Features) feature detector and descriptor, which is widely used for object detection and recognition in the computer vision community [6]. Our experiments show that even when a significant portion of the banner is fragmented, SURF is able to match the banner to the correct image in the database. When a match is found, or in other words the extracted region is detected as a fragment of a larger image, the extracted banner image is simply replaced with the correct image from the database.

This process is illustrated in Figure 2. In Step 1, the image processing pipeline incorrectly identifies four separate regions in the banner. Later, in Step 2, these four extracted regions are all matched to the actual banner image stored in the database (and are replaced by it), resulting in an accurate banner extraction.

We must point out that even in the presence of the image database we build in Step 2 of this process, the banners extracted through image processing in Step 1 still provide valuable information for detecting trick banners; specifically, those that come in non-image formats. For example, in our experiments we observed static banners delivered in Flash files, or as regular links in HTML iframes stylized to look like buttons. While image processing can identify and extract such non-traditional banners, attempts to match them to image files in Step 2 would fail since there does not exist a corresponding image file on the webpage. Therefore, in cases where we cannot find any successful match in Step 2, we do not discard the regions extracted in Step 1 but instead input them to the classifier as-is.

5. TRICK BANNER CLASSIFICATION

Once possible trick banner regions have been identified on the webpage, five visual features are extracted to help classify each region as either a trick banner or a genuine download link. These features include *a*) image color, *b*) image size, *c*) image placement, *d*) presence of animation, and *e*) image differences between consecutive page loads. In the remainder of this section, we provide details on each of these features, explain why they are useful for distinguishing trick banners from genuine download links, and then present the classification approach we adopt in this work.

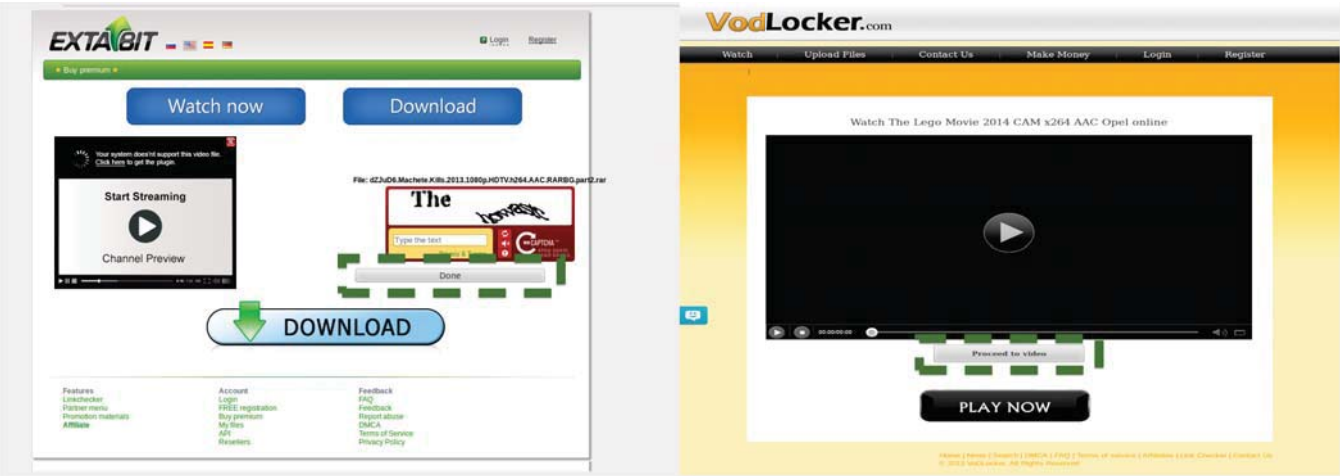


Figure 3: Sample webpages that illustrate that EMD scores vary with the color theme of the websites, and should only be compared after normalization. The correct banners are marked with dashed boxes.

5.1 Features

Color. Trick banners are often not designed by the site owners, and are usually served by third-party advertising networks just like regular ad banners. Consequently, the designers of trick banners do not know the exact website their banners are going to be displayed on, which leads them to follow common webpage theme specifications in their visual designs. As a result, trick banners often do not fit the general color theme of the website, but instead display distinctive color signatures. In contrast, genuine download buttons usually cohere to the overall website colors. This distinction suggests that banners can be classified based on their color similarity to that of the overall website.

This classification requires first a description of the colors inside the banner region. Experiments indicate both genuine and trick banners can be quite complex in terms of the color patterns they contain. Most banners are composed of a multitude of colors, highlights, and gradients, and also include other small images. Color histograms are ideal for the purposes of capturing the global color features of the banner. Histograms are constructed simply by binning the color of each pixel in the banner region. The histograms are finally normalized by their total mass such that they are invariant of the number of pixels in the banner region.

Classification based on color histograms also requires a method to compute the similarity between histogram pairs. In our work, we compute color histogram similarity using the Earth Mover's Distance (EMD) [41], which is widely used in content-based image retrieval applications. In short, EMD is a metric between two distributions (of equal total mass) that measures the minimal cost incurred to transform one distribution to the other. The comparison is made between each banner and the whole page histograms. We made our computations on the RGB channel.

Note that the EMD score of a banner is computed with respect to the color histogram of the overall webpage and, thus, this score provides a measure *relative* to the other banners within the webpage. For instance, on the left-side webpage in Figure 3, the EMD score of the correct banner (marked with dashed lines) is 17443.67, and the trick banners on the same webpage have higher scores than the correct banner. However, we cannot generalize this outcome and consider scores higher than 17443.67 obtained from different pages to indicate a trick banner. To illustrate, the webpage on the right side of Figure 3 has a correct banner that has an EMD score of 19773.83, and while this value is still lower than the EMD scores of the trick banners on the same webpage, it is higher than 17443.67. Therefore, a normalization is required so that the EMD scores of all banners in training and classification are comparable. Mapping the EMD scores of banners in each website to a fixed interval such as $[0, 1]$ is sufficient for this purpose.

Size. Websites often reserve fixed sections in their visual layouts where advertisements can be displayed. The sizes of these reserved spaces are not strictly controlled. However, in accordance with the Interactive Advertising Bureau online advertisement size guidelines [26], these sections are usually large, horizontal, or vertical rectangular regions in which standard web banners that advertising networks serve can fit. In contrast, genuine links on a file sharing website usually take the form of a single, relatively small image that serves as a download button. Consequently, in order not to throw off their disguise by displaying unusually large fake buttons that contrast with the rest of the page's style, many trick banners resort to tricks such as using large empty borders around a smaller fake button image, or including two or more button images on a single banner as if they were separate clickable entities. Thus, image size is a strong feature for distinguishing trick banners from genuine buttons.

We measure the size of these images in the x and y dimensions in the number of pixels. In order to deal with varying webpage sizes, we first normalize the numbers to $[0, 1]$ with respect to the absolute size of the enclosing webpage.

Placement. Navigation links on a website, including the genuine download buttons, are tightly integrated with the rest of the site's content. In contrast, advertisement banners are often laid out separately in reserved spaces in order not to interfere with the coherence of the website's interface and content. They are usually placed at the page header, footer, sidebars, or are otherwise isolated from the actual page content. Therefore, the position of the banner can be used as an indicator for trick banners. We use the x and y

positions of the geometric center of the banner as the placement feature. Similarly to the size feature, we first normalize the values to $[0, 1]$.

Animation. Another significant indicator for trick banners is the use of animations that are employed to draw the attention of the user. Animated banners rapidly display a sequence of images, typically in the form of GIF images. Most genuine download links do not contain animations. In fact, during our study we didn't encounter any genuine links that contained animations.

Animation is a binary feature, indicating whether or not the banner is animated. The presence of animation is detected by first checking whether the banner image format allows animations. If it does, the number of frames embedded in the file is used to decide the presence of animation. This animation check is performed only on the images that were selected from the database because it is not possible to reach fragmented image information unless it matches with a cached image.

Visual Differences in Multiple Page Views. A common method for deploying advertisements on a page for a website owner is to utilize advertising networks, which serve a different ad banner every time a user visits the page. Similarly, large content publishers may use their own advertising infrastructures that rotate the banners displayed on the page each time the page is visited. Consequently, the visual contents of banner spaces on webpages tend to be very dynamic, often changing every time the page is loaded or refreshed in the browser. In contrast, the user interfaces of webpages are often comprised of a fixed set of images that seldom change once the website's design has been finalized. To promote usability and provide a consistent user experience, menus, navigation links, and buttons on the page are placed at specific positions and use static images.

We take advantage of the dynamic characteristics of banners and the static nature of the rest of the UI elements on a webpage to propose a trick banner detection feature based on comparing multiple views of a single webpage. Specifically, we first take two screen captures of the same page obtained through two separate requests to the web server. We then visually compare them, extract the parts that have changed between the requests, and mark those as potential banner regions.

5.2 Classifier

A binary classifier is a function that takes as input a set of features, such as the visual features described above, and outputs a binary decision – in this case, *trick banner* or *non-trick banner*. The machine learning literature offers a wide variety of binary classifiers [22]. In this paper, we choose to use the popular random forest classifier, and train it using the method proposed by Breiman [7]. The details of our training data collection methodology is explained in Section 7. Note that although the random forest is used for all experiments in this paper, we have observed that the results are comparable using other state-of-the-art classifiers such as the support vector machine.

6. IMPLEMENTATION

We implemented the trick banner detection methodology we have discussed above in a prototype system called TrueClick, as a Firefox browser extension that uses external image processing libraries. In this section, we explain the implementation-specific details of our system.

6.1 Overview

TrueClick is implemented as a browser extension that runs on demand when the user visits a file sharing website containing trick banners and clicks on a button to activate the system. Once the analysis of the banner images is complete, TrueClick can either mark the detected trick banners as such, or block them entirely. In this prototype implementation, we elected to visually obscure the trick banners from the user.

An overview of the architecture of TrueClick is presented in Figure 4. When the user triggers the analysis on a given web page, the *Screen Grabber* and *Image Grabber* components first take screenshots of the webpage and cache all image files downloaded from the web server in an image database, respectively. Then, the screenshots and image database are input to the *Image Extractor* component, which identifies the banner regions on the screenshots, and attempts to match them to the files in the database, as previously explained in Section 4. Once all banner regions are detected, they are sent to the *Feature Extractor* which computes the five trick banner detection features on each image. Finally, TrueClick runs the resulting feature vectors through its classifier and determines the regions on the webpage where trick banners are displayed.

Note that during this process, all extracted images smaller than 16×16 (i.e., the standard size for favicon files) are discarded in order to speed up the detection process and display more relevant warnings to the user since they are most likely not banners.

In the following, we elaborate on the details of the Screen Grabber and Image Grabber components.

6.2 Screen Grabber

The Screen Grabber component is primarily responsible for taking a screenshot of the webpage as rendered by the browser. In order to ensure that the resulting screen capture is identical to what the user of the browser sees, TrueClick copies every pixel displayed in Firefox's main browsing window in a hidden HTML *canvas* internal to the browser, and dumps the results into an image file.

This component is also tasked with providing the necessary information for the Feature Extractor to identify the visual difference in multiple views of the same webpage. To this end, once TrueClick is activated, the Screen Grabber issues an additional HTTP request for the displayed webpage, renders it in a hidden browser window, and uses this to take a second screenshot of the page with (potentially) different banners. Note that during this process, care must be taken to ensure that the dimensions and display properties of the hidden window are identical to that of the original window so that the two screenshots obtained match and the subsequent comparison can be carried out accurately.

6.3 Image Grabber

The Image Grabber component identifies the image files referenced by the webpage, and builds a temporary image database to be used by the Image Extractor to match fragmented banners against. However, simply parsing the DOM tree of the webpage is not an effective way of accomplishing this task since many banners are displayed dynamically after page load by JavaScript ad libraries. Similarly, even

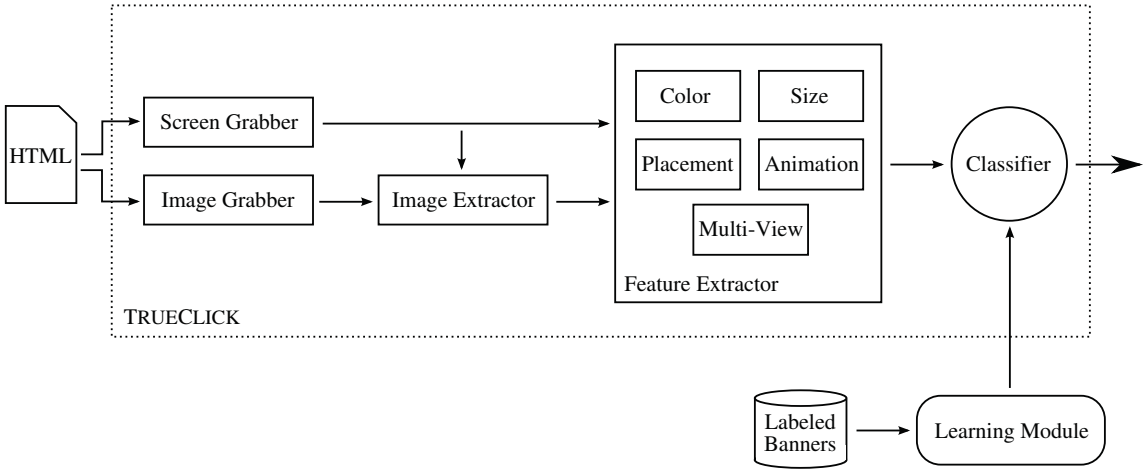


Figure 4: Overview of the architecture and various components of TrueClick.

when the URLs to the image files can be detected, downloading the banners from those locations is not a reliable way of obtaining the images, because the URLs provided by ad delivery frameworks often rotate between different banners and serve different image files with each request.

In order to address these problems, Image Grabber transparently intercepts HTTP responses from web servers and inspects the payload. Once it has been determined that the data corresponds to an image file, it is copied and inserted to the image database. This technique has the additional benefit of avoiding downloading the same image files a second time, saving bandwidth and allowing for quick detection of trick banners.

7. EVALUATION

In this section, we describe our experiments to measure the accuracy and usability of our solution in identifying trick banners in the wild. We evaluated the effectiveness of our classifier on a data set of banners we collected, conducted a user study to demonstrate that TrueClick is practical, and finally, compared the detection effectiveness to an existing ad detection system, AdBlock Plus.

7.1 Data Collection

To train and evaluate our system, we collected trick banner samples and images of genuine download buttons from popular file sharing websites, including one-click hosters, bittorrent sites, and online video streaming sites. Our training data set consists of only English websites, while the evaluation data set contains both English and non-English websites. We chose to perform the data collection and labeling procedure manually instead of crawling these websites, so that we could use cues from the pages to determine whether the collected samples are trick banners or not with high confidence and train our classifier with an accurate data set. We reached the actual file download pages by searching for popular movies and computer programs at the file sharing websites and services. In the banners we collected, we looked for keywords that could be intended to trick users such as *Download*, *Watch*, *Now*, *Save*, *Play*, *Get it*, and *Store*.

We note that such file sharing websites are not exclusively used for illegal media trading, but are also often used to distribute media to large audiences (e.g., software updates, non-commercial documentaries). Hence, our aim is to protect users in general, even though some might be engaged in illicit behavior when they are tricked by malicious banners.

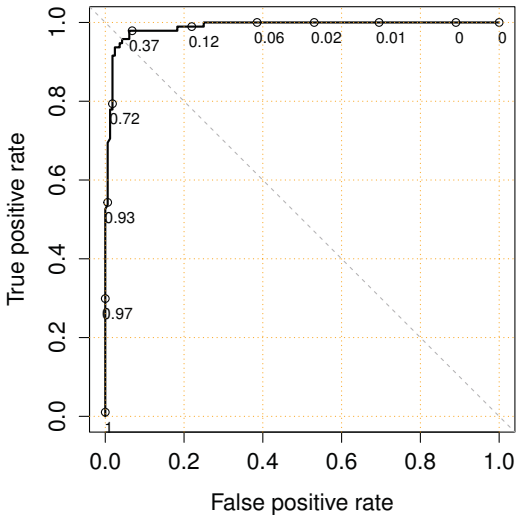


Figure 5: ROC curve for a 10-fold cross validation of TrueClick's random forest classifier.

To train and evaluate our classifier, we used 165 trick banners and 94 images corresponding to genuine download links for a total of 259 banner samples from 88 file sharing websites. For the comparison with AdBlock Plus, we used a disjoint set of 415 trick banners collected from 82 websites. In total, we collected 674 banner samples from 170 file sharing websites.

7.2 Evaluation of the Classifier

To build a classifier to distinguish between trick banners and genuine download links, we used the R statistical machine learning environment and, specifically, the ipred package, to train a random forest classifier from which the importance of predictors were assessed.

Figure 5 displays a ROC curve for a 10-fold cross validation of the resulting random forest classifier over our training

Feature	Importance (bits)
x-position	23.37
y-position	25.91
Size	100.06
Color	28.71
Animation	21.98
Multiview	52.14

Table 1: Information gain for each feature.

Experiment	Correct Clicks	Incorrect Clicks	Median Score	Avg. Score
Original page	29	91	0	0.725
w/ TrueClick	103	17	3	2.575

Table 2: Results of the TrueClick user study, showing the number of correct and incorrect clicks. The presence of incorrect clicks when using TrueClick is due to trick banners missed by our system.

set, using a cutoff value of 0.1. An n-fold cross validation partitions the entire data set into n equal-sized samples, or folds, trains on $n-1$ folds, and then validates the resulting model on the remaining fold. This process is repeated for each fold. The ROC curve plots the true positive rate against the false positive rate of the best-performing classifier as the discrimination threshold – i.e., the boundary between the trick banner and genuine link classes – is varied over $[0, 1]$. The value of the ROC curve lies in the guidance it provides in selecting thresholds to bias towards true positives at the expense of false positives, and vice versa.

The ROC curve shows that our classifier achieves a 96.97% true positive rate given a false positive rate of 3.03%, which is lower than the critical false discovery rate threshold of 5%. In other words, 3.03% of trick banners were incorrectly classified as correct banners. As shown in Table 1, the size feature is the most effective in distinguishing trick banners from legitimate download links. The table is generated using out-of-bag samples from the training data.

7.3 Effectiveness of Trick Banner Detection

We tested TrueClick's usability and effectiveness in guiding users to identify and click on the genuine download links on a file sharing website by conducting a user study, and comparing its detection performance to AdBlock Plus.

User Study. We performed our user study on 40 undergraduate and graduate computer science students. While we did not explicitly evaluate the participants' technical savviness, it is reasonable to expect them to be relatively expert computer and Internet users.

We first briefed all participants that they were going to take part in a user study on identifying the genuine download links on English-only file sharing websites, and then instructed them to click on the link or button they thought was legitimate on the webpages they were shown. Next, we presented each participant with unmodified pages from three different file sharing websites for them to perform this task on. In order to control for the fact that advertisement banners change every time a page is requested, we did not use the actual websites in our test, but instead created identical offline replicas with a fixed set of banners. Once the participants completed the first three tasks, we repeated the experiment using the same three websites, but this time in a browser window running TrueClick so that our system could analyze the page content and mark detected trick banners as such. We observed each participant complete all six tasks, recorded the number of correct and incorrect clicks for each of them, and assigned scores based on the number of correct clicks. The results are shown in Table 2.

These results demonstrate that in the experiments where the participants were assisted by TrueClick, there was a **3.55** factor improvement in the scores on average. We also checked these results for statistical significance using a standard paired difference test, namely, the Wilcoxon signed-rank test. The results of the test ($V = 0$, $p < 4.82 \times 10^{-7}$) confirm that the scores obtained with and without TrueClick indeed constitute non-identical populations.

Comparison to AdBlock Plus. Existing systems such as Adblock Plus are capable of identifying and blocking trick banners in some cases. To understand whether TrueClick is necessary given the existence of such systems, we compared its detection effectiveness with Adblock Plus.

We conducted experiments with 415 manually-identified trick banners from 82 websites that were not used in the training phase of the previous experiments. TrueClick correctly identified **380 (91.6%)** of these as trick banners using the previously-generated classifier without extra training or manual tuning, whereas Adblock detected 190 (45.8%). In contrast, there were only 8 banners detected by Adblock Plus, but not TrueClick.

We stress that these result do not suggest TrueClick is a substitute for Adblock Plus. TrueClick, as discussed and evaluated in this paper, focuses on detection of trick banners, and this experiment is not sufficient to draw conclusions on its ability to detect ordinary, benign advertisements. Therefore, we conclude that while Adblock Plus provides an efficient filter against general Internet advertising, supporting it with TrueClick significantly improves protection against potentially malicious trick banners.

8. DISCUSSION

The system we propose for automated trick banner detection has some limitations that we highlight here. First, we stress that in contrast to other work on trick banners and attacks against users, TrueClick is intended to address the specific case of images that masquerade as genuine download or play links on webpages. While the techniques we make use of here could be extended to cover other attacks, that is not the focus of this work.

As we noted in our evaluation of the random forest classifier TrueClick uses, the models we are able to build over our data set generate a non-trivial number of false positives. However, our models nevertheless classify the majority of trick banners correctly even when the threshold is set to produce a 0% false positive rate. Therefore, this represents a significant reduction in the number of trick banners that users must navigate, and our user experiments demonstrate that this translates to much better security decisions in practice when TrueClick is deployed.

We note that due to variations in content between consecutive page loads, we have observed our system to classify some *non-clickable* regions of the page as trick banners. However, due to the way in which we deploy our classifier

in the browser, this does not have generally have a deleterious effect on the user experience as the mislabeled content is non-interactive and obscuring it does not affect the functionality of the page.

Finally, an unlikely but interesting limitation of our system we discovered during our experiments involved a small number of image files that contained a corrupt GIF header. Although these files could successfully be displayed in a browser window, attempting to run our analysis on them caused the image processing library in our implementation to fail and abort the detection process. We only encountered three such images over the course of our experiments. After manual analysis of the files, we concluded that the images were likely created by a buggy image editor. Still, this observation demonstrates that purposely injecting errors inside image files could be used by trick banner creators as an evasion technique against automated analysis by TrueClick and similar tools, and highlights the importance of building an implementation with analysis routines robust against errors in image file headers.

9. CONCLUSION

In this work, we have highlighted the problem of trick banners that masquerade as benign links to download files or play videos by emulating the visual characteristics of the correct links. We have defined a number of visual features that distinguish trick banners from genuine links, including image size, color, placement on the enclosing webpage, whether they contain animation effects, and whether they consistently appear with the same visual properties on consecutive loads of the webpage. Using these features, we have built TrueClick, which uses image processing and machine learning to automatically detect trick banners. Our approach operates purely over visual features and, after an initial training, requires no further manual effort, for instance to compile blacklists as current approaches do.

We have evaluated our system over a data set of manually labeled trick banners and benign image links. Our experiments showed that our classifier achieves a 96.97% true positive rate given a false positive rate of 3.03%, showing that TrueClick can correctly detect the majority of the trick banners on file sharing websites with a reasonable low false positive rate. We tested our implementation of TrueClick on 40 users, and found that TrueClick resulted in a 3.55 factor improvement in correct link selection. We also demonstrate that TrueClick serves as an effective and useful complement to existing approaches for identifying trick banners such as Adblock Plus. We conclude that TrueClick successfully assists even technically-sophisticated users in correctly selecting benign image links despite the presence of malicious trick banners.

It remains an open question whether TrueClick could be supported with static analysis of webpages in order to further improve detection effectiveness, which is a promising direction for future research.

Acknowledgment

This work was supported by the Office of Naval Research (ONR) under grant N000141210165, and Secure Business Austria.

10. REFERENCES

[1] Adblock Plus. `https://adblockplus.org`.
[2] ADSafe. `http://www.adsafe.org`.
[3] Privoxy. `http://www.privoxy.org`.
[4] Squid. `http://www.squid-cache.org`.
[5] E. Adar, D. S. Tan, and J. Teevan. Benevolent Deception in Human Computer Interaction. In *CHI*, 2013.
[6] H. Bay, A. Ess, T. Tuytelaars, and L. V. Gool. Speeded-Up Robust Features (SURF). *Computer Vision and Image Understanding*, 110(3):346–359, June 2008.
[7] L. Breiman. Random forests. *Machine Learning*, 45(1):5–32, Oct. 2001.
[8] J. Chandon and M. S. Chtourou. Factors Affecting Click-Through Rate. In C. P. Haugtvedt, K. A. Machleit, and R. Yalch, editors, *Online Consumer Psychology: Understanding and Influencing Consumer Behavior in the Virtual World*, chapter 6. Psychology Press, Jan. 2005.
[9] T. Chen, S. Dick, and J. Miller. Detecting Visually Similar Web Pages: Application to Phishing Detection. *ACM TOIT*, 10(2):5:1–5:38, June 2010.
[10] G. Conti and E. Sobiesk. Malicious Interface Design: Exploiting the User. In *WWW*, 2010.
[11] M. Finifter, J. Weinberger, and A. Barth. Preventing Capability Leaks in Secure JavaScript Subsets. In *NDSS*, 2010.
[12] K. Gallagher and J. Parsons. A Framework for Targeting Banner Advertising on the Internet. In *HICSS*, 1997.
[13] F. Gargiulo and C. Sansone. Combining Visual and Textual Features for Filtering Spam Emails. In *ICPR*, 2008.
[14] R. C. Gonzalez and R. E. Woods. *Digital Image Processing*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 2001.
[15] Google. Google Caja. `https://developers.google.com/caja`.
[16] M. C. Grace, W. Zhou, X. Jiang, and A.-R. Sadeghi. Unsafe Exposure Analysis of Mobile In-App Advertisements. In *ACM WiSec*, 2012.
[17] C. Grier, S. Tang, and S. T. King. Secure Web Browsing with the OP Web Browser. In *IEEE S&P*, 2008.
[18] J. Grossman and M. Johansen. Million Browser Botnet. Black Hat USA, 2013.
[19] S. Guarnieri and B. Livshits. Gatekeeper: Mostly Static Enforcement of Security and Reliability Policies for Javascript Code. In *USENIX Security*, 2009.
[20] S. Guha, B. Cheng, and P. Francis. Privad: Practical Privacy in Online Advertising. In *USENIX NSDI*, 2011.
[21] H. Haddadi, P. Hui, and I. Brown. MobiAd: Private and Scalable Mobile Advertising. In *ACM MobiArch*, 2010.
[22] T. Hastie, R. Tibshirani, and J. Friedman. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. Springer Series in Statistics. Springer Science+Business Media, LLC, New York, NY, USA, 2nd edition, 2009.

[23] D. Hopwood. Jacaranda. http://www.jacaranda.org.

[24] L. Huang, A. Moshchuk, H. J. Wang, S. Schechter, and C. Jackson. Clickjacking: Attacks and defenses. In *USENIX Security*, 2012.

[25] InformAction. NoScript. http://noscript.net.

[26] Interactive Advertising Bureau. Ad Unit Guidelines. http://www.iab.net/standards/adunits.asp.

[27] Interactive Advertising Bureau. IAB Internet Advertising Revenue Report. 2012 Full Year Results. http://www.iab.net/media/file/IAB_Internet_Advertising_Revenue_Report_FY_2012_rev.pdf.

[28] A. Juels. Targeted Advertising ... And Privacy Too. In *Topics in Cryptology: CT-RSA*. Springer-Verlag, 2001.

[29] I. Leontiadis, C. Efstratiou, M. Picone, and C. Mascolo. Don't Kill My Ads!: Balancing Privacy in an Ad-supported Mobile Application Market. In *HotMobile*, 2012.

[30] Z. Li, K. Zhang, Y. Xie, F. Yu, and X. Wang. Knowing Your Enemy: Understanding and Detecting Malicious Web Advertising. In *ACM CCS*, 2012.

[31] S. Maffeis and A. Taly. Language-Based Isolation of Untrusted JavaScript. In *IEEE CSF*, 2009.

[32] Marketing Terms. Trick banner definition. http://www.marketingterms.com/dictionary/trick_banner/.

[33] M. Maurer and D. Herzner. Using Visual Website Similarity for Phishing Detection and Reporting. In *CHI Extended Abstracts*, 2012.

[34] E. Medvet, E. Kirda, and C. Kruegel. Visual-Similarity-Based Phishing Detection. In *SecureComm*, 2008.

[35] Microsoft. Microsoft Web Sandbox. http://www.websandbox.org.

[36] M. Mimoso. Malware Campaign Leverages Ad Networks, Sends Victims to Blackhole. http://threatpost.com/malware-campaign-leverages-ad-networks-sends-victims-to-blackhole, Sept. 2013.

[37] K. Onarlioglu, U. O. Yilmaz, E. Kirda, and D. Balzarotti. Insights into User Behavior in Dealing with Internet Attacks. In *NDSS*, 2012.

[38] P. Pearce, A. P. Felt, G. Nunez, and D. Wagner. AdDroid: Privilege Separation for Applications and Advertisers in Android. In *ASIACCS*, 2012.

[39] C. Reis and S. D. Gribble. Isolating Web Programs in Modern Browser Architectures. In *EuroSys*, 2009.

[40] A. Reznichenko, S. Guha, and P. Francis. Auctions in Do-Not-Track Compliant Internet Advertising. In *ACM CCS*, 2011.

[41] Y. Rubner, C. Tomasi, and L. J. Guibas. The Earth Mover's Distance as a Metric for Image Retrieval. 40(2):99–121, 2000.

[42] W. Salusky. Malvertising. http://isc.sans.edu/diary/Malvertising/3727, June 2007.

[43] U. Shankar and C. Karlof. Doppelganger: Better Browser Privacy without the Bother. In *ACM CCS*, 2006.

[44] S. Shekhar, M. Dietz, and D. S. Wallach. AdSplit: Separating Smartphone Advertising from Applications. In *USENIX Security*, 2012.

[45] Symantec. Malware Security Report: Protecting Your Business, Customers, and the Bottom Line. www.verisign.com/verisigntransition101/files/MalwareSecurityReport.pdf, 2011.

[46] M. Ter Louw, K. T. Ganesh, and V. N. Venkatakrishnan. AdJail: Practical Enforcement of Confidentiality and Integrity Policies on Web Advertisements. In *USENIX Security*, 2010.

[47] V. Toubiana, A. Narayanan, D. Boneh, H. Nissenbaum, and S. Barocas. Adnostic: Privacy Preserving Targeted Advertising. In *NDSS*, 2010.

[48] H. J. Wang, C. Grier, A. Moshchuk, S. T. King, P. Choudhury, and H. Venter. The Multi-Principal OS Construction of the Gazelle Web Browser. In *USENIX Security*, 2009.

[49] L. Wenyin, G. Huang, L. Xiaoyue, Z. Min, and X. Deng. Detection of Phishing Webpages Based on Visual Similarity. In *WWW*, 2005.

[50] L. Zeltser. Malvertising: Some Examples of Malicious Ad Campaigns. http://blog.zeltser.com/post/6247850496/malvertising-malicious-ad-campaigns, June 2011.

JShield: Towards Real-time and Vulnerability-based Detection of Polluted Drive-by Download Attacks

Yinzhi Cao[†], Xiang Pan, Yan Chen, and Jianwei Zhuge[††]
Northwestern University [†]Columbia University [††]Tsinghua University

ABSTRACT

Drive-by download attacks, which exploit vulnerabilities of web browsers to control client computers, have become a major venue for attackers. To detect such attacks, researchers have proposed many approaches such as anomaly-based [22, 23] and vulnerability-based [44, 50] detections. However, anomaly-based approaches are vulnerable to data pollution, and existing vulnerability-based approaches cannot accurately describe the vulnerability condition of all the drive-by download attacks.

In this paper, we propose a vulnerability-based approach, namely JShield, which uses novel opcode vulnerability signature, a deterministic finite automaton (DFA) with a variable pool at opcode level, to match drive-by download vulnerabilities. We investigate all the JavaScript engine vulnerabilities of web browsers from 2009 to 2014, as well as those of portable document files (PDF) readers from 2007 to 2014. JShield is able to match all of those vulnerabilities; furthermore, the overall evaluation shows that JShield is so lightweight that it only adds 2.39 percent of overhead to original execution as the median among top 500 Alexa web sites.

1. INTRODUCTION

In the past few years, drive-by download attacks exploiting browser vulnerabilities have become a major venue for attackers to control benign computers including those of reputable companies. For example, in February 2013, both Facebook and Apple confirm being hit in "watering hole attack" [4], a variance of drive-by download attack. In such an attack, the attacker compromises a web site commonly visited by victims, injects drive-by download attacks, and then waits for victims to come, just as a predator sitting at a water hole in a desert for prey.

To defeat such attacks, many anomaly-based approaches [21–23, 46], which tune the detection engine based on attacker-generated exploits and benign web sites, have been proposed and significantly improved state of the art. However, although anomaly-based approaches can detect unknown attacks, they could succumb in adversarial environments. Inspired by the data pollution on polymorphic worm detection [41], we designed and implemented polluted drive-by download attacks, which could significantly reduce the detection rates of those anomaly-based detectors.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.
ACSAC '14, December 08 - 12 2014, New Orleans, LA, USA
Copyright 2014 ACM 978-1-4503-3005-3/14/12$15.00
http://dx.doi.org/10.1145/2664243.2664256.

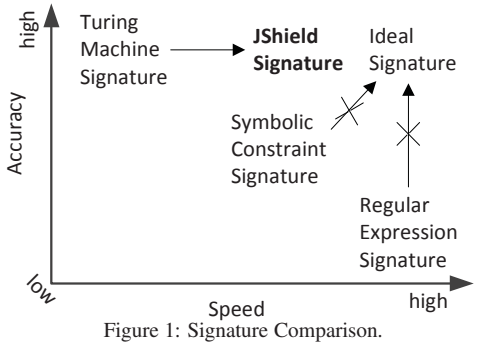


Figure 1: Signature Comparison.

To countermeasure such data pollution, we turn to vulnerability-based approaches. In this emerging direction, although the precise description of a network protocol layer vulnerability signature has already been used in the network intrusion detection systems (NIDS) [32, 54], its application on Web drive-by download attack detection is very limited. In BrowserShield [44], Reis et al. rewrite JavaScript codes and enforce stateless security policies (signatures), such as checking the parameter length of a function call. The other work by Song et al. [50] matches the inter-module communication with vulnerability signatures. However, neither of these techniques can precisely represent the exact vulnerability conditions of drive-by download attacks. More specifically, the rules in BrowserShield do not have stateful data structures to record control or data flow, and the signatures in the work by Song et al. lack sufficient information for control flow at the inter-module level.

In summary, we believe that an approach detecting drive-by download attacks should address the following **challenges:**

- **Signatures for Stateful Intra-module Vulnerabilities.** The signature for drive-by download attacks targeting stateful intra-module vulnerabilities should contain both control and data flows during matching.
- **Resilience to Polluted Data.** The system should be resilient to polluted training and testing samples provided by the attackers.
- **High Performance.** The runtime detection system should have an acceptable overhead to the pure execution of web pages.

In this paper, we propose JShield, vulnerability-based detection of drive-by download attacks. JShield has been adopted by Huawei, the world's largest telecommunication company. We position JShield opcode signatures in Figure 1 with other known vulnerability signatures. Neither symbolic constraint signature nor regular expression signature can represent a drive-by download vulnerability that takes Javascript, a Turing complete language, as input, since they do not have loops. Meanwhile, a traditional Turing machine signature is so complex that it requires a signature as large

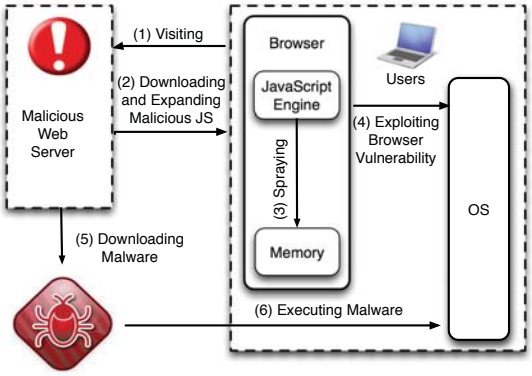


Figure 2: Six Steps of a Drive-by Download Attack.

as a whole browser. Thus, we abstract lower level Turing machine to a higher level opcode Turing machine and design detection format of our opcode signature. To further make it scalable, we also use regular expression (filter format of opcode signature) to filter a large number of benign traffic.

The current signature generation process involves some manual effort. However, we believe that the amount of manual work is small due to the small number (approximately 10) of vulnerabilities each year. Actually, even for a large amount of signatures, most network intrusion detection/prevention system (IDS/IPS) vendors, such as Snort, Cisco and Juniper, all generate these signatures manually [5, 13].

Besides being the $first$ to design polluted drive-by download attacks and evaluate their effectiveness on the state-of-the-art anomaly-based approaches, we make the following contributions in this paper:

- **Stateful Drive-by Download Vulnerability Signatures.** Our vulnerability signature is a deterministic finite automaton (DFA) with a variable pool recording both control and data flows to detect stateful intra-module vulnerabilities of drive-by download attacks.
- **Vulnerability Signature at the Opcode Level.** A vulnerability signature at the opcode level can precisely describe a given vulnerability.
- **Fast Two-stage Matching.** We design a two-stage matching process to speed up the detection. The filtering stage adopts fast regular expression matching for a given test sample, and then if the sample is captured at the filtering stage, it is subject to a further matching with opcode signature.

To evaluate JShield, we investigate the vulnerability coverage of JShield and find that JShield is able to handle all the recent JavaScript engine vulnerabilities of web browser and portable document files (PDF) reader. The overall evaluation shows that JShield has introduced little performance overhead to pure execution. The average overhead of top 500 Alexa web sites is 9.42% and the median is 2.39%.

2. THREAT MODEL

The paper focuses on drive-by download attacks consisting of two major stages: pre-exploit stage and post-exploit stage, which can further divide into six steps as shown in Figure 2.

At the pre-exploit stage, a benign user is lured to visit a malicious web site (step one). Then, malicious contents are downloaded, and malicious JavaScript codes, possibly obfuscated by eval, setTimeout, and DOM events, get expanded by execution (step two). During execution, some malicious JavaScripts also fill the heap with shellcodes and nop sleds to overcome address space layout randomization and facilitate attacks (step three).

After all the preparation, the malicious JavaScript exploits a certain vulnerability (step four), and thus the injected shellcode takes control of the browser to download malware (step five), which is subsequently executed on the victim machine (step six).

To distinguish our threat model from others, we also mention other attacks below - these however are *out of scope* of the paper.

Attacks without any JavaScript Interaction. Similar to existing works like Zozzle [23], we only focus on the JavaScript part of a drive-by download attack. If an attack is purely related to HTML parser, CSS parser or font processing, neither Zozzle with abstract syntax tree (AST) features nor JShield with opcode vulnerability signatures can detect the attack. Instead, due to lack of full featured obfuscation techniques, this type of attacks should be prevented by traditional NIDS.

Other Web-based Attacks. Other web-based attacks, such as cross-site scripting (XSS) attacks, cross-site request forgery (CSRF), and so on are out of scope of the paper. People should rely on existing defense mechanisms [19, 52] for those attacks.

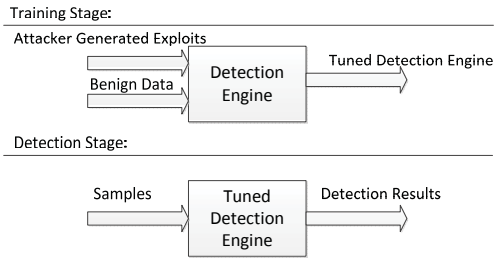


Figure 3: Anomaly-based Approaches to Detect Drive-by Download Attacks.

3. POLLUTED DRIVE-BY DOWNLOAD ATTACKS

Anomaly-based approaches [18, 22, 23, 46] first train a detection engine based on exploits generated by attackers as well as benign samples collected from the Internet, and then perform detection by the tuned engine. An overview architecture is shown in Figure 3. Normally, a machine learning engine is deployed in the training stage, which extracts malicious and benign features from training data and then trains the detection engine.

In this section, we discuss the efficacy of anomaly-based detection in adversarial environment. We understand that the advantage of anomaly-based detections is that they can detect unknown attacks, however in this paper we only focus on the behavior of anomaly-based detection in adversarial environment.

Generally, an anomaly-based detection could be evaded in two ways, namely, polluting attacker generated exploits in training stage and altering malicious samples in detection stage. We introduce data pollution in detection stage first because of its effectiveness. For data pollution, we use naive Bayes engine adopted by Zozzle [23] as an example to show how to evade anomaly-based approaches.

3.1 Polluting Samples at Detection Stage

An attacker can alter malicious samples by injecting benign features (BF), thus increasing the probability of classifying those samples as benign and evading naive Bayes engine. Intuitively, the benign features decrease the anomaly by reducing the significance

Table 1: Zozzle's Detection Rate.

	Original Rate	Rate after Pollution at Detection Stage
True Positive	93.1%	36.7%
False Positive	0.5%	0.5%

of the malicious features statistically. Here is the detailed reason. Assume there is a sample with malicious feature (MF). According to the definition, $P(M|MF)$, the probability of malice given the existence of one malicious feature, is larger than 0.5. Now let us assume that one benign feature (BF) is introduced to that file. Given that MF and BF are independently distributed in naive Bayes, we have Equation 1.

$$\begin{aligned} P(M|MF,BF) &= \frac{P(MF,BF|M) * P(M)}{P(MF,BF)} \\ &= \frac{P(MF|M) * P(BF|M) * P(M)}{p(MF) * P(BF)} \\ &= \frac{P(BF|M)}{P(BF)} * P(M|MF) \end{aligned} \quad (1)$$

Since fewer BFs exist in malicious files, $\frac{P(BF|M)}{P(BF)} < 1$. Thus, Equation 1 shows that the existence of one benign feature reduces the probability of the sample's malice. If enough benign features are introduced, $P(M|MF, BF, BF1, BF2...)$, the probability of the sample's malice, will be eventually less than 0.5, resulting in a mislead of the Bayes classifier.

3.2 Real-world Experiments

We use Zozzle [23], the most recent and successful machine learning detection, as a case study to show how to evade an anomaly-based approach, however, our discussion is not restricted to Zozzle. Since Zozzle is not open source, we strictly followed what has been described in the paper [23], implemented our version of Zozzle, and reproduced comparable detection rate for unpolluted samples as the one reported by Zozzle.

The testing data set is from Huawei, which contains unclassified malicious JavaScript codes collected from their customer reporting, other anti-virus software reporting, etc. After filtering all the landing pages, we collect 880 malicious samples (Zozzle adopts 919 JavaScript malware samples [23]). Meanwhile, Top 5,000 Alexa web sites [3] are used as benign samples during training. After manual and automatic selection documented in Zozzle, we collect 300 benign and malicious features.

Since an attacker cannot acquire our benign features used in the system, we collect all the common features among Top 5,000 Alexa web sites and add them to malicious samples. The size of each malicious sample increases 45% on average, *i.e.*, 396KB. As shown in Table 1, pollution at the detection stage decreases the overall accuracy to 36.7%.

In addition to Zozzle, we also evaluated five popular anti-virus programs[1] selected from an anti-virus software review web site [1], which are: 1) Avira Antivirus Premium 2013, 2) AVG Internet Security 2013, 3) Kaspersky Internet Security 2012, 4) Norton Internet Security 2013 and 5) Trend Micro Titanium Internet Security 2013. The results are shown in Table 2. Before data pollution, the detection rates of anti-virus software except AV4 are all above 85%. However, after data pollution, the detection rates of them are all below 4%. Therefore, existing anti-virus software, which are blackbox to us and some of which belong to regular expression based approach, are not robust to data pollution either.

4. OVERVIEW

[1]This order does not correspond to the order of AV1 to AV5 in Table 2.

Table 2: The detection rate of five anti-virus programs before and after data pollution.

Anti-virus	Original Rate	Rate After Pollution
AV1	98.00%	0.58%
AV2	89.33%	3.58%
AV3	92.41%	2.00%
AV4	20.67%	0.08%
AV5	87.58%	2.00%

As discussed in previous section, attacker generated samples adopted by anomaly-based detections could be polluted, and thus we resort to vulnerability-based detections. In this section, we first present two types of deployment for JShield. Then, we model drive-by download vulnerabilities based on their control and data flows.

4.1 Deployment

JShield is a dynamic vulnerability signature based approach to de-obfuscate and detect drive-by download attacks. There are two major types of deployment for JShield: 1) at the Web Application Firewalls (WAF) or Web IDS/IPS and 2) at Web malware scanning services. For the former, JShield is deployed as a component of anti-virus software, or as a detection engine at Internet Service Providers (ISP) gateways. For example, Huawei deploys our system in their intelligent cloud, inspecting potential malicious traffic from their switches and routers. On the other hand, JShield can also be deployed on the sever side as a Web malware scanning service, by search engines such as Google and Bing, or by a security company for online web malware scanning.

Compared to an anomaly based approach like Zozzle [23] which needs to retrain the detection engine to accommodate new drive-by download exploits, JShield only needs to update its vulnerability signature database for new drive-by download vulnerabilities. Due to that fact that the number of vulnerabilities is always much less than the amount of exploits, the update overhead of JShield will be small.

4.2 Vulnerability Modeling

Traditionally, there are three types of signatures in literature [20]: Turing machine signature, symbolic constraint signature, and regular expression signature. We look into those signatures in the context of a drive-by download attack where the vulnerable program is a browser and the input exploit is a JavaScript program. The Turing machine signature generation process from Brumley et al. [20] would output a signature as large as a browser, hence making it unusable. On the other hand, neither symbolic constraint signature nor regular expression can represent a complex language like JavaScript which have loops. Thus, for preciseness, we need to have a new signature between a Turing machine signature generated by Brumley et al. and a symbolic constraint signature. For matching speed, we first use regular expression signature to filter a majority of benign JavaScript and then detect malicious JavaScript using the precise detection format of our opcode signature. Since regular expression signature is well known, we focus on the detection format of our opcode signature.

To model a vulnerability, we define a vulnerability condition as a function that takes a certain path of a program's control and data flow graph, and output whether the path is exploitable or not. Formally, given $c \in C$, where C is all possible paths of the program's control flow graph, and $d \in D$, where D is all possible paths of the program's data flow graph, a vulnerability condition k is a function,

$k : C \times D \rightarrow \{Safe, Exploit\}$

In order to match a certain vulnerability, its corresponding vulnerability signature need to match both the path in the control flow

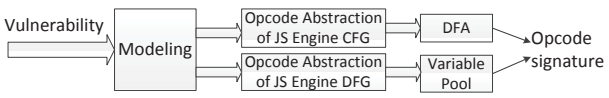


Figure 4: Vulnerability Modeling.

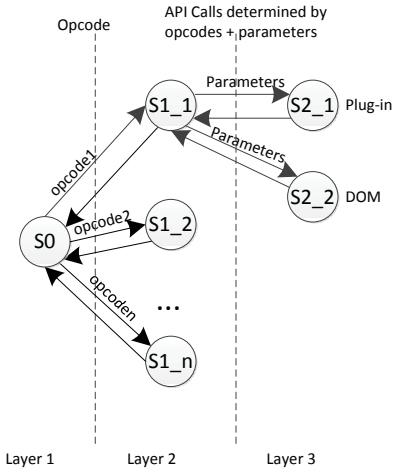


Figure 5: Simplified Opcode Level Control Flow Graph of JavaScript Interpreter.

graph and the one in the data flow graph. In the context of a drive-by download attack where the vulnerability exist in a JavaScript engine or a plugin, our observation is that control and data flow graphs of the opcode input is an abstraction of the control and data flow graphs of the native JavaScipt engine. Thus, as shown in Figure 4, we can model the underline vulnerability in the JavaScript engine or plugin by abstracting the control and data flow to the opcode level.

To study the *control flow*, we investigate the source code of several JavaScript engines [14, 17, 39], and find that the main body of a JavaScript interpreter switches to different branches of codes based on the input opcode through a code pattern similar to *select input case opcode*. Then, different opcodes and parameters determine subsequent API calls to external objects, plugin or DOM. Therefore, we form the control flow graph (CFG) of JavaScript engine into an opcode driven three-layer structure as shown in Figure 5. The CFG will shift based on the input opcode sequence. In other words, opcode CFG is built upon the original application CFG.

Next, we categorize vulnerabilities into two types: JavaScript engine vulnerability (including a web browser JavaScript engine and/or a plugin JavaScript engine such as Adobe Reader JavaScript engine) and external JavaScript engine vulnerability. We will explain them respectively.

Let us assume that the control flow graph of a JavaScript engine vulnerability condition is triggered by travelling through S1_1 and S1_2 of Figure 5. Since there is only one path to travel to S1_1 and/or S1_2, which is to offer the corresponding opcode, the opcode level signature represents this vulnerability.

For an external JavaScript engine vulnerability, the API calls to those components such as plugin and DOM are determined by opcode sequences and parameters. Song et al. [50] show that the inter-module communication can represent a vulnerability and thus our opcode signature can achieve the same functionality.

Therefore, we propose an opcode level deterministic finite automaton (further explained in Section 5) to match the opcode control flow, an abstraction of the JavaScript engine control flow, in a vulnerability condition.

```
1 switch (opcode) {
2   case get_by_id:
     // (1) look up the prototype chain
     // (2) invoke getter method
     // (3) move results to register r0
3         break;
4   case put_by_id:
     // (4) move null to the prototype
5    break;
6 }
```

Figure 6: Pseudo JavaScript Engine Codes for CVE-2009-1833.

```
1  var obj = new Object();
2  obj.__proto__.__defineGetter__("a"
                        , function () {
3                 this.__proto__ = null;
4               gc();
5               return 0;
6               });
7  obj.a;
```

Figure 7: Example I: CVE-2009-1833: Malicious JavaScript Exploit that can Trigger a Mozilla Firefox JavaScript Engine Vulnerability.

To study the *data flow*, JShield needs to record additional states related to the vulnerability. Therefore, we propose a variable pool (further explained in Section 5) to match the opcode data flow, an abstraction of JavaScript engine data flow, in a vulnerability condition.

Now, we illustrate the point by a concrete running example from CVE database. In Figure 6, we show how to trigger CVE-2009-1833 in the pseudo code of JavaScript engine. The vulnerability is triggered by two conditions: (i) looking up through prototype chain to get a getter function, and (ii) setting the prototype itself to be null inside the getter function.

When we abstract the JavaScript engine vulnerability to the opcode level and take a look at a concrete exploit example triggering vulnerability in Figure 7 and Figure 8, we find that the opcode level CFG is an abstraction of the underline level JavaScript engine CFG. S1_1 in Figure 5 is visited by *get_by_id* and S1_2 is visited by *put_by_id*. For the data flow, to match CVE-2009-1833, JShield needs to remember the memory address of the prototype.

In sum, the JShield signature needs to match both the control flow graph and the data flow graph of opcode sequence of a JavaScript code, an abstraction for the control and data flow graph of the underline JavaScript engine, for a drive-by download vulnerability condition as shown in Figure 4.

5. OPCODE SIGNATURE

As discussed in Section 4.2, a successful opcode signature needs to match both the control flow and the data flow of a vulnerability condition. In this section, we introduce the detailed design of opcode signature matching drive-by download vulnerabilities. To speed up the matching process, two types of opcode signature, the detection format and the filter format, are described here by their definition, structure and matching process. In the end, we present the robustness of opcode vulnerability signature to polymorphic attacks.

In the current version of JShield, all the opcode vulnerability signatures are generated manually for each vulnerability. However, we believe that the amount of involved manual work is small due to the small number ($<$100) of vulnerabilities each year. Actually, even for a large amount of signatures in an intrusion detection system

```
[ 199] get_by_id    r0, r1, a(@id1)
[   0] enter
[   1] convert_this r-7
[   3] mov       r0, r-7
[   6] put_by_id  r0, __proto__(@id0), Null(@k0)
[  15] ret Int32: 0(@k1)
```

Figure 8: Generated Opcode Snippet when Executing JavaScript Codes in Figure 7.

like Snort, those signatures are all generated manually [13]. Further, we also expect future improvement can automate vulnerability signature generation. As an analogy, Shield [54] proposes protocol level vulnerability signatures, and then Brumley et al. [20] propose their automatic generation.

5.1 Definition

Opcode signature is a signature to match an opcode sequence, an instruction set generated by a JavaScript interpreter for efficient execution. For example, opcodes in Figure 8 are the results of transmitting JavaScript code in Figure 7. Op code signature has two formats: a detection format used for matching and a filter format used for fast filtering.

We first formalize detection format of opcode signature as a deterministic finite automaton (DFA) plus a variable pool in Definition 1.

DEFINITION 1. *We define detection format of opcode signature as a 10-tuple (Q, Σ, P, V, g, G, f, q_0, p_0, F), where*

- *Σ is finite set of input symbols, P is finite set of variables, V are the value set of P, and Q is finite set of states.*
- *g, is a function $P \rightarrow V$.*
- *G is the set of all possible g.*
- *f is a transition function, $Q \times \Sigma \times G \rightarrow Q \times G$.*
- *$q_0 \in Q$ is a start state, $p_0 \subseteq G$ is an initial variable pool, and $F \subseteq Q$.*

For input $a_i \in \Sigma$ and a variable pool $p \subseteq G$, the next state of the automaton obeys the following conditions:

1. $r_0 = q_0, p = p_0$.
2. $< r_{i+1}, p > = f(r_i, a_i, p)$, *for i = 0,..., n-1.*
3. $r_n \in F$.

In Definition 2, we formalize the filter format[2] of the opcode vulnerability signature as a regular expression.

DEFINITION 2. *We define the filter format of opcode signature as a 5-tuple (Q, Σ, f, q_0, F), where*

- *Σ is finite set of input symbols, and Q is finite set of states.*
- *f is a transition function, $Q \times \Sigma \rightarrow Q$.*
- *$q_0 \in Q$ is a start state, and $F \subseteq Q$.*

For input $a_i \in \Sigma$, the next state of the automaton obeys the following conditions:

1. $r_0 = q_0$.
2. $r_{i+1} = f(r_i, a_i)$, *for i = 0,..., n-1.*
3. $r_n \in F$.

5.2 Structure

We introduce the structures of the detection and filter format of opcode signatures in this section.

[2]Unless specified, opcode signature refers to the detection format of opcode signature. The filter format refers to the filter format of opcode signature.

Detection Format:

#	Method	Opcode	Condition	Action	Next
(1)	match	get_by_id	isFromProtoChain()	x=proto	(2)
	default			quit	N/A
(2)	match	enter	true	i=0	(3)
	default			quit	N/A
(3)	match	enter	true	i=i+1	(3)
	match	ret	i==0	quit	N/A
	match	ret	i>0	i=i-1	(3)
	match	put_by_id	x==dst & src==null	report	N/A
	default			jmp	(3)

Filter Format:

```
get_by_id enter .* put_by_id
```

Figure 9: Opcode Signature for CVE-2009-1833.

5.2.1 Detection Format

The detection format of an opcode signature, as shown in Figure 9, can be formalized into the following three concepts: clause, sentence, and signature.

A *clause* in an opcode signature consists of five fields: "method", "opcode", "condition", "action", and "next". The "method" field specifies what to be taken in this clause, where two methods are currently defined: "match" and "default". "Match" means to match the opcode, and "default" means that the default actions should be taken if no matches are found in other clauses. Then if both "opcode" field matches the input opcode and the expression in "condition" field is true, the action in the "action" field will be taken and current state will be transferred to the number in "next" field, which represents the sentence number that will be explained right after this paragraph.

Multiple clauses plus an index number together build *a sentence*, a state in automaton. The number is used to differentiate one sentence from the others. The clauses in one sentence are in sequence, which means if JShield finds the first match, the remaining ones will be skipped. If no matches are found, the action corresponded with the "default" clause will be taken.

A *signature* consists of multiple sentences. During matching, the automaton will transfer from one sentence to anther based on the matching results.

5.2.2 Filter Format

The filter format of an opcode signature, as shown in Figure 9, can be formalized as a regular expression, which takes a series of opcodes as input. For detection and filter format of opcode signature, the following statement holds: "Each detection format of an opcode signature has a corresponding filter format of that opcode signature".

Here is the reason. Given a detection format of an opcode signature, for each sentence, we extract all *Opcode* fields and align them into a unit by bracket symbol of regular expression. Then, by following the jmp operations in *Action* field, we align the units into a regular expression. If a self loop is recognized, a symbol $*$ is introduced. The end of the regular expression is one or multiple opcodes in bracket that leads to the vulnerability.

5.2.3 Data Structure of Both Formats

The filter format is simply stored as a regular expression. To speed up matching process, we construct a reverse index for the detection format of opcode signatures by the opcode field. Suppose we have two signatures: Sig1 and Sig2. Each signature has two sentences. Each sentence has two clauses. Under each opcode, both two signatures exist. Under each signature, both two sentences exist. Under each sentence, only the clause with that opcode is

Algorithm 1 Matching Detection Format

```
State ← Starting_State
for Input_Opcode in All_Opcodes do
    for Signature in Pool[Input_Opcode] do
        Sentence ⇐ Signature[State]
        Clause ⇐ Sentence.Clause
        if equal(Method, Match) then
            if (IsAllTrue(Clause.Conditions) then
                Take Actions
                Signature.State ⇐ New_State
                Break
            end if
        else[equal(Method, Default)]
            Default Actions
            Signature.State ⇐ Default_State
            Break
        end if
    end for
end for
```

Sample One:

```
1 this.__defineGetter__("x", function (x){
2             'foo'.replace(/0/g,[1].push)
3           });
4 for (let y in [,,,])
5    for (let y in [,,,])
6        x = x;
```

Sample Two:

```
1 while (true) {
2   Array.prototype.push(0);
3 }
```

Figure 10: Example II: Different Samples that Trigger CVE-2009-0353 (Extracted and Simplified from CVE Library).

present. Other clauses of that sentence are under other opcodes.

5.3 Generating Opcode Vulnerability Signature

We generate the opcode vulnerability signatures semi-automatically with the following three steps.

1. Based on the semantics of the vulnerability (*e.g.*, from the CVE description or vulnerability patches), we locate the opcodes that are involved in the vulnerability. We create a DFA with each involved opcode being a node (state).
2. From the data flow part, we extract the critical data structure involved in the vulnerability related to each opcode operation and define a variable in the variable pool of "Action" field in opcode signature.
3. We combine the DFA and the variable pool together by introducing each variable to the "Condition" field of opcode signature based on the data flow connection between opcodes.

Again, we use CVE-2009-1833 in Figure 7 as an example. We first automatically generate control and data flow [28]. Then, manual work is involved to find out the semantics of the vulnerability, *e.g.*, for CVE-2009-1833, line 3 and line 7 together cause the vulnerability. From the control flow graph part, the sequence of three opcodes (get_by_id, enter, and put_by_id) will lead to the vulnerability condition. On contrary, another sequence of three opcodes (get_by_id, enter, and ret) will lead to a safe state. Therefore, we create a detection format of our opcode vulnerability signature with three states in DFA as shown in Figure 9, and meanwhile, we use a counter i to record the number of opcode enter and ret. Next, from data flow part, we find that line 3 and line 7 are connected by the memory address of the prototype, and therefore, we use x in variable pool to record that data. In the end, we combine the DFA and the variable pool to the detection format of our opcode vulnerability signature, and follow steps discussed in Section 5.2.2 to generate the filter format of opcode vulnerability signature.

5.4 Matching Opcode Vulnerability Signature

The matching process of opcode signatures can be divided into two parts: pre-matching by the filter format of opcode and matching by the detection format of opcode signature.

At the filtering stage, we match the opcode sequence outputted from de-obfuscation engine with the filter format of opcode signature. If a sequence of opcodes does not match with the filter format of opcode signature, we drop it off because it will not match with the detection format of that opcode signature either. By filtering a large amount of unmatched samples using fast regular expression operation, we can accelerate the total matching process.

After the filtering stage, we match the opcode sequence with detection format. The pseudo-code of the matching algorithm is shown in Algorithm 1. Given one opcode as an input, the matching algorithm goes over every signature with that opcode. For each signature, JShield directly fetches the corresponding clause that belongs to the sentence of the current state because JShield has already indexed all the signatures by opcodes. Then, JShield checks whether the conditions are satisfied, and accordingly takes actions. The complexity for this process is O(Maximum number of Signatures per Opcode $\times$ Number of Opcodes).

5.5 Robustness to Polymorphic Attacks

A CVE-2009-0353 example in Figure 10, which is triggered when repeating push operation of an array exceeding the memory limit, shows that JShield reduces polymorphic attacks. Instead of reporting an out-of-memory error, illegal memory address will be overwritten.

In Figure 10, we show two different snippets of JavaScript triggering CVE-2009-0353, which fire push operation in two different ways at the JavaScript level. However, at the opcode level, both need to call opcode *get_by_id* first to get *push* method, and then repeat using opcode *call* (only one *call* is shown in the figure). After generating the call graph for two JavaScript engines [14, 16] by doxygen [7] and thus examining functions that calls *push* method, we find that the only way of calling *push* method is through the opcode *call*. In other words, the opcode signature maps to the vulnerability.

For polymorphic attacks, we show that the number of polymorphism reduces at the opcode level because (i) one opcode signature maps to multiple source code representations of that vulnerability; and (ii) one source code representation of a vulnerability maps to one specific opcode signature given an opcode instruction set.

The reasons are as follows. Assume a vulnerability is represented by an opcode signature. We follow the state transition and get opcode sequence as follows: op_code1, op_code2, op_code3 and so on. Each opcode can be included in multiple JavaScript statement. For example, *op_call* can be triggered by direct function call and *getter* property of an object. Similarly, *op_jmp* can be triggered by *while* loop, *for* loop, and so on. We will choose different JavaScript statement and write corresponding JavaScript source code. To the opposite, if we have a source code representation of a vulnerability and the opcode instruction set is fixed, we can feed

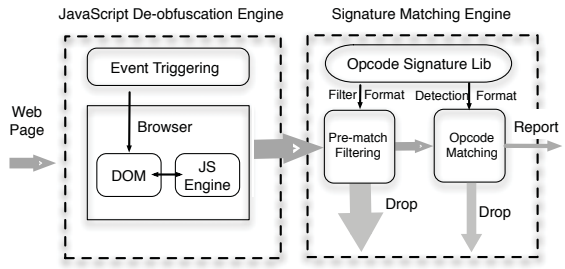


Figure 11: System Architecture.

the source code into the interpreter with the opcode instruction set. One unique opcode sequence is outputted from the interpreter.

6. SYSTEM ARCHITECTURE

Figure 11 shows the overall architecture of JShield, which consists of two main engines, a JavaScript de-obfuscation engine and a signature matching engine. The former takes a web page as input, de-obfuscates JavaScripts, and then outputs the corresponding opcode sequence; the latter takes an opcode sequence as input, matches the sequence with opcode signature, and finally gives a report about whether the incoming web page is malicious .

The detailed process is as follows. When a web page is fed to a JavaScript de-obfuscation engine of JShield, it is executed on a real browser with event triggering module, which mimics user's behaviors to trigger all the DOM events. If the web page contains PDF, JShield adopts MPScan [35] to hook an Adobe Reader and output all the opcode sequences. After de-obfuscation, the signature matching engine first filters opcode sequences outputted from JavaScript de-obfuscation engine. For the opcode sequences not filtered out, JShield further matches them with detection format of opcode vulnerability signatures.

7. IMPLEMENTATION

We use WebKit r101347 together with Qt 4.8.1 on Linux to implement JShield. Web pages are directly fed into a modified version of WebKit that is integrated with event triggering engine. External libraries and virtual functions are loaded into WebKit through JavaScript files. Then opcodes are generated and fed into the opcode matching engine.

De-obfuscation. We introduce the de-obfuscation engine of JShield, which increases the total amount of inspected source code written by attackers, from two aspects: events recording, and event triggering.

We first modify class Document to make every document maintain a queue, which records all the registered event listeners. Then we modify function addEventListener in Node.cpp file. We choose to modify this function in order to make sure that all listeners would be registered, because this function is called every time a listener is added. So when an event listener is trying to register itself on a node, the node will call its method addEventListener(), in which the node adds the listener in the queue maintained by the node's Document object. The modified addEventListener() function also determines whether to call eventTriggering() function to trigger the new registered event listener right away, based on whether the onload event listener is invoked.

Then, we add two functions: void trigerAllWindowEvents() in DOMWindow.cpp file, used to trigger all the event listeners registered on Window object, and void eventTriggering() in Document.cpp file, used to iterate and trigger all its children nodes recorded in the queue. Next, we modify the function void dispatchWindowLoadEvent() in Document.cpp to call trigerAllWindowEvents() and eventTriggering() functions and trigger all the event listeners registered so far.

Signature Matching Engine. We first extract opcodes from WebKit engine of which the interpretation mode for JavaScript is enabled. All the opcodes are interpreted by *privateExecute* function in *Interpreter.cpp* at JavaScriptCore of WebKit. We extract all the register address together with opcode names during the interpretation and feed them into the opcode matching engine.

Next, we use the standard regular expression library [12] in C++ to match the filter format of opcode signature, which is stored as a string. After filtering, the detection format of opcode signatures is stored as a *map* container of STL [15], which uses a red-black tree structure. All the signatures are indexed by opcodes. The data domain of opcode signatures are implemented by a perl interpreter in C [8].

8. EVALUATION

We first introduce our methodology in Section 8.1. Next, we evaluate vulnerability coverage in Section 8.2 and robustness to data pollution in Section 8.3. In the end, We evaluate the performance of JShield including *JavaScript obfuscation engine*, *signature matching engine*, and the overall system in Section 8.4.

8.1 Methodology

To evaluate JShield, we obtain JavaScript engine vulnerabilities and plugin vulnerabilities from CVE database, the details of which can be found in Section 8.2. We adopt three metrics to evaluate signature matching:

- *Vulnerability Coverage Rate.* Vulnerability coverage rate is defined as the number of vulnerabilities covered by a certain approach divided by the number of all the vulnerabilities.
- *Robustness to Data Pollution.* We evaluate the robustness to data pollution by detecting attacks injected with benign features and attack variants using the same vulnerability.
- *Performance.* We measure the latency caused by the event triggering process, the signature matching process, and the overall JShield.

8.2 Vulnerability Coverage

Two sets of vulnerabilities are evaluated on JShield, which are JavaScript engine vulnerabilities and plug-in vulnerabilities. JShield contributes on detecting JavaScript engine vulnerabilities, but for plug-in vulnerabilities, we are on par with existing works [44, 50].

8.2.1 Data Source

We evaluate vulnerability coverage of JShield on those vulnerabilities in national vulnerability database (NVD) [11]. (i) To acquire JavaScript engine vulnerabilities of web browsers, we search the keyword "JavaScript Engine" at the search engine provide by NVD, and examine the "References to Advisories, Solutions, and Tools" part of all the results. Each "External Source" will be examined. We pay special attention to the exploiting codes posted by each external source. (ii) To acquire JavaScript engine vulnerabilities of pdf readers, we search the keyword "JavaScript reader" and "JavaScript pdf" at the search engine provide by NVD. The same procedure for a JavaScript engine vulnerability will be applied to this type of vulnerability. (iii) To acquire inter-module plugin vulnerabilities, we obtain information from Song et al [50].

During investigation, we output the opcode sequence of exploiting codes. Based on the output opcodes, we will form an opcode signature. If we can successfully construct an opcode signature, we

Table 3: Feasibility Comparison of JShield with Other Vulnerabilities-based Drive-by Download Attack Detection.

Vulnerability Position	BrowserShield [44]	Song et al. [50]	JShield
JS Engine	3/22	0/22	22/22
PDF JS Engine	4/18	0/18	18/18
Plug-in	20/21	21/21	21/21

Note: All the numbers in the table are theoretical detection ratio for each approach. BrowserShield represents those with simple policy without control and data flow matching, and Song et al. represent those with only intra-module communications.

Table 4: JShield's Detection Accuracy under Data Pollution.

	Original	Pollted
TP for Web Pages	100%	100%
FP for Web Pages	0%	0%
TP for PDF	100%	100%
FP for PDF	0%	0%

Note: TP is short for true positive, and FP is short for false positive.

will consider JShield can detect the vulnerability. If the signature contains only one state, we will consider BrowserShield [44] can detect the vulnerability. If the vulnerability does not contain any multi-module communication, we will consider Song et al. [50] cannot detect the vulnerability.

8.2.2 Coverage Results

Three types of vulnerabilities are evaluated here, which are JavaScript engine vulnerabilities of web browsers, JavaScript engine vulnerabilities of PDF readers, and inter-module plug-in vulnerabilities, respectively.

- *JavaScript Engine Vulnerabilities of Web Browsers.* We evaluate all twenty four JavaScript engine vulnerabilities from 2009 to 2014. We do not find any test cases for CVE-2011-2991 and thus skip this vulnerability. CVE-2011-4682, which bypass the same origin policy, is out of scope of this paper and thus skipped. For year 2014, because both NVD and CVE prevent public view of recent vulnerabilities due to security concerns, we can only find two JavaScript engine vulnerability at the time of our investigation.
  The results show that JShield can detect all of those vulnerabilities. BrowserShield [44] can only detect 3 out of 22 JavaScript engine vulnerabilities. Song et al. [50] cannot detect any of those vulnerabilities because they are not plug-in vulnerabilities. A summary is shown in the second row of Table 3.
- *JavaScript Engine Vulnerabilities of PDF Reader.* Except for five cross-site scripting vulnerabilities, all the JavaScript engine vulnerabilities of pdf readers from 2007 to 2014. We cannot look in the references related to two Google chrome pdf vulnerability because Google chrome group restrict the permission of viewing recent vulnerabilities. The results of detecting the rest vulnerabilities are shown in the third row of Table 3.
- *Inter-module Plug-in Vulnerabilities.* We obtain all the plug-in vulnerabilities from Song et al. [50], and find that JShield can detect all the plug-in vulnerabilities as shown in the fourth row of Table 3.

8.3 Robustness to Data Pollution

The same malicious data set and methodology as Section 3.1 are used here for web pages. Benign data set for web pages are extracted from Top 500 Alexa web sites. Both malicious and benign PDFs are acquired from security company containing 214 benign and 213 malicious, and the same data pollution for web pages are performed. Since there is no training stage for JShield, the pollution of training data is not applicable for JShield. Results are shown in Table 4. Excluding those samples only contain HTML but no

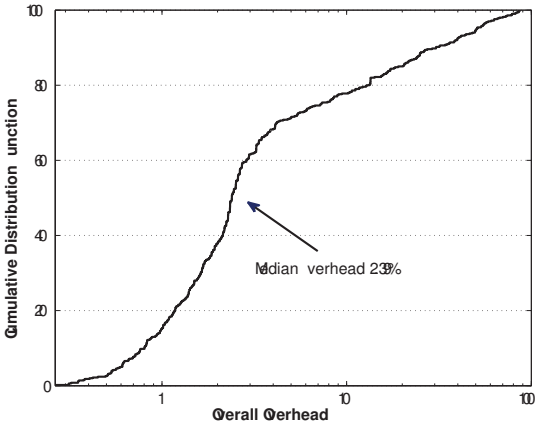


Figure 12: Cumulative Distribution Function (CDF) of Overhead Introduced by JShield on Top 500 Alexa Web Site (The average is 9.42%, and the median is 2.39%.)

JavaScript, JShield can detect all those samples. Note that there is no difference between the detection rate of original data and data with injected benign features. As pointed in Section 2, those attacks are out of our threat model. We do not have any false positives because the vulnerability signatures containing control and data flow information can precisely describe the corresponding vulnerability. Interestingly, we even find that there is no web site passing the first stage, *i.e.* regular expression matching of the filter format.

8.4 Performance

We measure the performance of signature matching engines of JShield and overall performance respectively in Section 8.4.1 and 8.4.2.

8.4.1 Signature Matching Performance

We measure the JavaScript execution latency introduced by JShield against the original execution latency of top ten Alexa [3] web sites. We measure the latency by injecting codes into WebKit engine before and after JavaScript execution. Normally JavaScript is executed multiple times so we accumulated all the latency together. Both the filtering and the detection format opcode matching are measured.

Note that for top one hundred Alexa web sites, we do not find any that can bypass the pre-filtering stage against all the vulnerabilities in our library. In other words, there is no *false positive* for top one hundred Alexa web sites given our data set.

In Figure 13, we show the latency introduced by the filtering and the matching process of JShield. For matching with detection signatures, the latency is comparable to the latency (2 to 14 times slower) introduced by Nozzle [43] without object sampling. In the worst case, Twitter is about 6 times slower than normal execution. The fastest web site Wikipedia is within 2% because there is little JavaScript hosted on Wikipedia.

For filtering, since regular expression matching is very fast, it is predictable that the latency is small. As shown in the figure, filtering overhead is within 2%.

8.4.2 Overall Performance

In Figure 12, we show the overall performance overhead introduced by JShield. The experiment is performed on Top 500 Alexa [3] web site. We test each web site ten times with and without JShield support, get the average latency and then calculate the overhead. The average overhead among 500 web sites is 9.42%, the median overhead is 2.39%, and the standard deviation is 3.17%.

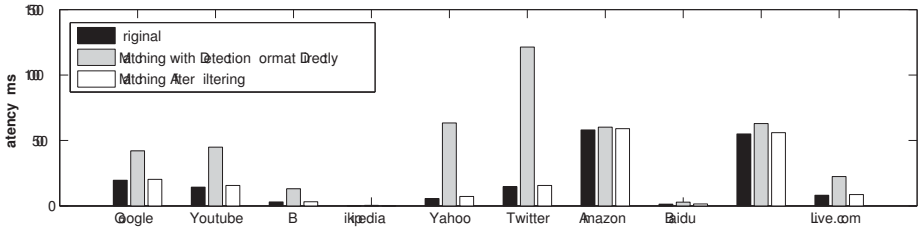


Figure 13: JavaScript Execution Latency Introduced by Matching Filtering and Detection Format of Opcode Signature in JShield (Note that none of the web sites in the graph will actually pass to stage two - detection format matching given our library. We force JShield perform detection format matching for measurement purpose only.)

Since the pre-filtering process - regular expression matching is very fast, most of the overhead is introduced by the event triggering engine.

9. DISCUSSION

Does the opcode signature set change if we implement JShield with a different browser kernel?

Answer: Yes, one opcode signature set binds to one specific JShield implementation, which is similar to the case that one binary can only be executed upon one operating system. However, vulnerabilities in any browser can be represented in one JShield implementation since we do not require the vulnerability to exist.

Do the attackers need to know features extracted by anomaly-based detection method to pollute samples?

Answer: No, at detection stage, benign features are extracted from normal web sites instead of those used by anomaly-based detection engine (generated from normal web sites and malicious web sites).

10. RELATED WORK

In this section, we discuss the related work from three aspects: pre-exploit stage, exploit stage, and the others.

10.1 Pre-exploit Stage

Approaches Detecting Heap-spraying Attacks. Instead of detecting a drive-by vulnerability, researchers propose to detect the heap spraying stage of a drive-by download attack. For example, Nozzle [43] and Egele et al. [25] detect every object/string created by JavaScript to examine whether it is executable. DieHarder [40] provides a new memory allocator for securing the heap from execution.

Those approaches [25, 43] detecting heap spraying can be evaded by newly-emerged technology such as Heap Taichi [24], and most importantly, heap spraying is not a necessary step for a drive-by download attack. As shown in the dissected and categorized Malware samples reported by Zozzle [23], 6 out of 19 samples are not using heap spraying. There are two reasons:

- Address space layout randomization (ASLR) is not enabled by every browser. For example, Internet Explorer 7 on Windows Vista prior to SP1 does not enable ASLR by default [2].
- Several other techniques, such as JIT spraying [10] and spraying by calling other languages including VBScript and ActionScript, can substitute JavaScript heap spraying.

De-obfuscating JavaScripts. Rozzle [30] adopts symbolic execution, multi-execute JavaScript code, and partially mitigates differences between multiple browsers. Other previous approaches [18, 23, 33, 46] mostly execute JavaScript and acquire de-obfuscated JavaScript code. Revolver [29] compares the similarity between different JavaScript samples and cluster them based on AST features, however as shown in Figure 10, two samples with different ASTs can trigger the same vulnerability.

As an important step of detecting malicious JavaScript code, those de-obfuscation techniques can be deployed together with JShield. For example, Rozzle can be included in JShield as a component in the de-obfuscation engine to defeat those mechanisms that detect the monitoring environment. Since we do not have any open source Rozzle available, we leave it as our future work.

10.2 Exploit Stage

There are two ways detecting drive-by download attacks.

Vulnerability-based Approaches. BrowserShield [44], a vulnerability based detection, checks whether a JavaScript operation violates pre-defined policy, thus leading to an attack. Similarly, Song et al. [50] proposes a vulnerability signature to detect plug-in vulnerabilities by checking inter-module communication.

Signatures used in BrowserShield [44] and Song et al. [50] cannot represent stateful intra-module vulnerability such as the one in *Example I*. Further, neither BrowserShield [44] nor Song et al. [50] have considered polymorphic attacks targeting at the same JavaScript vulnerability, like the one in *Example I* of Section 3.

Anomaly-based Approaches. Researchers propose many anomaly-based approaches, such as Zozzle [23], JSAND [22], CUJO [46], and Wepawet [18]. As illustrated in Section 3, anomaly-based approaches have several limitations in adversarial environment, *i.e.*, an attacker can utilize a polymorphic variance of existing JavaScript exploit codes or inject false malicious features to bypass anomaly-based approaches.

10.3 Others

Static Methods. There are also many static analysis of detecting malicious web page, including but not restricted to Prophiler [21], Seifert et al. [48, 49], Ma et al. [36], and so on. Obfuscation technique, such as embedding into *eval* and *document.write*, can evade those approaches.

Protection Mechanisms. There are many protection mechanisms [26, 45, 51, 53] sandboxing a browser principal, which isolate a browser principal from other parallel browser principals and the host operating system. Blade [34] detects whether an executable is downloaded through a browser GUI. If it is not from a browser GUI, the downloaded executable will be quarantined. BrowserGuard [27] adopts similar behaviour based approach to detect downloaded files. Cisco IronPort [6], SpyProxy [37] and WebShield [31] detect drive-by download at middle box and transfer safe contents back to the client. All of those are effective protection mechanisms, but it is far from deploying them upon all the client browsers and enterprise network.

Behavior Based Detection. Provos et al. [42] and Google Safe Browsing [9] use anti-virus software and execution based heuristics to detect the malicious behavior of downloaded malware. Many other approaches [38, 47, 55] use high-interaction client honeypots for detection. Their detection scope is limited because the vulnerability condition might not be triggered in their specific detection environment.

11. CONCLUSION

Due to possible data pollution for anomaly-based detection of drive-by download attacks and no complete vulnerability representation for vulnerability-based detection, we propose opcode vulnerability signature, consisting of a definitive state automaton and a variable pool to represent both control and data flow of a vulnerability condition. We implement a prototype system, called JShield, which de-obfuscates JavaScript by event triggering and then performs opcode signature matching. Next, we investigate all the JavaScript engine vulnerabilities of web browsers from 2009 to 2014, and those of PDF from 2007 to 2014. We find that JShield can detect all of them. We also find that JShield can represent all the inter-module plug-in vulnerabilities obtained from Song et al. [50].

12. ACKNOWLEDGEMENT

This material is based upon work supported in part by Qatar National Research Fund under award ID SP0022512 and National Natural Science Foundation of China under grant NO. 61472209. 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 foundations.

13. REFERENCES

[1] 2013 compare the best antivirus software products. http://anti-virus-software-review.toptenreviews.com/.

[2] Address space layout randomization. http://en.wikipedia.org/wiki/Address_space_layout_randomization#Microsoft_Windows.

[3] Alexa Top Websites. http://www.alexa.com/topsites.

[4] Apple hit by hackers who struck facebook. http://online.wsj.com/article/SB10001424127887324449104578314321123497696.html.

[5] Cisco ips signatures. http://tools.cisco.com/security/center/ipshome.x?i=62&shortna=CiscoIPSSignatures#CiscoIPSSignatures.

[6] Cisco ironport. http://www.cisco.com/web/ironport/index.html.

[7] Doxygen. http://www.stack.nl/~dimitri/doxygen/.

[8] Embedding perl (using perl from c). http://docstore.mik.ua/orelly/perl/prog3/ch21_04.htm.

[9] Google safe browsing. https://developers.google.com/safe-browsing/.

[10] JIT spraying. http://en.wikipedia.org/wiki/JIT_spraying.

[11] National vulnerability database. http://nvd.nist.gov/.

[12] Regular expression library in c++. http://www.cplusplus.com/reference/std/regex/.

[13] Snort rules. http://www.snort.org/snort-rules/.

[14] Spidermonkey javascript engine. https://developer.mozilla.org/en-US/docs/SpiderMonkey.

[15] Standard template library. http://www.sgi.com/tech/stl/.

[16] V8 javascript engine. https://code.google.com/p/v8/.

[17] Webkit source codes. http://webkit.org/building/checkout.html.

[18] Wepawet. http://wepawet.iseclab.org/.

[19] Barth, A., Jackson, C., and Mitchell, J. Robust defenses for cross-site request forgery. In *CCS* (2008).

[20] Brumley, D., Newsome, J., Song, D., Wang, H., and Jha, S. Towards automatic generation of vulnerability-based signatures. In *SP: the 2006 IEEE Symposium on Security and Privacy* (2006).

[21] Canali, D., Cova, M., Vigna, G., and Kruegel, C. Prophiler: a fast filter for the large-scale detection of malicious web pages. In *WWW* (2011).

[22] Cova, M., Kruegel, C., and Vigna, G. Detection and analysis of drive-by-download attacks and malicious javascript code. In *WWW* (2010).

[23] Curtsinger, C., Livshits, B., Zorn, B., and Seifert, C. Zozzle: fast and precise in-browser javascript malware detection. In *the 20th USENIX conference on Security* (2011).

[24] Ding, Y., Wei, T., Wang, T., Liang, Z., and Zou, W. Heap taichi: exploiting memory allocation granularity in heap-spraying attacks. In *ACSAC* (2010).

[25] Egele, M., Wurzinger, P., Kruegel, C., and Kirda, E. Defending browsers against drive-by downloads: Mitigating heap-spraying code injection attacks. In *DIMVA* (2009).

[26] Grier, C., Tang, S., and King, S. T. Secure web browsing with the OP web browser. In *SP: IEEE Symposium on Security and Privacy* (2008).

[27] Hsu, F.-H., Tso, C.-K., Yeh, Y.-C., Wang, W.-J., and Chen, L.-H. Browserguard: A behavior-based solution to drive-by-download attacks. *Selected Areas in Communications, IEEE Journal on* (2011).

[28] Jensen, S. H., Møller, A., and Thiemann, P. Type analysis for javascript. In *SAS: the International Symposium on Static Analysis* (2009).

[29] Kapravelos, A., Shoshitaishvili, Y., Cova, M., Kruegel, C., and Vigna, G. Revolver: An automated approach to the detection of evasive web-based malware. In *USNIX Security Symposium* (2013).

[30] Kolbitsch, C., Livshits, B., Zorn, B., and Seifert, C. Rozzle: De-cloaking internet malware. In *SP: IEEE Symposium on Security and Privacy* (2012).

[31] Li, Z., Tang, Y., Cao, Y., Rastogi, V., Chen, Y., Liu, B., and Sbisa, C. Webshield: Enabling various web defense techniques without client side modifications. In *NDSS* (2011).

[32] Li, Z., Xia, G., Gao, H., Yi, T., Chen, Y., Liu, B., Jiang, J., and Lv, Y. Netshield: Massive semantics-based vulnerability signature matching for high-speed networks. In *SIGCOMM* (2010).

[33] Lu, G., Coogan, K., and Debray, S. K. Automatic simplification of obfuscated javascript code (extended abstract). In *ICISTM: 6th International Conference of Information Systems, Technology and Management* (2012).

[34] Lu, L., Yegneswaran, V., Porras, P. A., and Lee, W. BLADE: An attack-agnostic approach for preventing drive-by malware infections. In *CCS* (2010).

[35] Lu, X., Zhuge, J., Wang, R., Cao, Y., and Chen, Y. De-obfuscation and detection of malicious pdf files with high accuracy. In *HICSS* (2013).

[36] Ma, J., Saul, L. K., Savage, S., and Voelker, G. M. Beyond blacklists: learning to detect malicious web sites from suspicious urls. In *SIGKDD* (2009).

[37] Moshchuk, A., Bragin, T., Deville, D., Gribble, S., and Levy, H. Spyproxy: Execution-based detection of malicious web content. In *16th USENIX Security Symposium* (2007).

[38] Moshchuk, A., Bragin, T., Gribble, S. D., and Levy, H. M. A crawler-based study of spyware on the web. In *NDSS* (2006).

[39] Mozilla. Narcissus javascript engine. http://mxr.mozilla.org/mozilla/source/js/narcissus/.

[40] Novark, G., and Berger, E. D. DieHarder: securing the heap. In *CCS* (2010).

[41] Perdisci, R., Dagon, D., Lee, W., Fogla, P., and Sharif, M. Misleading worm signature generators using deliberate noise injection. In *In Proceedings of the 2006 IEEE Symposium on Security and Privacy* (2006).

[42] Provos, N., Mavrommatis, P., Rajab, M. A., and Monrose, F. All your iframes point to us. In *17th USENIX Security Symposium* (2008).

[43] Ratanaworabhan, P., Livshits, B., and Zorn, B. Nozzle: A defense against heap-spraying code injection attacks. In *18th USENIX Security Symposium* (2009).

[44] Reis, C., Dunagan, J., Wang, H. J., Dubrovsky, O., and Esmeir, S. Browsershield: vulnerability-driven filtering of dynamic html. In *OSDI* (2006).

[45] Reis, C., and Gribble, S. D. Isolating web programs in modern browser architectures. In *EuroSys* (2009).

[46] Rieck, K., Krueger, T., and Dewald, A. Cujo: efficient detection and prevention of drive-by-download attacks. In *ACSAC* (2010).

[47] Seifert, C., Steenson, R., Holz, T., Yuan, B., , and Davis, M. A. Know your enemy: Malicious web servers. http://www.honeynet.org/papers/mws/.

[48] Seifert, C., Welch, I., and Komisarczuk, P. Identification of malicious web pages with static heuristics. In *ATNAC: Telecommunication Networks and Applications Conference* (2008).

[49] Seifert, C., Welch, I., Komisarczuk, P., Aval, C. U., and Endicott-Popovsky, B. Identification of malicious web pages through analysis of underlying dns and web server relationships. In *LCN: Local Computer Networks* (2008).

[50] Song, C., Zhuge, J., Han, X., and Ye, Z. Preventing drive-by download via inter-module communication monitoring. In *ASIACCS* (2010).

[51] Tang, S., Mai, H., and King, S. T. Trust and protection in the illinois browser operating system. In *OSDI* (2010).

[52] Ter Louw, M., and Venkatakrishnan, V. Blueprint: Precise browser-neutral prevention of cross-site scripting attacks. In *30th IEEE Symposium on Security and Privacy* (2009).

[53] 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 *18th Usenix Security Symposium* (2009).

[54] Wang, H. J., Guo, C., Simon, D. R., and Zugenmaier, A. Shield: vulnerability-driven network filters for preventing known vulnerability exploits. In *SIGCOMM* (2004).

[55] Wang, Y., Beck, D., Jiang, X., and Roussev, R. Automated web patrol with strider honeymonkeys: Finding web sites that exploit browser vulnerabilities. In *NDSS* (2006).