

USENIX Association

**Proceedings of the
22nd USENIX Security Symposium**

**August 14–16, 2013
Washington, D.C.**

Conference Organizers

Program Chair

Sam King, *University of Illinois and Adrenaline Mobility*

Program Committee

Nikita Borisov, *University of Illinois*

Elie Bursztein, *Google*

Srdjan Capkun, *ETH Zurich*

Shuo Chen, *Microsoft Research*

Sonia Chiasson, *Carleton University*

Adam Chlipala, *Massachusetts Institute of Technology*

William Enck, *North Carolina State University*

Adrienne Porter Felt, *Google*

Kevin Fu, *University of Michigan*

Roxana Geambasu, *Columbia University*

Ian Goldberg, *University of Waterloo*

Matthew Green, *John Hopkins University*

Chris Grier, *University of California, Berkeley*

Thorsten Holz, *Ruhr-Universität Bochum*

Jaeyeon Jung, *Microsoft Research*

Benjamin Livshits, *Microsoft Research*

Jonathan McCune, *Google*

Fabian Monrose, *University of North Carolina, Chapel Hill*

Niels Provos, *Google*

Prateek Saxena, *National University of Singapore*

Stuart Schechter, *Microsoft Research*

Hovav Shacham, *University of California, San Diego*

Micah Sherr, *Georgetown University*

Elaine Shi, *University of Maryland, College Park*

Cynthia Sturton, *University of California, Berkeley*

Shuo Tang, *University of Illinois and Adrenaline Mobility*

Patrick Traynor, *Georgia Institute of Technology*

David Wagner, *University of California, Berkeley*

Tara Whalen, *Carleton University*

Michal Zalewski, *Google*

Nickolai Zeldovich, *Massachusetts Institute of Technology*

Invited Talks Chair

Michael Bailey, *University of Michigan*

Invited Talks Committee

Elie Bursztein, *Google*

Wenke Lee, *Georgia Institute of Technology*

Stefan Savage, *University of California, San Diego*

Poster Session Coordinator

William Enck, *North Carolina State University*

Rump Session Chair

Nikita Borisov, *University of Illinois*

Steering Committee

Matt Blaze, *University of Pennsylvania*

Dan Boneh, *Stanford University*

Casey Henderson, *USENIX*

Tadayoshi Kohno, *University of Washington*

Fabian Monrose, *University of North Carolina, Chapel Hill*

Niels Provos, *Google*

David Wagner, *University of California, Berkeley*

External Reviewers

Moheeb Abu Rajab

Robert Gawlik

Alex Moshchuk

Jay Stokes

Devdatta Akhawe

Matthew Hicks

Kurt Thomas

Chaitrali Amrutkar

Pieter Hooimeijer

Sebastian Uellenbeck

Kevin Butler

Ling Huang

Joel Weinberger

Henry Carter

Rob Jansen

Andrew White

Chang Ee Chien

Shrinivas Krishnan

Yinglian Xie

Shane S. Clark

Ben Leong

Fang Yu

Manuel Costa

Charles Lever

Apostolis Zarras

Thurston Dang

Wenchao Li

Yulong Zhang

Lucas Davi

Zhenkai Liang

Thomas Zimmermann

Murph Finnicum

Nicolás Lidzborski

Kevin Z. Snow

Matt Fredrikson

Haohui Mai

Emil Stefanov

22nd USENIX Security Symposium
August 14–16, 2013
Washington, D.C.

Message from the Program Chair..... vii

Wednesday, August 14, 2013

Network Security

- Greystar: Fast and Accurate Detection of SMS Spam Numbers in Large Cellular Networks using Grey Phone Space** 1
Nan Jiang, University of Minnesota; Yu Jin and Ann Skudlark, AT&T Labs; Zhi-Li Zhang, University of Minnesota

- Practical Comprehensive Bounds on Surreptitious Communication Over DNS** 17
Vern Paxson, University of California, Berkeley, and International Computer Science Institute; Mihai Christodorescu, Qualcomm Research; Mobin Javed, University of California, Berkeley; Josyula Rao, Reiner Sailer, Douglas Lee Schales, and Marc Ph. Stoecklin, IBM Research; Kurt Thomas, University of California, Berkeley; Wietse Venema, IBM Research; Nicholas Weaver, ICSI and University of California, San Diego

- Let Me Answer That For You: Exploiting Broadcast Information in Cellular Networks.....** 33
Nico Golde, Kévin Redon, and Jean-Pierre Seifert, Deutsche Technische Universität Berlin and Telekom Innovation Laboratories

Potpourri

- Dowsing for Overflows: A Guided Fuzzer to Find Buffer Boundary Violations** 49
Istvan Haller and Asia Slowinska, VU University Amsterdam; Matthias Neugschwandtner, Vienna University of Technology; Herbert Bos, VU University Amsterdam

- MetaSymplot: Day-One Defense against Script-based Attacks with Security-Enhanced Symbolic Analysis** 65
Ruowen Wang, Peng Ning, Tao Xie, and Quan Chen, North Carolina State University

- Towards Automatic Software Lineage Inference.....** 81
Jiyong Jang, Maverick Woo, and David Brumley, Carnegie Mellon University

Mobile Security I

- Securing Embedded User Interfaces: Android and Beyond.....** 97
Franziska Roesner and Tadayoshi Kohno, University of Washington

- Automatic Mediation of Privacy-Sensitive Resource Access in Smartphone Applications** 113
Benjamin Livshits and Jaeyeon Jung, Microsoft Research

- Flexible and Fine-grained Mandatory Access Control on Android for Diverse Security and Privacy Policies** 131
Sven Bugiel, Saarland University; Stephan Heuser, Fraunhofer SIT; Ahmad-Reza Sadeghi, Technische Universität Darmstadt and Center for Advanced Security Research Darmstadt

(Wednesday, August 14, continues on p. iv)

Applied Crypto I

Proactively Accountable Anonymous Messaging in Verdict	147
Henry Corrigan-Gibbs, David Isaac Wolinsky, and Bryan Ford, <i>Yale University</i>	
ZQL: A Compiler for Privacy-Preserving Data Processing.....	163
Cédric Fournet, Markulf Kohlweiss, and George Danezis, <i>Microsoft Research; Zhengqin Luo, MSR-INRIA Joint Centre</i>	
DupLESS: Server-Aided Encryption for Deduplicated Storage	179
Mihir Bellare and Sriram Keelveedhi, <i>University of California, San Diego</i> ; Thomas Ristenpart, <i>University of Wisconsin-Madison</i>	

Large-Scale Systems Security I

Trafficking Fraudulent Accounts: The Role of the Underground Market in Twitter Spam and Abuse	195
Kurt Thomas, <i>University of California, Berkeley, and Twitter</i> ; Damon McCoy, <i>George Mason University</i> ; Chris Grier, <i>University of California, Berkeley, and International Computer Science Institute</i> ; Alek Kolcz, <i>Twitter</i> ; Vern Paxson, <i>University of California, Berkeley, and International Computer Science Institute</i>	
Impression Fraud in On-line Advertising via Pay-Per-View Networks.....	211
Kevin Springborn, <i>Broadcast Interactive Media</i> ; Paul Barford, <i>Broadcast Interactive Media and University of Wisconsin-Madison</i>	
The Velocity of Censorship: High-Fidelity Detection of Microblog Post Deletions	227
Tao Zhu, <i>Independent Researcher</i> ; David Phipps, <i>Bowdoin College</i> ; Adam Pridgen, <i>Rice University</i> ; Jedidiah R. Crandall, <i>University of New Mexico</i> ; Dan S. Wallach, <i>Rice University</i>	

Thursday, August 15, 2013

Large-Scale Systems Security II

You Are How You Click: Clickstream Analysis for Sybil Detection	241
Gang Wang and Tristan Konolige, <i>University of California, Santa Barbara</i> ; Christo Wilson, <i>Northeastern University</i> ; Xiao Wang, <i>Renren Inc.</i> ; Haitao Zheng and Ben Y. Zhao, <i>University of California, Santa Barbara</i>	
Alice in Warningland: A Large-Scale Field Study of Browser Security Warning Effectiveness	257
Devdatta Akhawe, <i>University of California, Berkeley</i> ; Adrienne Porter Felt, <i>Google, Inc.</i>	
An Empirical Study of Vulnerability Rewards Programs	273
Matthew Finifter, Devdatta Akhawe, and David Wagner, <i>University of California, Berkeley</i>	

Applied Crypto II

Secure Outsourced Garbled Circuit Evaluation for Mobile Devices.....	289
Henry Carter, <i>Georgia Institute of Technology</i> ; Benjamin Mood, <i>University of Oregon</i> ; Patrick Traynor, <i>Georgia Institute of Technology</i> ; Kevin Butler, <i>University of Oregon</i>	
On the Security of RC4 in TLS	305
Nadhem AlFardan, <i>Royal Holloway, University of London</i> ; Daniel J. Bernstein, <i>University of Illinois at Chicago and Technische Universiteit Eindhoven</i> ; Kenneth G. Paterson, Bertram Poettering, and Jacob C.N. Schuldt, <i>Royal Holloway, University of London</i>	
PCF: A Portable Circuit Format for Scalable Two-Party Secure Computation	321
Ben Kreuter, <i>University of Virginia</i> ; Benjamin Mood, <i>University of Oregon</i> ; abhi shelat, <i>University of Virginia</i> ; Kevin Butler, <i>University of Oregon</i>	

Protecting and Understanding Binaries

Control Flow Integrity for COTS Binaries337
Mingwei Zhang and R. Sekar, <i>Stony Brook University</i>	
Native x86 Decompilation Using Semantics-Preserving Structural Analysis and Iterative Control-Flow Structuring353
Edward J. Schwartz, <i>Carnegie Mellon University</i> ; JongHyup Lee, <i>Korea National University of Transportation</i> ; Maverick Woo and David Brumley, <i>Carnegie Mellon University</i>	
Strato: A Retargetable Framework for Low-Level Inlined-Reference Monitors369
Bin Zeng and Gang Tan, <i>Lehigh University</i> ; Úlfar Erlingsson, <i>Google Inc.</i>	

Current and Future Systems Security

On the Security of Picture Gesture Authentication383
Ziming Zhao and Gail-Joon Ahn, <i>Arizona State University and GFS Technology, Inc.</i> ; Jeong-Jin Seo, <i>Arizona State University</i> ; Hongxin Hu, <i>Delaware State University</i>	
Explicating SDKs: Uncovering Assumptions Underlying Secure Authentication and Authorization399
Rui Wang, <i>Microsoft Research Redmond</i> ; Yuchen Zhou, <i>University of Virginia</i> ; Shuo Chen and Shaz Qadeer, <i>Microsoft Research Redmond</i> ; David Evans, <i>University of Virginia</i> ; Yuri Gurevich, <i>Microsoft Research Redmond</i>	
Enabling Fine-Grained Permissions for Augmented Reality Applications with Recognizers415
Suman Jana, <i>The University of Texas at Austin</i> ; David Molnar and Alexander Moshchuk, <i>Microsoft Research</i> ; Alan Dunn, <i>The University of Texas at Austin</i> ; Benjamin Livshits, Helen J. Wang, and Eyal Ofek, <i>Microsoft Research</i>	

Hardware and Embedded Security I

CacheAudit: A Tool for the Static Analysis of Cache Side Channels431
Goran Doychev, <i>IMDEA Software Institute</i> ; Dominik Feld, <i>Saarland University</i> ; Boris Köpf and Laurent Mauborgne, <i>IMDEA Software Institute</i> ; Jan Reineke, <i>Saarland University</i>	
Transparent ROP Exploit Mitigation Using Indirect Branch Tracing447
Vasilis Pappas, Michalis Polychronakis, and Angelos D. Keromytis, <i>Columbia University</i>	
FIE on Firmware: Finding Vulnerabilities in Embedded Systems using Symbolic Execution463
Drew Davidson, Benjamin Moench, Somesh Jha, and Thomas Ristenpart, <i>University of Wisconsin—Madison</i>	

Friday, August 16, 2013

Hardware and Embedded Security II

Sancus: Low-cost Trustworthy Extensible Networked Devices with a Zero-software Trusted Computing Base479
Job Noorman, Pieter Agten, Wilfried Daniels, Raoul Strackx, Anthony Van Herrewege, Christophe Huygens, Bart Preneel, Ingrid Verbauwheide, and Frank Piessens, <i>KU Leuven</i>	
Securing Computer Hardware Using 3D Integrated Circuit (IC) Technology and Split Manufacturing for Obfuscation495
Frank Imeson, Arij Emtenan, Siddharth Garg, and Mahesh V. Tripunitara, <i>University of Waterloo</i>	
KI-Mon: A Hardware-assisted Event-triggered Monitoring Platform for Mutable Kernel Object511
Hojoon Lee, <i>Korea Advanced Institute of Science and Technology (KAIST)</i> ; HyunGon Moon, <i>Seoul National University</i> ; DaeHee Jang and Kiwhan Kim, <i>Korea Advanced Institute of Science and Technology (KAIST)</i> ; Jihoon Lee and Yunheung Paek, <i>Seoul National University</i> ; Brent ByungHoon Kang, <i>Korea Advanced Institute of Science and Technology (KAIST)</i>	

(Friday, August 16, continues on p. vi)

Mobile Security II

WHYPER: Towards Automating Risk Assessment of Mobile Applications527
Rahul Pandita, Xusheng Xiao, Wei Yang, William Enck, and Tao Xie, <i>North Carolina State University</i>	
Effective Inter-Component Communication Mapping in Android with Epicc: An Essential Step Towards Holistic Security Analysis543
Damien Oteau and Patrick McDaniel, <i>Pennsylvania State University</i> ; Somesh Jha, <i>University of Wisconsin</i> ; Alexandre Bartel, <i>University of Luxembourg</i> ; Eric Bodden, <i>Technische Universität Darmstadt</i> ; Jacques Klein and Yves Le Traon, <i>University of Luxembourg</i>	
Jekyll on iOS: When Benign Apps Become Evil559
Tielei Wang, Kangjie Lu, Long Lu, Simon Chung, and Wenke Lee, <i>Georgia Institute of Technology</i>	

Large Scale Systems Security III

Measuring the Practical Impact of DNSSEC Deployment.....	.573
Wilson Lian, <i>University of California, San Diego</i> ; Eric Rescorla, <i>RTFM, Inc.</i> ; Hovav Shacham and Stefan Savage, <i>University of California, San Diego</i>	
ExecScent: Mining for New C&C Domains in Live Networks with Adaptive Control Protocol Templates589
Terry Nelms, <i>Damballa, Inc. and Georgia Institute of Technology</i> ; Roberto Perdisci, <i>University of Georgia and Georgia Institute of Technology</i> ; Mustaque Ahamed, <i>Georgia Institute of Technology and New York University Abu Dhabi</i>	
ZMap: Fast Internet-wide Scanning and Its Security Applications605
Zakir Durumeric, Eric Wustrow, and J. Alex Halderman, <i>University of Michigan</i>	

Web Security

Eradicating DNS Rebinding with the Extended Same-Origin Policy621
Martin Johns and Sebastian Lekies, <i>SAP Research</i> ; Ben Stock, <i>Friedrich-Alexander-Universität Erlangen-Nürnberg</i>	
Revolver: An Automated Approach to the Detection of Evasive Web-based Malware637
Alexandros Kapravelos and Yan Shoshitaishvili, <i>University of California, Santa Barbara</i> ; Marco Cova, <i>University of Birmingham</i> ; Christopher Kruegel and Giovanni Vigna, <i>University of California, Santa Barbara</i>	
Language-based Defenses against Untrusted Browser Origins653
Karthikeyan Bhargavan and Antoine Delignat-Lavaud, <i>INRIA Paris-Rocquencourt</i> ; Sergio Maffeis, <i>Imperial College London</i>	

Attacks

Take This Personally: Pollution Attacks on Personalized Services671
Xinyu Xing, Wei Meng, and Dan Doozan, <i>Georgia Institute of Technology</i> ; Alex C. Snoeren, <i>University of California, San Diego</i> ; Nick Feamster and Wenke Lee, <i>Georgia Institute of Technology</i>	
Steal This Movie: Automatically Bypassing DRM Protection in Streaming Media Services.....	.687
Ruoyu Wang, <i>University of California, Santa Barbara and Tsinghua University</i> ; Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna, <i>University of California, Santa Barbara</i>	

Message from the 22nd USENIX Security Symposium Program Chair

It is my great pleasure to welcome you to the 22nd USENIX Security Symposium. We have an outstanding event in store for you, and for that, I thank all of you—the authors, the invited speakers, the program committee members and other organizers, the external reviewers, the sponsors, the USENIX staff, and the attendees. The USENIX Security Symposium would not be the premier venue that it is if it were not for your involvement.

This year, USENIX Security received 277 submissions. As in previous years, the program committee used a multi-round reviewing process. The authors of submissions were not revealed to the reviewers, and every paper was reviewed by at least two reviewers. Papers that received a positive score in the first round were reviewed by one to four additional reviewers. The program committee met to discuss the submissions in April at the Microsoft Research campus in Redmond, Washington. Jaeyeon Jung at Microsoft Research devoted a huge amount of time to ensure that all aspects of the meeting ran smoothly; I am deeply grateful to her for all of her hard work as host. I would also like to thank Microsoft Research and USENIX for funding the meals during the PC meeting.

After very careful and extensive deliberations, the program committee decided to accept or conditionally accept 44 papers—a record for USENIX Security. The quality of these papers is very high—a testimony to the strength of our community!

The entire program committee invested a tremendous effort in reviewing and discussing these papers. Please join me in thanking the program committee and all the external reviewers, listed on page ii, for their countless hours of work. I would also like to thank Sonia Chiasson for serving as deputy chair and handling the submissions for which I had a conflict.

We also have a wonderful selection of invited talks for you. I would like to thank the invited talks committee—Michael Bailey, Elie Bursztein, Wenke Lee, and Stefan Savage—for all of the hard work they invested toward ensuring an exciting, interesting, educational, and invigorating invited talks track. The Poster and Rump Sessions have also been hits at previous USENIX Security Symposia, and I think you will find them to be “can’t miss” events at this year’s USENIX Security too. I would like to thank Will Enck for serving as this year’s Poster Session Chair, and Nikita Borisov for serving as this year’s Rump Session Chair.

I am also deeply grateful to the entire staff at USENIX. They have worked incredibly hard to help make USENIX Security one of the top conferences in the field. Please join me in thanking them. Please also join me in thanking Joseph Schwartz for capturing USENIX Security on video.

Finally, I would like to thank all of the authors who submitted their research papers, posters, and Rump Session talks.

Welcome to Washington, DC for the 22nd USENIX Security Symposium! I hope you enjoy the conference.

Sam King, University of Illinois and Adrenaline Mobility
USENIX Security '13 Program Chair

Greystar: Fast and Accurate Detection of SMS Spam Numbers in Large Cellular Networks using Grey Phone Space

Nan Jiang
University of Minnesota

Yu Jin
AT&T Labs

Ann Skudlark
AT&T Labs

Zhi-Li Zhang
University of Minnesota

Abstract

In this paper, we present the design of Greystar, an innovative defense system for combating the growing SMS spam traffic in cellular networks. By exploiting the fact that most SMS spammers select targets randomly from the finite phone number space, Greystar monitors phone numbers from the grey phone space (which are associated with data only devices like laptop data cards and machine-to-machine communication devices like electricity meters) and employs a novel statistical model to detect spam numbers based on their footprints on the grey phone space. Evaluation using five month SMS call detail records from a large US cellular carrier shows that Greystar can detect thousands of spam numbers each month with very few false alarms and 15% of the detected spam numbers have never been reported by spam recipients. Moreover, Greystar is much faster in detecting SMS spam than existing victim spam reports, reducing spam traffic by 75% during peak hours.

1 Introduction

The explosion of mobile devices in the past decade has brought with it an onslaught of unwanted SMS (Short Message Service) spam [1]. It has been reported that the number of spam messages in the US has risen 45% in 2011 to 4.5 billion messages [2]. In 2012, there were 350K variants of SMS spam messages accounted for globally [3] and more than 69% of the mobile users claimed to have received text spam [4]. The sheer volume of spam messages not only inflicts an annoying user experience, but also incur significant costs to both cellular service providers and customers alike. In contrast to email spam where the number of possible email addresses is unlimited - and therefore the spammer generally needs a seed list beforehand, SMS spammers can more easily reach victims by, e.g., simply enumerating all numbers from the *finite* phone number space. This,

combined with wide adoption of mobile phones, makes SMS a medium of choice among spammers. Furthermore, the increasingly rich functionality provided by smart mobile devices also enables spammers to carry out more sophisticated attacks through both voice and data channels, for example, using SMS spam to entice users to visit certain websites for product advertisement or other illicit activities.

Because SMS spam inflicts financial loss to mobile users and adverse impact to cellular network performance, the objective of defense techniques is to restrict spam numbers quickly before they reach too many victims. To this end, instead of applying popular solutions in controlling email spam (e.g., filtering based on sending patterns), which can cause a high false alarm rate, cellular carriers often seek help from their customers to alert them of emerging spamming activities. More specifically, cellular carriers deploy reporting mechanism for spam victims to report received spam messages and then examine and restrict these reported spam numbers accordingly. Such spam detection techniques using victim spam reports are very accurate, thanks to the human intelligence added while submitting these reports. However, these methods can suffer from significant delay due to the low report rate and slow user responses, rendering them inefficient in controlling SMS spam.

To address the issues in existing solutions, in this paper, we carry out extensive analysis of SMS spamming activities using five months of SMS call detail records collected from a large cellular network in the US and the SMS spam messages reported from the spam recipients to that cellular carrier. We find that a majority of spammers choose targets randomly from a few area codes or the entire phone number space, and initiate spam traffic at high rates. To detect such aggressive random spammers, we advance a novel notion of *grey phone space*. Grey phone space comprises a collection of *grey phone numbers* (or grey numbers in short). Grey numbers are associated with two types of mobile devices: data only

devices (e.g., many laptop data cards and data modems, etc.) and machine-to-machine (M2M) communication devices (e.g., utility meters and medical devices, etc.). These grey numbers usually do not participate actively in SMS communication as other mobile numbers do (e.g., those associated with smartphones), they thereby form a grey territory that legitimate mobile users rarely enter. In the mean time, the wide dispersion of grey numbers makes them hard to be evaded by spammers who choose targets randomly.

On top of grey phone space, we propose the design of *Greystar*. Greystar employs a novel statistical model to detect spam numbers based on their interactions with grey numbers and other non-grey phone numbers. We evaluate Greystar using five months of SMS call records. Experimental results indicate that Greystar is superior to the existing SMS spam detection algorithms, which rely heavily on victim spam reports, in terms of both accuracy and detection speed. In particular, Greystar detected over 34K spam numbers in five months while only generating two false positives. In addition, more than 15% of the detected spam numbers have never been reported by mobile users. Moreover, Greystar reacts fast to emerging spamming activities, with a median detection time of 1.2 hours after spamming activities occur. In 50% of the cases, Greystar is at least 1 day ahead of victim spam reports. The high accuracy and fast response time allow us to restrict more spam numbers soon after spamming activities emerge, and hence to reduce a majority of the spam messages in the network. We demonstrate through simulation on real network data that, after deploying Greystar, we can reduce 75% of the spam messages during peak hours. In this way, Greystar can greatly benefit the cellular carriers by alleviating the load from aggressive SMS spam messages on network resources as well as limiting their adverse impact on legitimate mobile users.

The remainder of this paper is organized as follows. We introduce the SMS architecture and the datasets used in our study in Section 2. We then motivate the design of Greystar in Section 3. In Section 4 we study the SMS activities of spammers and legitimate users. The definition of grey numbers is presented in Section 5. In Section 6, we explain in detail the design of Greystar. Evaluation results are presented in Section 7. Section 8 discusses the related work and Section 9 concludes the paper.

2 Background and Datasets

In this section, we briefly describe the cellular network focused in our study. We then introduce the datasets and our ground truth for identifying spam phone numbers.

2.1 SMS Architecture in UMTS

The cellular network under study utilizes primarily UMTS (Universal Mobile Telecommunication System), a popular 3G mobile communication technology adopted by many mobile carriers across the globe. Here we introduce the architecture for delivering SMS messages inside UMTS networks (for other aspects regarding UMTS networks, e.g., mobile data channels, see [5]). Fig. 1 depicts a schematic view of the architecture. When sending an SMS message, an end user equipment (UE_A) directly communicates with a cell tower (or node-B), which forwards the message to a Radio Network Controller (RNC). The RNC then delivers the message to a Mobile Switching Center (MSC) server, where the message enters the Signaling System 7 (SS7) network and is stored temporarily at a Short Message Service Center (SMSC). From the SMSC, the message will be routed to the serving MSC of the recipient (UE_B), then to the serving RNC and Node-B, and finally reaches UE_B . Similarly, messages originated from other carrier networks (e.g., from UE_C) will also traverse the SS7 network and bypass the serving MSC before arriving at UE_B ¹.

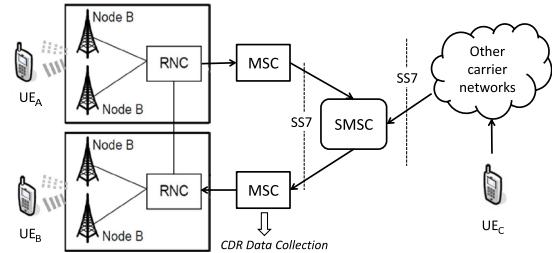


Figure 1: SMS architecture in UMTS networks.

2.2 Datasets

In this paper, we use two different datasets for our study.

SMS Call Detail Records (CDRs) are used for understanding SMS user/spammer activities and evaluating the performance of the proposed Greystar system. These records were collected at the serving MSC’s of SMS recipients (see Fig. 1). This means that CDR records represent SMS messages targeting registered mobile customers of the UMTS network under study² and have been

¹Note that similar SMS architecture is also adopted in other types of 3G/4G cellular networks. Additionally, in this paper, we only focus on SMS through the voice control channel. Short message services through mobile data channels, such as iMessage, Tweets and MMS, etc., are out of the scope of this paper (though defenses for fighting email spam can be applied to detect short message spam through data channels, which we shall discuss in Section 8).

²SMS messages targeting mobile users in other carrier networks and landline numbers are not seen at the serving MSCs and hence are not

successfully routed through the SS7 network. The CDR dataset spans 5 months from Jan 2012 to May 2012. Each record contains the SMS receiving time, the originating number, the terminating number and the International Mobile Equipment Identity (IMEI) for the device associated with the terminating number³. We note that CDR records do not contain text content of the original SMS messages.

Victim spam reports contain spam messages reported by spam recipients to the carrier. The said cellular carrier deploys an SMS spam reporting service for its users: when a user receives an SMS text and deems it as a spam message, s/he can forward the message to a *spam report number* designated by the cellular service provider. Once the spam is forwarded, an acknowledgment message is returned, which asks the user to reply with the spammer’s phone number (referred to as the *spam number* hereafter). Once the above two-stage process is completed within a predefined time interval, a spam report is created, which includes the reporter’s phone number, the spam number, the reporting time and the text content of the reported spam message. We employ six months of spam reports from Jan 2012 to June 2012 in order to cover spam numbers observed between Jan and May but are reported after May due to the delay of the spam reports (see Section 3.2).

We emphasize that no customer personal information was collected or used for our study. All customer identifies were *anonymized* before any analysis was conducted. In particular, for phone numbers, only the area code (i.e., the first 3 digits of the 10 digit North American numbers) was used and the remaining digits were hashed. Similarly we only retain the first 8-digit Type Allocation Code (TAC) of the IMEI to identify device types and anonymize the remaining 8-digit to preserve customers’ privacy. In addition, to adhere to the confidentiality under which we have access to the data, in places we only present normalized views of our results while retaining the scientifically relevant magnitudes.

2.3 Obtaining Ground Truth

Although victim spam reports provide us with ground truth for some spam numbers, they are by no means comprehensive and can be noisy (see Section 3.2). Therefore,

included in CDR records.

³IMEI’s are stored at MSC’s and are updated every time users connect to the network. Although we have observed that spammers sometimes modify the IMEIs of their spamming devices (e.g., through special equipment like SIM boxes), IMEI spoofing among legitimate users is rare. Therefore we can reliably identify the types of user devices based on their corresponding IMEIs. Meanwhile, since all the CDRs are collected at MSCs, we can identify the original phone numbers that initiate the SMS messages. Hence our approach is not affected even when spammers employ spoofing techniques to change their caller IDs.

in this paper, we employ a more reliable source of ground truth. In particular, we request the fraud agents from the said UMTS carrier to manually verify spam number candidates detected by us. These fraud agents are exposed to much richer (and more expensive) sources of information. For example, fraud agents can investigate the ownership and the price plan information of the candidates, examine their SMS sending patterns and correlate them with known spam numbers in terms of their network locations and active times, etc. The final decision is made conservatively by corroborating different evidence.

Admittedly, fraud agents can make mistakes during their investigation. Meanwhile, their breadth may be limited by not being able to inspect all mobile numbers in the network. Nevertheless, fraud agents provide us with the most authoritative ground truth available for our study. It is worth mentioning that such investigation by fraud agents has been deployed independently for SMS spam number detection and restriction for more than one year and no false alarm has yet been observed (e.g., no user complaint is observed so far regarding incorrectly restricted phone numbers). Therefore, in our study, we will treat fraud agents as a black box authority, i.e., we submit a list of spam number candidates to fraud agents and they return a list of confirmed spam numbers.

3 Objectives and Existing Solutions

In this section, we discuss the objectives of developing an effective defense against SMS spam by comparing the difference between SMS spam and traditional email spam. We then review the most widely adopted SMS spam detection method based on crowdsourcing victim spam reports and point out its inefficacy. In the end, we present the rationale of the proposed Greystar system.

3.1 SMS Spam Defense Objectives

In a conventional SMS spamming scenario, an SMS spammer (note that we refer to an *SMS spammer* as the person who employs a set of spam numbers to launch SMS spam campaigns) first invests in a set of phone numbers and special high-speed devices, such as 3G modems and SIM boxes [6]. Using these devices, s/he then initiates unsolicited SMS messages to a large number of mobile phone numbers. Akin to traditional email spam, the objective of SMS spam is to advertise certain information to entice further actions from the message recipients, e.g., calling a fraud number or clicking on a URL link embedded in the message which points to a malicious site. However, SMS spamming activities exhibit unique characteristics which shift the focus of the defense mechanisms and hence render inapplicable or

inefficient existing solutions for defending against traditional email spam.

Email service providers usually detect and filter email spam at their mail servers, to which they have full access. There they can build accurate spam filters by exploiting rich features in emails including the text content. Spam filters at end user devices are also a common choice, where email clients (apps) filter spam while retrieving emails from remote mail servers. Though blacklist of email spammers are sometimes used to assist spam classification [7–9], restricting email spam senders is usually not the main focus of the defense, since it requires close collaboration between email providers and network carriers. Moreover, it is observed that many spam emails are originated from legitimate hosts due to botnet activities [10], which makes restricting spam originators an inapplicable solution.

In comparison to emails which are generally stored on servers and wait for users to retrieve them, SMS messages are delivered instantly to the recipients through the SS7 network. Along the path, SMS messages are only cached temporarily at SMSC (only when the recipients are offline), leaving little time for cellular carriers to react to them. The task becomes even more challenging especially when the SMS traffic volume peaks during busy hours. Filtering SMS spam at end user devices (e.g., using mobile apps) is also not applicable given many SMS capable devices (e.g., feature phones) do not support running such apps. In addition, for a user with a pay-per-use SMS plan, she is already charged for the spam message once it arrives at her device. More importantly, even when SMS spam filters are deployed at SMSC’s and end user devices, SMS spammers can still inflict significant loss to the carrier and other mobile users. This is because the huge number of spam messages can lead to a significant increase in the SMS traffic volume at the cell towers serving the spam senders, possibly causing congestion and hence deteriorating voice/data usage experience of nearby users. For example, we have found the SMS traffic volume at cell towers can easily get multiplied by more than 10 times due to the activities of spammers. Therefore, the focus of the SMS spam defense is to *control spam numbers as soon as possible before they reach a large number of victims*.

An efficient SMS spam detection algorithm is hence expected to react quickly to emerging spamming activities. Meanwhile, the focus on restricting spam numbers places a strong emphasis on the accuracy of the algorithm. First, it requires a spam detection algorithm to limit false alarms, because false alarms can lead to incorrect restriction of legitimate users from accessing SMS services. Second, it demands the algorithm detect as many spam numbers as possible so as to minimize the impact of SMS spam activities on the network. Such

high accuracy requirements are hard to achieve solely based on the SMS sending patterns of the spammers. For example, it is difficult to separate spam campaigns from legitimate SMS campaigns, such as a school sending messages to its students to alert adverse weather conditions. These legitimate senders can exhibit characteristics that are common to SMS spammers⁴. Spammers may also alter their sending patterns to mimic legitimate users to avoid detection. As a result, cellular carriers often seek the assistance from their customers to alert them of emerging SMS spam activities.

3.2 Spam Detection by Crowdsourcing Victim Spam Reports

The emphasis on high accuracy gives rise to the wide adoption of spam detection methods based on victim spam reports which were introduced in Section 2. Victim spam reports represent a more reliable and cleaner source of SMS spam samples, as all the spam messages contained in the reports have been vetted and classified by mobile users (using human intelligence). To further mitigate the possible errors caused during the two-step reporting process, cellular carriers often crowdsource spam reports from different users. For example, a simple yet effective strategy is to identify a spam number after receiving reports from K distinct users. Meanwhile, defense mechanisms based on victim spam reports are also of low cost, because only numbers reported by users need to be further analyzed. Due to this reason, spam reports are usually a trigger for more sophisticated investigation on the senders, such as their sending patterns, service plans, etc..

Despite the high accuracy and low cost, detecting SMS spam based on spam reports is analogous to performing spam filtering at user devices. The major drawback is detection delay, which we illustrate in Fig. 2 based on the CDR data from January 2012. The red solid curve in Fig. 2 measures how long it takes for a spam number to be reported after spam starts (a.k.a. *report delay*). We consider a spam number starts spamming when it first reaches at least 50 victims in an hour (see Section 4 for discussion on spamming rates). From Fig. 2, we observe that less than 3% of the spam numbers are reported within 1 hour after spamming starts. More than 50% of the spam numbers are reported 1 day after. The report delay is mainly due to the extremely low report rate from users. In fact, less than 1 in 10,000 spam messages were

⁴Maintaining a whitelist of such legitimate intensive SMS users can be challenging. First, we have little information to identify the white list if the users are outside the network. Second, even for the users inside the network, the whitelist can still be dynamic, with new businesses/organizations initiating/stopping SMS broadcasting services every day. More importantly, users are not obliged to report to the carrier when they intend to start such services.

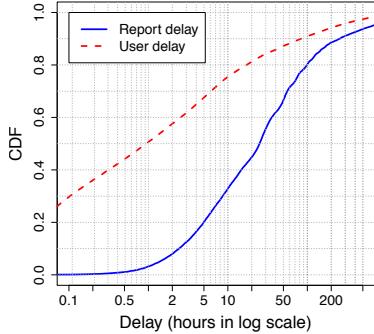


Figure 2: Lags of user reports.

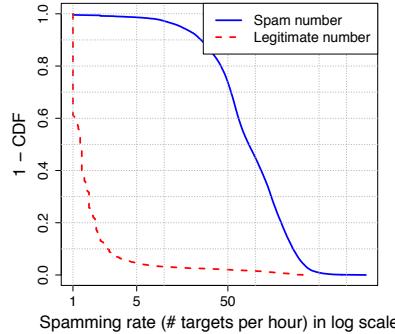


Figure 3: Spammer rate.

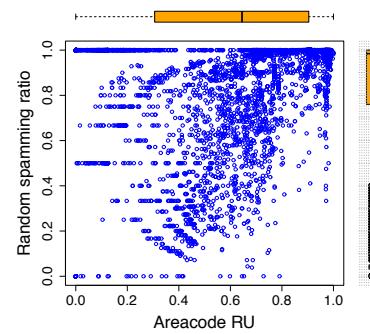


Figure 4: Target selection strategies.

reported during the five month observation period. Aside from causing a long detection delay, the low report rate also leads to many missed detections (see Section 7).

In addition, even when a victim reports a spam message, how long it takes him/her is at the reporter’s discretion. The blue dotted curve in Fig. 2 shows how fast a user reports a spam message after receiving it (*user delay*). Note that each user can receive multiple spam messages (possibly with different text content) from the same sender and hence can report the same sender multiple times. Thus, we define *user delay* as the time difference between when a user reports a spam message and the *last* time that the user receives spam from that particular spam number before the report. We observe in Fig. 2 , among users who report spam, half of the spam messages are reported more than 1 hour after they receive the spam messages. Around 20% spam are reported even after a day. Due to such a long delay, spammers have already inflicted significant loss to the network and its customers.

In addition to the problem of detection delay, the current two-stage reporting method is error-prone. We find around 10% reporters fail to provide a valid spam number at the second stage. Moreover, spam report based methods are vulnerable to attacks, as attackers can easily game with the detection system by sending bogus reports to Denial-of-Service (DoS) legitimate numbers. All these drawbacks render spam detection using victim spam reports an insufficient solution.

3.3 Overview of Greystar

Recognizing the drawbacks of existing victim report based solutions, we introduce the rationale behind Greystar. The objective of Greystar is to accurately detect SMS spam while at the same time being able to control spam numbers as soon as possible before they reach too many victims. To this end, we advance a novel notion of grey phone numbers. These grey numbers usually do

not communicate with other mobile numbers using SMS, they thereby form a grey territory that legitimate mobile users rarely enter. On the other hand, as we shall see in Section 4, it is difficult for spammers to avoid touching these grey numbers due to the random target selection strategies that they usually adopt. Greystar then passively monitors the footprints of SMS senders on these grey numbers to detect impending spam activities targeting a large number of mobile users.

Greystar addresses the problems in existing spam report based solutions as follows. First, the population of grey numbers is much larger and widely distributed (see Section 5), providing us with more “spam alerts” to capture more spam numbers more quickly. Second, by passively monitoring SMS communication with grey numbers, we avoid the user delay and errors introduced when submitting spam reports. Last, Greystar detects spammers based on their interactions with grey phone space. This prevents malicious users from gaming the Greystar detection system and launching DoS attacks against other legitimate users.

In the following, we first discuss related work in Section 8. We then study the difference of spamming and legitimate SMS activities in Section 4, which lays the foundation of the Greystar system. In Section 5 we introduce our methodology for identifying grey numbers. We then present the design of Greystar in Section 6 and evaluate it in Section 7.

4 Analyzing SMS Activities of Spammers and Legitimate Users

We first formally define SMS spamming activities. During a spamming process, a spammer selects (following a certain strategy) a sequence of *target phone numbers*, $X := \{x_1, x_2, \dots, x_i, \dots\}$ ($1 \leq i \leq n$), to send SMS messages to over a time window T . Each target phone number is a concatenation of two components, the 3-digit

area code x_i^a , which is location specific, and the 7-digit subscriber number x_i^s . Note that we only examine US phone numbers (which have 10 digits excluding the leading country code “1”). Phone numbers of SMS senders from other countries which follow the same North American Numbering Plan (NANP) are removed before the study. All the statistics in this section are calculated based on a whole month data from January 2012. To compare the activities of spam numbers and legitimate numbers, we obtain an equal amount of samples from both groups. In particular, the spam numbers are identified from victim spam reports and the legitimate numbers are randomly sampled from the remaining SMS senders appearing in the month-long CDR data set. Both samples of phone numbers are checked by fraud agents before the analysis to remove false positives and false negatives.

4.1 SMS Sending Rates

We first compare the SMS sending rates of known spam numbers and legitimate numbers. We measure the sending rate at the granularity of hours, i.e., the average number of unique recipients a phone number communicates with hourly. The CCDF curves of the sending rates are shown in Fig. 3.

From Fig. 3, spam numbers have a much higher SMS sending rate than legitimate numbers. This is not surprising given the purpose of spamming is to reach as many victims as possible within a short time period. In particular, more than 95% of spam numbers have a sending rate above 5 and more than 70% spam numbers exhibit a sending rate above 50. In contrast, more than 97% of the legitimate numbers have a sending rate below 5. As we can see in Section 6, by enforcing a threshold on the sending rate, we can filter out most of the legitimate numbers without missing many spam numbers.

Due to their high spamming rates, at the node-Bs that spam numbers are connected to, we find that the sheer volume of spamming traffic is astonishing. Spammer traffic can exceed normal SMS traffic by more than 10 times. Even at RNCs, which serve multiple node-Bs, traffic from spamming can account for 80% to 90% of total SMS traffic at times. Such a high traffic volume from spammers can exert excessive loads on the network, affecting legitimate SMS traffic. Furthermore, since SMS messages are carried over the voice control channel, excessive SMS traffic can deplete the network resource, and thus can potentially cause dropped calls and other network performance degradation. Meanwhile, the increasing malware app instances that propagate through the SMS channel also emphasize the importance of restricting SMS spam activities in cellular networks.

We note that, although most legitimate numbers send SMS at low rates (e.g., below 50), due to the large pop-

ulation size of the legitimate numbers, there are still many of them with high sending rates indistinguishable from those of spam numbers. Investigation shows that they belong to organizations which use the SMS service to disseminate information to their stakeholders, e.g., churches, schools, restaurants, etc. How to distinguish these legitimate intensive SMS senders from SMS spammers is the main focus of our Greystar system.

4.2 Spammer Target Selection Strategy

We next study how spammers select spamming targets. We characterize their target selection strategies at two levels, i.e., how spammers choose area codes and how they select phone numbers within each area code.

We define the metric *area code relative uncertainty* (ru_a) to measure whether a spammer favors phone numbers within certain area codes. The ru_a is defined as:

$$ru_a(X) := \frac{H(X^a)}{H_{\max}(X^a)} = \frac{-\sum_{q \in Q} P(q) \log P(q)}{\log |Q|},$$

where $P(q)$ represents the proportion of target phone numbers with the same area code q and $|Q|$ is the total number of area codes in the US. Intuitively, a large ru_a (e.g., greater than 0.7) indicates that the spammer uniformly chooses targets across all the area codes. In contrast, a small ru_a means the targets of the spammer come from only a few area codes.

We next define a metric *random spamming ratio* to measure how spammers select targets within each area code. Let P^a be the proportion of active phone numbers⁵ within area code a . For a particular spamming target sequence X^a of a spam number, if the spammer randomly chooses targets, the proportion of active phone numbers in X^a should be close to P^a . Otherwise, we believe the spammer has some prior knowledge (e.g., with an obtained target list) to select specific phone numbers to spam. Based on this idea, we carry out a one sided Binomial hypothesis test for each spammer and each area code to see if the corresponding target selection strategy is random within that area code. The random spamming ratio is then defined as the proportion of area codes within which a spammer selects targets randomly (i.e., the test fails to reject the randomness hypothesis with P-value=0.05). Note that, for each spam number, only area codes with more than 100 victims are tested to ensure the validity of the test.

⁵The active phone numbers are identified as all registered phone numbers inside the carrier’s billing database who have unexpired service plans. We find that the active numbers are uniform across all area codes, possibly due to frequent phone number recycling within carrier networks (e.g., phone numbers originally used by landlines are reassigned to mobile phones) and users switching between cellular carriers while retaining the same phone numbers.

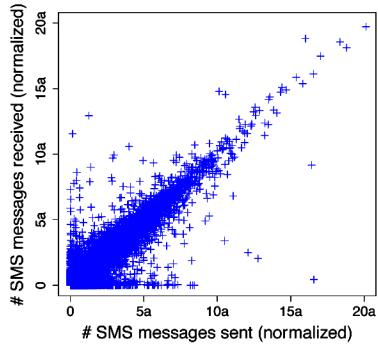


Figure 5: SMS sent vs. received.

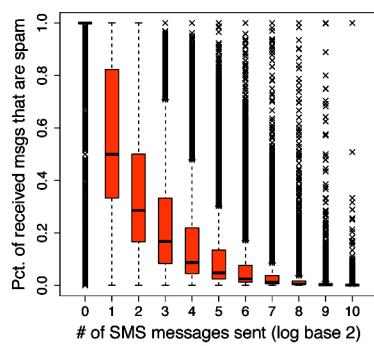


Figure 6: Activeness vs. spam prop.

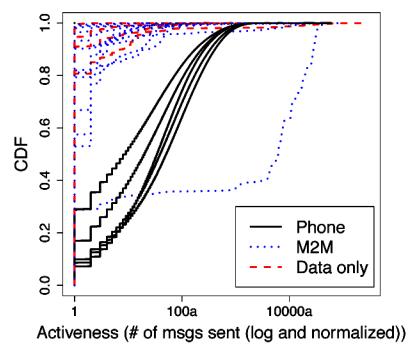


Figure 7: Device activeness (log).

Fig. 4 plots the ru_a (the x -axis) and the random spamming ratio (the y -axis) for individual spam numbers. For ease of visualization, we illustrate the marginal densities along both axes using boxplots. Based on the marginal density of ru_a , we find that many spam numbers (more than 60%, with $ru_a < 0.7$) concentrate on phone numbers within a few area codes. We find that spammers tend to focus on area codes with more users, i.e., those corresponding to large cities and metro areas, e.g., New York City, Chicago, Los Angeles, etc. In comparison, the remaining 40% of spam numbers select targets across many area codes or even the entire phone number space.

Meanwhile, based on the y -axis of Fig. 4, we find that, no matter how spam numbers choose area codes, a predominant portion of them select targets randomly within each area code. We refer to these spammers as *random spammers* hereafter. This also explains why spammers favor large metro areas, because they are likely to reach more active mobile users by randomly selecting phone numbers within these area codes.

In summary, due to the finite phone space, spammers can simply enumerate phone numbers to send spam messages. Compared to having a target phone number list before spamming, this random target selection strategy is effective and of low cost, and hence has been adopted by most SMS spammers. Due to their predominance, in this paper, we focus on detecting these random spammers. Meanwhile, the spammers who utilize non-random target selection strategies (e.g., the points at the bottom of Fig. 4) will be discussed in Section 7.3.

4.3 Mobile User SMS Activities

Since many SMS spammers adopt random target selection strategies, mobile users (within the same area code) have the same exposure to spam. In other words, given a fixed (long enough) observation period, these mobile users are expected to receive an equal amount of spam messages. In this section, we study the SMS activities

of legitimate mobile users and demonstrate that certain users can be used for detecting spam activities.

We first obtain a general understanding on the volume of SMS activities from legitimate mobile users in the network. Fig. 5 shows the number of messages sent (x -axis) and received (y -axis) by each user over a month⁶. We observe that a majority of users send and receive a similar amount of SMS messages and thereby form an approximate diagonal line. However, there are mobile users who deviate from such a pattern noticeably. For example, the points close to the x -axis represent users who send far more SMS messages than the ones they receive. These users consist of senders who own a large subscriber base, e.g., cellular providers, university emergency contact lines, political campaign lines, etc. In contrast, we observe quite a few points that reside near the y -axis. Investigation shows that they are phone numbers which receive periodic updates (e.g., electricity readings) from machine-to-machine (M2M) devices through SMS messages (see Section 5.2 for discussion of M2M devices).

Fig. 5 implies the different magnitude that mobile users engaged in SMS communication. To quantify the intensity of SMS activities from mobile users, we define (SMS) *activeness* as the number of messages sent from a mobile user during the observation period. Intuitively, for users who are less active, the spam messages tend to account for a more dominant proportion of their overall SMS communication. We illustrate this point in Fig. 6, where we bin all users based on their activeness (x -axis, in log scale), and calculate the distribution of the proportion of spam messages out of all SMS messages received by each user within each bin. Note that spam messages are identified as the SMS messages originated from spam numbers contained in victim spam reports. From Fig. 6, we observe an upward shift of spam message proportions as the activeness decreases. Interestingly, we find quite a

⁶We note that the constants used for normalization (denoted as a and b) vary across individual figures.

few numbers which have sent no more than 1 SMS message during the one month period. For a majority of these numbers, all the messages they have received are spam (as indicated by the fact that most probability mass is squeezed to a small region close to 1). This implies that these SMS inactive numbers are good indicators of spamming activities, i.e., SMS senders who communicate with them are more likely to be spammers.

5 SMS Grey Phone Number Space

In order to utilize these SMS inactive numbers for spam detection, we want to first answer the following questions. Why do these numbers have a low volume of SMS activity? Is there an inexpensive way to identify a stable set of such numbers for building the detection system? To answer these questions, we carry out an in-depth analysis of SMS inactive users. We then define grey phone space and propose a method for identifying the grey phone space using CDR records. In the end, we study properties of grey phone space and show the potential of using it to detect spamming activities.

5.1 Investigating Service Plans

Cellular carriers often provide their customers with a rich set of features to build their personal service plans. Users are free to choose the best combination of features to balance their needs and the cost. For example, a frequent voice caller often opts in an unlimited voice plan and a user who watches online videos a lot can choose a data plan with a larger data cap. Therefore, service plans encode demographic properties of the associated users. We hence study the correlations between different service plan features and SMS activeness to understand these SMS inactive users.

More specifically, we extract all the service plans associated with the legitimate user samples, which include features related to voice, data and SMS services. We calculate the Pearson correlation coefficients of the SMS activeness and individual plan features (treated as binary variables). The features are then ranked according to the correlation values. We summarize the top 5 features that are positively and negatively correlated with SMS activeness in Table 1.

Top 5 negatively correlated	Top 5 positively correlated
Text restricted	Monthly unlimited voice/text
Voice restricted	Messaging unlimited
Text msg pay per use	Rollover family plan
Voice/data prepaid	Unlimited SMS/MMS
Large cap data plans	Small cap data plans

Table 1: Corr. of activeness and plan features.

The top 5 features with negative correlations are in the first column of Table 1. Many of these SMS inactive users are enrolled in the pay-per-use SMS plan, a common economical choice for users who rarely access SMS services. Interestingly, a large number of SMS inactive users have restrictions on their voice/text plans and have been simultaneously enrolled in large cap data plans. Such restrictions only apply for mobile users with data only devices, such as tablets and laptop data cards, etc. In contrast, the top 5 features with positive correlations are summarized in the second column. Most of SMS active users have unlimited SMS plans, a favorable choice of frequent SMS communicators. Many of them have also enrolled in small cap data plans and unlimited MMS plans, which are dedicated for smartphone users.

Though service plans demonstrate clear distinctions between SMS inactive and active users, relying on service plans to identify SMS inactive users is not effective in practice due to two reasons. First, service plans change frequently, especially when users upgrade their devices. Second, query service plan information persistently during run time can be very expensive. Fortunately, our analysis above also reveals that service plans are strongly correlated with the device types, e.g., data only device users are less active compared to smartphone users. Can we use device types as a proxy to identify SMS inactive users instead? We shall explore such possibilities in the following section.

SMS towards data only devices. Like phones, laptops and other data only devices are also equipped with SIM cards and hence, once connected to the network, are able to receive SMS messages. We therefore can capture CDR records to these devices at MSCs. However, manufacturers often restrict text usage on these devices by masking the APIs related to SMS functions. Meanwhile, at the billing stage, text messages to these data only devices (with a text restricted plan) are not charged by the carrier. There are exceptions such as laptops enrolled in regular text messaging plans, however, such cases are rare based on our observations.

5.2 Identifying Grey Phone Space

The device associated with each phone number can be found in the CDR data based on the first eight-digit TAC of the IMEI. We use the most updated TAC to device mapping from the UMTS carrier in January 2013 and have identified 27 mobile device types (defined by the carrier) which we summarize in Table 2. We note that finer grained analysis at individual device level is also feasible. However, we find that, except for the vehicle tracking devices which we shall see soon, devices within each category have strong similarity in their SMS activeness distributions. Hence we gain little by defining grey

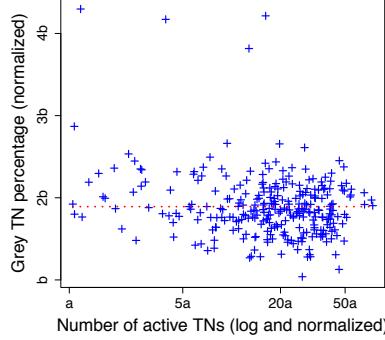


Figure 8: Grey number distribution.

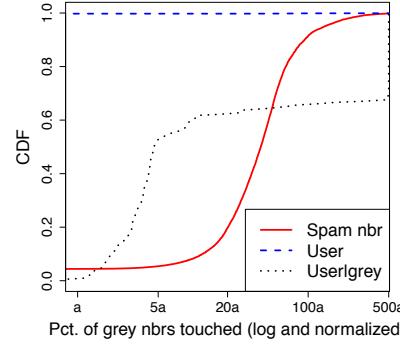


Figure 9: Grey ratio.

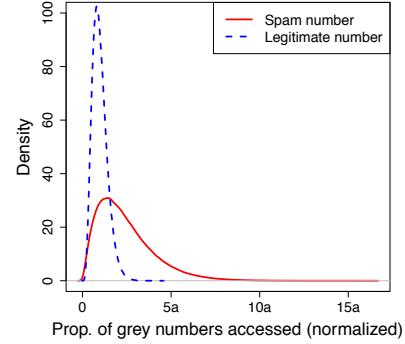


Figure 10: Distr. of θ and θ^* .

numbers at the device level.

Type	Examples
Data-only	Laptop data cards, tablets, netbooks, eReaders, 3G data modems, etc.
M2M	Security alarms, telematics, vehicle tracking devices, point-of-sale terminals, medical devices, etc.
Phone	Smartphones, feature phones, quick messaging phones, PDAs, etc.

Table 2: Device categories and examples.

Fig. 7 shows the CDF distributions of SMS activeness of phone numbers associated with different device types. We observe three clusters of CDF curves. The first one consists of curves concentrating at the top-left corner, representing devices with very low SMS activeness. This cluster covers all data only devices and a majority of machine-to-machine devices (see [11] for more discussions of M2M devices). The second cluster lies in the middle of the plot, which includes all phone devices. The third cluster contains only one M2M device type, which covers all vehicle tracking devices. Interestingly, the curve of such devices shows a bi-modal shape, where some devices communicate frequently using SMS while other devices mainly stay inactive. Based on Fig. 7, we define grey numbers as the ones that are associated with devices in the first cluster, i.e., data only devices and M2M devices excluding the vehicle tracking device category. The collection of all grey number are referred to as the grey phone space. The grey numbers are representatives of a subset of SMS inactive users⁷. Meanwhile, the grey phone space defined in this way is stable because it

⁷We use devices in the first cluster as our definitions of grey space, however, as we have seen in Fig. 7, even within the grey number categories there are still (a very few) numbers that are highly active in SMS communication. The proposed beta-binomial classification model (discussed in detail in Section 6) will take into account this fact. Intuitively, the model detects a spam number only when it is observed to have significant interaction with the grey space. Given a majority of the grey numbers that are SMS inactive, the chance that a phone number is mis-

is tied to mobile devices instead of specific phone numbers, whose behaviors can change over time (e.g., when a user upgrades the device). Furthermore, grey numbers can be identified directly based on the IMEIs in the CDR data with little cost, as opposed to querying and maintaining service plan information for individual users.

5.3 Characterizing Grey Phone Space

We next study the distribution of grey numbers and show how grey phone space can help us detect spamming activities.

Fig. 8 shows the size of each area code in the phone space (the x -axis, in terms of the number of active phone numbers) and the proportion of grey phone numbers out of all active phone numbers in that area code (the y -axis). The correlation coefficient of two dimensions is close to 0, indicating that grey numbers exist in both densely and sparsely populated areas. The wide distribution of grey numbers ensures a better chance of detecting spam numbers equipped with random spamming strategies. To illustrate this point, we calculate the proportion of grey numbers out of all the numbers accessed by spam numbers (red solid curve) and legitimate users (blue dotted curve). We observe that a predominant portion of legitimate users never touch grey phone space. In fact, less than 1% of the users have ever accessed grey numbers in the 1 month observation period. In addition, we show the same distribution for legitimate users (who have sent to at least 50 recipients in a month) conditioned on having touched at least one grey number. Compared to the spam numbers which tend to access more grey numbers (red solid curve), these legitimate users communicate with much fewer grey numbers. In most cases, the access of grey numbers is triggered by users replying to spam numbers who usually use M2M devices to launch spam.

classified as a spam number due to its interaction with these outliers in the grey space is very small.

5.4 Discussion: Greyspace vs. Darkspace

In addition to the grey phone space, the “dark” phone space (i.e., formed by unassigned phone numbers) can also be a choice for detecting spam activities using the same technique proposed in this paper. Analogous concepts of grey IP addresses and dark IP addresses for detecting anomalous activities have been explored in [12,13]. However, unlike IP addresses which are often assigned to organizations in blocks (i.e., sharing the same IP prefix), the phone number space is shared by different cellular service providers, landline service providers and even (IP) TV providers. Even if some phone numbers are assigned in blocks initially to a certain provider, the frequent phone number assignment changes caused by new user subscription, old user termination, recycling of phone numbers and phone number porting in/out between different providers will ultimately result in the shared ownership of the phone number space as we have seen today. For example, different cellular and landline providers can have phone numbers under the same legitimate area code. It is difficult to tell which phone number belongs to which provider without inquiring the right provider.

This poses significant challenges when we want to identify dark (unassigned) phone numbers. As dark phone numbers can be anywhere in the phone number space (within legitimate area codes) and can belong to any provider, it is rather difficult to determine a dark number, at least from the perspective of a single provider. For instance, just because a phone number is not assigned to any user/device belonging to a particular provider, it does not necessarily mean that such a number is dark. In other words, accurate detection of dark numbers requires the collaboration of all the owners of the phone number space, which is an intractable task. Meanwhile, such dark number repository needs to be updated frequently to reflect the changes of phone number assignments.

In comparison, grey numbers can be defined easily with respect to a particular provider: these are phone numbers assigned to devices belonging to customers of that provider where there are usually less SMS activities originated from these numbers (devices). Meanwhile, whether a number is grey is readily available to us (based on the existing the IMEI numbers inside CDR records) without any extra work.

6 System Design

In this section, we first present an overview of Greystar. We then introduce the detection model and how we choose parameters for the model.

6.1 System Overview

The logic of Greystar is illustrated in Alg. 1, which runs periodically at a predefined frequency. In our experiment, we run Greystar hourly. Greystar employs a time window of W (e.g., W equals 24 hours in our studies). The footprint of each SMS originating number s , e.g., the sets of grey and non-grey numbers accessed by s (denoted as G_s and N_s , respectively), are identified from the CDR data within W . After that, a filtering process is conducted which asserts two requirements on originating numbers to be classified, i.e., in the past 24 hours: i) the sender is active enough (which has sent messages to no less than $M = 50$ recipients. Recall the high sending rates of known spam numbers in Fig. 3); and ii) the sender has touched at least one grey number. These two criteria, especially the second one, can help significantly reduce the candidates to be classified in the follow-up step. In fact, we find that, on average, less than 0.1% of users send SMS to grey numbers in each day. More importantly, these users cover a majority of active SMS spammers in the network as we shall see in Section 7. As a consequence, this filtering step can noticeably reduce the system load as well as potential false alarms.

Algorithm 1 Greystar algorithm.

```
1: Input: CDR records  $D$  from the past  $W = 24$  hours,  $M=50$ ;  
2: Output: Spam number candidates  $C$ ;  
3: From  $D$ , extract all SMS senders  $Orig$ ;  
4: for each  $s \in Orig$  do  
5:   Extract the CDR records associated with  $s$ :  $D_s \subset D$ ;  
6:   From  $D_s$ , identify the grey numbers  $G_s$  and non-grey  
    numbers  $N_s$  accessed by  $s$ ;  
7:   if  $|G_s| + |N_s| \geq M$  and  $|G_s| > 0$  then  
8:     if  $detect\_spamnbr(G_s, N_s)=1$  then  
9:        $C := C \cup \{s\}$ ;  
10:    end if  
11:  end if  
12: end for
```

Once a sender passes the filtering process, the function $detect_spamnbr$ is called to classify the sender into either a spam number or a legitimate number based on G_s and N_s associated with that sender. In this paper, we propose a novel Beta-Binomial model for building the classifier, which we explain in detail next.

6.2 Classifier Design

We assume a random SMS spammer selects spamming targets following a two-step process. First, the spammer chooses a specific target phone number block. Second, the spammer uniformly chooses target phone numbers from that block. Let θ denote the density of grey numbers in the target block and $X := \{x_i\}, 1 \leq i \leq n$ be the

sequence of target phone numbers selected. Meanwhile, let k be the number of grey numbers in X . The target selection process can then be formulated as the following generative process.

1. Choose a target block with grey number density θ ;
2. Choose $x_i \sim Bernoulli(\theta)$, $1 \leq i \leq n$;

We note that θ varies as a spammer chooses different phone number blocks. The choice of phone number blocks is arbitrary. For example, A spammer can choose a large phone block across multiple area codes or a small one consisting of only a fraction of phone numbers within one area code. Therefore, θ itself can be considered as a random variable. We assume θ follows a Beta distribution⁸, i.e., $\theta \sim Beta(\alpha, \beta)$, with a probability density function as:

$$P(\theta|\alpha, \beta) = \frac{\Gamma(\alpha + \beta)}{\Gamma(\alpha)\Gamma(\beta)} \theta^\alpha (1 - \theta)^\beta,$$

where Γ is the gamma function. Therefore, the random variable k follows a Beta-Binomial distribution:

$$P(k|n, \alpha, \beta) = \binom{n}{k} \frac{\Gamma(k + \alpha)\Gamma(n - k + \beta)}{\Gamma(n + \alpha + \beta)} \frac{\Gamma(\alpha + \beta)}{\Gamma(\alpha)\Gamma(\beta)}$$

The target selection process of legitimate users can be expressed using the same process. Because legitimate users tend to communicate less with grey numbers, their corresponding θ^* 's are usually much smaller. Let α^* and β^* be the parametrization of the Beta distribution associated with θ^* . For a phone number that has accessed n targets, out of which k are grey numbers, we classify it as a spam number (i.e., `detect_spamnbr` returns 1) if

$$\frac{P(spammer|k, n)}{P(legitimate|k, n)} = \frac{P(k|n, \alpha, \beta)P(spammer)}{P(k|n, \alpha^*, \beta^*)P(legitimate)} > 1,$$

where the first equation is derived using the Bayes theorem. It is equivalent to

$$\frac{P(k|n, \alpha, \beta)}{P(k|n, \alpha^*, \beta^*)} > \frac{P(legitimate)}{P(spammer)} = \eta$$

In practice, it is usually unclear how many spammers are in the network, therefore, to estimate η directly is challenging. We instead choose η through experiments.

⁸In Bayesian inference, the Beta distribution is the conjugate prior probability distribution for the Bernoulli and binomial distributions. Instead of using the Bernoulli model, we can model the second stage of the target selection process as sampling from a multinomial distribution corresponding to different device types. In this case, the conjugate prior distribution of the multinomial parameters is the Dirichlet distribution. However, our preliminary experiments show little performance gain from applying the more sophisticated model in comparison to the increased computation cost.

6.3 Parameter Selection

There are five parameters to be estimated in the classifier, $\hat{\alpha}, \hat{\beta}, \hat{\alpha}^*, \hat{\beta}^*$ and η . We use the data from January 2012 to determine these parameters. To obtain ground truth, we submit to the fraud agents a list of all the SMS senders that i) have sent to more than 50 recipients in a 24 hour time window; and ii) at least one of the recipients is grey (recall the filtering criteria in Algorithm 1). Fraud agents carry out investigation on these numbers for us and label spam numbers in the list. We then divide the January data into two subsets, the first two weeks of data for fitting the Beta-binomial models (i.e., to determine the first four parameters) and the rest of data is reserved for testing the classifier to estimate η .

In particular, using the training data set, we estimate the parameters for two Beta-binomial models using maximum likelihood estimation. With the estimated parameters, we illustrate the probability density function $\theta \sim Beta(\alpha, \beta)$ and $\theta^* \sim Beta(\hat{\alpha}^*, \hat{\beta}^*)$ in Fig. 10. The density functions agree with our previous observations in Fig. 9. The mass of the probability function corresponding to the legitimate users concentrates on a narrow region close to 0, implying that legitimate users communicate much less with grey numbers than non-grey numbers. In contrast, the density associated with spam numbers widely spreads out, indicating more grey numbers are touched by spam numbers due to their random target selection strategies.

We evaluate the accuracy of the classifier given different choices of η on the test data set and the Receiver Operating Characteristic (ROC) curve is displayed in Fig. 11. The x -axis represents the false alarm rate (or the false positive rate) and the y -axis stands for the true detection rate (or the true positive rate). From Fig. 11, with a certain η , Greystar can detect more than 85% spam numbers without producing any false alarm. We will choose this η value in the rest of our experiments⁹.

7 Greystar Evaluation

In this section, we conduct an extensive evaluation of Greystar using five months of CDR data and compare it with the methods based on victim spam reports in terms of accuracy, detection delay and the effectiveness in reducing spam traffic in the network.

⁹Note that the exact parameter values used in Greystar are proprietary and we are not able to release them in the paper. We have also tested the choice of η using different partitioning of the training/test data. The η remains stable across experiments.

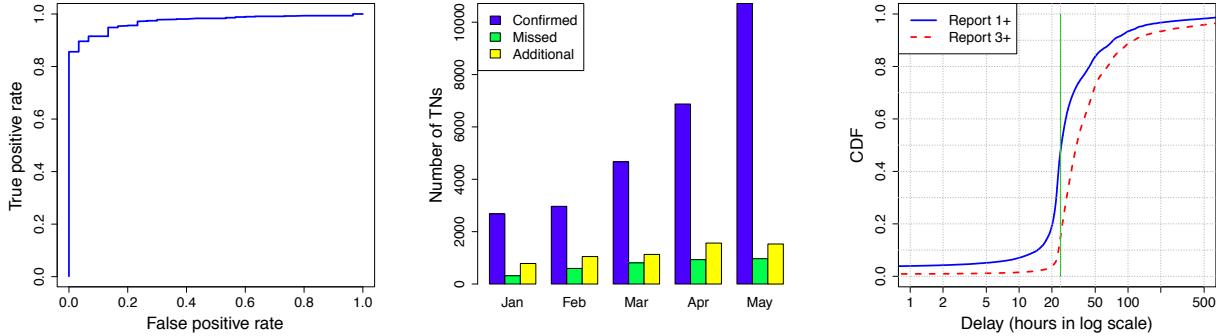


Figure 11: ROC curve (false positive rate vs. true positive rate).

Figure 12: Accuracy evaluation (in comparison to victim spam reports).

Figure 13: Detection speed compared to spam report based methods.

7.1 Accuracy Evaluation

To estimate the accuracy and the false alarm rate, we again consult with the fraud agents to check the numbers from Greystar detection results. False negatives (or missed detections), on the other hand, are more difficult to identify. Given the huge number of negative examples classified, we are unable to have all of them examined by the fraud agents to identify all missed detections because of the high manual investigation cost. As an alternative solution, we compare Greystar detection results with victim spam reports to obtain a lower bound estimate of the missed detections.

More formally, let S_g denote the detection results from Greystar and S_c be the spam numbers contained in the victim spam reports received during the same time period. We define *missed detections* of Greystar as $S_c - S_g$. In addition, we define *additional detections* of Greystar as $S_g - S_c$ to measure the value brought by Greystar to the existing spam defense solution. The monthly accuracy evaluation results are displayed in Fig. 12.

The blue bars in Fig. 12 illustrate the spam numbers validated by fraud agents in each month. Greystar is able to detect thousands of spam numbers per month. The ascending trend of detected spam numbers coincides with the increase of victim spam reports in the five-month observation window. This implies that Greystar is able to keep up with the increase of spam activities. In addition to the large number of true detections, Greystar is highly accurate given only two potential false alarms are identified by fraud agents in 5 months. Interestingly, these two numbers are associated with tenured smartphone users who suddenly behave abnormally and initiate SMS messages to many recipients whom they have never communicated with in the past. We suspect these users have been infected by SMS spamming malware that launch spam campaigns from the users' devices without their consent. To identify SMS spamming malware and hence

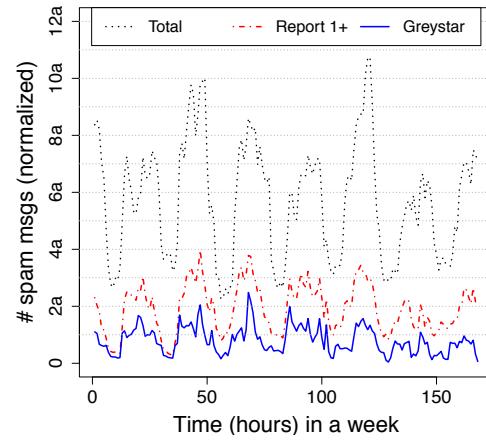


Figure 14: Number of spam messages after restriction.

removing such false alarms will be our future work.

In comparison to the victim spam reports, Greystar detects over 1000 additional spam numbers that were not reported by spam victims while missing less than 500 monthly. Meanwhile, although a majority of the spam numbers detected by Greystar are also reported by spam victims, Greystar can detect these numbers much faster than methods based on victim reports, and consequently can suppress more spam messages in the network. We illustrate this point in the next section.

7.2 Detection Speed and Benefits to Cellular Carriers

We note that, to reduce noise, cellular carriers often rely on multiple spam reports (e.g., K reports) from different victims to confirm a spam number. We refer to such a crowdsourcing method as the $K+$ algorithm. To evaluate the speed of Greystar, we compare it with two versions of the $K+$ algorithms, namely, 1+ and 3+. Comparing with

$1+$ supplies us with the lower bound of the time difference and comparison with $3+$ illustrates the real benefit brought by Greystar to practical spam defense solutions. More specifically, we measure how many hours Greystar detects a spam number ahead of $1+$ and $3+$, respectively. Fig. 13 shows the CDF curves of the comparison results, where we highlight the location on the x -axis corresponding to 24 hours with a green vertical line. We observe that Greystar is much faster than $K+$ algorithms. For example, Greystar is one day ahead of $1+$ in 50% of the cases and is one day before $3+$ in more than 90% of the times.

We find that, on average, it takes less than 1.2 hours for Greystar to detect a spam number after it starts spamming (i.e., starts sending messages to more than 50 victims in an hour). The fast response time of Greystar is accredited to the much larger population of grey numbers, from which Greystar can gather evidence to detect more spam numbers more quickly. In addition, collecting evidence passively from grey numbers eliminates the delay during the human reporting process (recall Fig. 2). Therefore, Greystar is characterized with a much faster detection speed than the $K+$ algorithm. Such a gain in the detection speed can lead to more successful reduction of spam traffic in the network. We illustrate this point next.

For simplicity, we assume a spam number can be instantly restricted after being detected. We run simulation on a one week dataset (the first week of January 2012) and calculate the number of spam messages appearing in each hour assuming a particular spam detection algorithm is deployed exclusively in the network. The results are illustrated in Fig. 14. The total spam messages are contributed by known spam numbers observed in that week. We observe that Greystar can successfully suppress the majority of spam messages. During peak hours when the total number of spam messages exceeds 600K, only around 150K remains after Greystar is deployed. In other words, Greystar leads to an overall reduction of 75% of spam messages during peak hours. In comparison, $1+$ only guarantees a spam reduction of 50% due to long detection delay. We note that, due to the noise in the spam reports, cellular providers often employ $K+$ ($K \geq 3$) instead of $1+$ to avoid false alarms. In this case, the benefit from Greystar is even more substantial.

7.3 Analysis of Missed Detectors

In this section, we investigate the missed detections (false negatives) from Greystar, i.e., the spam number candidates that were not detected by Greystar but have been reported by spam victims. There are around 500 such numbers in each month and totally around 27K missed detections. We note that we focus only on a subset of the candidates who are customers of the cellular

network under study, for whom we have access to a much richer set of information sources to carry out the investigation. We believe the conclusions from analyzing this subset of candidates also apply for other candidates outside the network.

We classify these candidates into three groups based on the volume of the associated CDR records.

No volume. We do not observe any CDR record for 19.5% of the numbers. We inquiry the SMS billing records for these numbers and find that many of them initiate a vast amount of SMS traffic to foreign countries, such as Canada and Jamaica, etc., and hence no CDR record has been collected to trigger Greystar detection.

Low volume. We find around 27% of the missed detections have accessed less than 50 recipients during the observation period. We study the text content inside the victim spam reports to understand the root cause of these missed detections. The most popular text content are party advertisements and promotions from local restaurants. Users are likely to have registered with these merchants in the past and hence received ads from them. For the rest of the numbers, we find many send out spam messages to advertise mobile apps and premium SMS services. From the users' comments posted on online forums and social media sites [14, 15], we find two of the advertised apps are messenger/dating apps which have issues with their default personal settings. Without manual correction, these apps, once initiated, will send out friend requests to a few random users of the apps. Spam messages from the remaining numbers are also likely to be sent out without users' consent, especially the ones that broadcast premium SMS services. We suspect they are caused by apps abusing permissions or even behaviors of malware apps. For example, one app advertised by spam is reported to contain malware that sent SMS text to the contact list on the infected device, where the text contains a URL for downloading that malware.

High volume. The rest of the phone numbers send SMS to a large number of recipients. From the reported spam text, we find 7.1% of them belong to legitimate advertiser who broadcast to registered customers and are somehow reported by the recipients. For the rest of numbers, we find their spam topics are quite different from those of the detected ones. In particular, 11% of these numbers are associated with adult sites or hotlines, in comparison to only 0.06% among the detected numbers. Meanwhile, 17.6% of them advertise local shopping deals, as opposed to only 2.1% among the detected ones. Such difference suggests that these spam victims somehow gave out their phone numbers to spammers, e.g., while visiting malicious sites to register services or to purchase products. In addition, we extract the voice call history associated with these high volume candidates. Interestingly, we

find that about 4% of these numbers have initiated phone calls to many terminating numbers in the past. We suspect that these spammers employ auto-dialers to harvest active phone numbers (i.e., the ones that have answered the calls) from the phone number space. With the list of active phone numbers, spammers can send spam more effectively and avoid detection in the mean time.

Admittedly, there are spam numbers in these three categories that are missed by Greystar because they are equipped with a target number list obtained through auto-dialing or social engineering techniques (for example, accurate target lists can potentially be obtained by applying techniques discussed in [16]). SMS traffic from these users is not differentiable from that of the legitimate users. However, we emphasize that these missed detections only account for less than 9% of all the spam numbers detected and they will not have a significant impact on the efficacy of Greystar for reducing the overall spam traffic. In fact, we find that, on average, the missed detections sent 37% less spam messages in comparison to the spam numbers detected by Greystar. On the other hand, we do see the needs of combining Greystar and other methods to build a more robust defense solution. For example, many malicious activities can be better detected by correlating different channels (e.g., voice, SMS and data). Meanwhile, cellular carriers can collaborate with mobile marketplace to detect and control suspicious apps that can potentially initiate spam.

8 Related Work

The demographic features and network behaviors of SMS spammers were analyzed in [6]. [16] investigated the security impact of SMS messages and discussed the potential of denying voice service by sending SMS to large and accurate phone hitlists at a high rate. Meanwhile, [16] also discussed several ways of harvesting active phone numbers, which can potentially be employed by SMS spammers to generate accurate target number lists to launch spam campaign more efficiently and to evade detection. Similar short message services carried by the data channel were also studied. For example, [17] characterized spam campaigns from “wall” messages between Facebook users. [18–21] analyzed Twitter spam. [22, 23] studied talkback spam on weblogs. Meanwhile, akin to SMS spammers, the behaviors of email spammers were characterized in [24–27]. In comparison, we not only study the strategies of SMS spammers but also propose an effective spam detection solution based on our analysis.

In addition to the victim spam reports mentioned earlier, network behaviors of spammers, e.g., sending patterns, have been used in SMS spam detection, such as [28]. Similar network statistics based methods de-

signed for email spam detection can also be applied for identifying SMS spam, such as [29–32]. However, these methods often suffer from large false positive rates, because many legitimate customers can exhibit SMS sending patterns similar to those of spammers. In contrast, Greystar utilizes a novel concept of grey phone space to detect spam numbers, which yields an extremely low false alarm rate.

Some systems have been developed in the form of smartphone apps to classify spam messages on user mobile devices [33–35]. However, not all mobile devices support executing such apps. Furthermore, from a user’s perspective, this method is a late defense as the spam message has already arrived on his/her device and the user may already be charged for the spam message. Moreover, the high volume of spam messages that have already traversed the cellular network may have resulted in congestion and other adverse network performance impacts. Greystar is deployed inside the carrier network and hence do not have these drawbacks. As we have seen in Section 7, Greystar can quickly detect spam numbers once they start spamming and hence significantly reduce spam traffic volume in the network.

Similar to our work, many works have leveraged unwanted traffic for anomaly detection, such as Internet dark space [13, 36], grey space [12], honeynet [37, 38] and failed DNS traffic [39], etc. We are the first to advance the notion of grey phone space and propose a novel statistical method for identifying SMS spam using grey phone space.

9 Conclusion and Future Work

In this paper, we presented the design of Greystar, an innovative system for fast and accurate detection of SMS spam numbers. Greystar monitors a set of grey phone numbers, which signify impending spam activities targeting a large number of mobile users, and employs an advanced statistical model for detecting spam numbers according to their interactions with grey phone numbers. Using five months of SMS call detail records collected from a large cellular network in the US, we conducted extensive evaluation of Greystar in terms of the detection accuracy and speed, and demonstrated the great potential of Greystar for reducing SMS spam traffic in the network.

Our future work will focus on applying Greystar to detect other suspicious activities in cellular networks, such as telemarketing campaigns. Meanwhile, we will correlate Greystar detection results with cellular data traffic to detect malware engaged in such spamming activities.

Acknowledgments

The work was supported in part by the NSF grants CNS-1017647 and CNS-1117536, the DTRA grant HDTRA1-09-1-0050. We thank Peter Coulter, Cheri Kerstetter and Colin Goodall for their useful discussions and constructive comments. Finally, we thank our shepherd, Patrick Traynor, for his many suggestions on improving the paper.

References

- [1] Federal communications commission. Spam: unwanted text messages and email, 2012. <http://www.fcc.gov/guides/spam-unwanted-text-messages-and-email>.
- [2] Mobile spam texts hit 4.5 billion. <http://www.businessweek.com/news/2012-04-30/mobile-spam-texts-hit-4-dot-5-billion-raising-consumer-ire>.
- [3] C. Baldwin. 350,000 different types of spam sms messages were targeted at mobile users in 2012, 2013. <http://www.computerweekly.com/news/2240178681/350000-different-types-of-spam-SMS-messages-were-targeted-at-mobile-users-in-2012>.
- [4] 69% of mobile phone users get text spam, 2012. <http://abnews.go.com/blogs/technology/2012/08/69-of-mobile-phone-users-get-text-spam/>.
- [5] Y. Jin, N. Duffield, A. Gerber, P. Haffner, W.-L. Hsu, G. Jacobson, S. Sen, S. Venkataraman, and Z.-L. Zhang. Making sense of customer tickets in cellular networks. In *Proc. of the 30th IEEE International Conference on Computer Communications*, 2011.
- [6] I. Murynets and R. Jover. Crime scene investigation: Sms spam data analysis. In *Proc. of the 12th ACM Internet Measurement Conference*, 2012.
- [7] S. Sinha, M. Bailey, and F. Jahanian. Improving SPAM blacklisting through dynamic thresholding and speculative aggregation. In *Proc. of the 17th Annual Network and Distributed System Security Symposium*, 2010.
- [8] J. Jung and E. Sit. An empirical study of spam traffic and the use of DNS black lists. In *Proc. of the 4th ACM Internet Measurement Conference*, 2004.
- [9] A. Ramachandran, N. Feamster, and D. Dagon. Revealing botnet membership using dnsbl counter-intelligence. In *Proc. of the 2nd Workshop on Steps to Reducing Unwanted Traffic on the Internet*, 2006.
- [10] Y. Xie, F. Yu, K. Achan, R. Panigrahy, G. Hulten, and I. Osipkov. Spammering botnets: signatures and characteristics. In *Proc. of the 2008 ACM SIGCOMM Annual Conference*, 2008.
- [11] M. Shafiq, L. Ji, A. Liu, J. Pang, and J. Wang. A first look at cellular machine-to-machine traffic: large scale measurement and characterization. In *Proc. of the 2012 ACM International Conference on Measurement and Modeling of Computer Systems*, 2012.
- [12] Y. Jin, G. Simon, K. Xu, Z.-L. Zhang, and V. Kumar. Gray's anatomy: dissecting scanning activities using ip gray space analysis. In *Proc. of the 2nd Workshop on Tackling Computer Systems Problems with Machine Learning Techniques*, 2007.
- [13] R. Pang, V. Yegneswaran, P. Barford, V. Paxson, and L. Peterson. Characteristics of internet background radiation. In *Proc. of the 4th ACM Internet measurement conference*, 2004.
- [14] Sms watchdog. <http://www.smswatchdog.com>.
- [15] 800notes - Directory of unknown callers. <http://www.800notes.com>.
- [16] W. Enck, P. Traynor, P. McDaniel, and T. La Porta. Exploiting open functionality in sms-capable cellular networks. In *Proc. of the 12th ACM Conference on Computer and Communications Security*, 2005.
- [17] H. Gao, J. Hu, C. Wilson, Z. Li, Y. Chen, and B. Zhao. Detecting and characterizing social spam campaigns. In *Proc. of the 10th ACM Internet Measurement Conference*, 2010.
- [18] S. Ghosh, B. Viswanath, F. Kooti, N. Sharma, G. Korlam, F. Benevenuto, N. Ganguly, and K. Gummadi. Understanding and combating link farming in the twitter social network. In *Proc. of the 21st International World Wide Web Conference*, 2012.
- [19] K. Thomas, C. Grier, V. Paxson, and D. Song. Suspended accounts in retrospect: an analysis of Twitter spam. In *Proc. of the 11th ACM Internet Measurement Conference*, 2011.
- [20] 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 *Proc. of the 21st International World Wide Web Conference*, 2012.

- [21] C. Grier, K. Thomas, V. Paxson, and M. Zhang. @spam: the underground on 140 characters or less. In *Proc. of the 17th ACM Conference on Computer and Communications Security*, 2010.
- [22] E. Bursztein, P. Lam, and J. Mitchell. Track-back spam abuse and prevention. In *Proc. of the 2009 ACM workshop on Cloud computing security*, 2009.
- [23] E. Bursztein, B. Gourdin, and J. Mitchell. Reclaiming the blogosphere talkback a secure linkback protocol for weblogs. In *Proc. of the 16th European Symposium on Research in Computer Security*, 2011.
- [24] C. Kreibich, C. Kanich, K. Levchenko, B. Enright, G. Voelker, V. Paxson, and S. Savage. On the spam campaign trail. In *Proc. of the 1st USENIX Workshop on Large-Scale Exploits and Emergent Threats*, 2008.
- [25] C. Kreibich, C. Kanich, K. Levchenko, B. Enright, G. Voelker, V. Paxson, and S. Savage. Spamsraft: An inside look at spam campaign orchestration. In *Proc. of the 2nd USENIX Workshop on Large-Scale Exploits and Emergent Threats*, 2009.
- [26] C. Kanich, C. Kreibich, K. Levchenko, B. Enright, G. Voelker, V. Paxson, and S. Savage. Spamalytics: An empirical analysis of spam marketing conversion. *Communications of the ACM*, 52(9):99–107, 2009.
- [27] A. Pathak, F. Qian, C. Hu, M. Mao, and S. Ranjan. Botnet spam campaigns can be long lasting: evidence, implications, and analysis. In *Proc. of the 2009 ACM International Conference on Measurement and Modeling of Computer Systems*, 2009.
- [28] Q. Xu, E. Xiang, Q. Yang, J. Du, and J. Zhong. Sms spam detection using noncontent features. *Intelligent Systems, IEEE*, 27(6):44–51, 2012.
- [29] T. Ouyang, S. Ray, M. Rabinovich, and M. Allman. Can network characteristics detect spam effectively in a stand-alone enterprise? In *Proc. of the 12th Passive and Active Measurement conference*, 2011.
- [30] M. Sirivianos, K. Kim, and X. Yang. Introducing social trust to collaborative spam mitigation. In *Proc. of the 30th IEEE International Conference on Computer Communications*, 2011.
- [31] S. Hao, N. Syed, N. Feamster, A. Gray, and S. Krasser. Detecting spammers with snare: spatio-temporal network-level automatic reputation engine. In *Proc. of the 18th USENIX Security Symposium*, 2009.
- [32] A. Pitsillidis, K. Levchenko, C. Kreibich, C. Kanich, G.M. Voelker, V. Paxson, N. Weaver, and S. Savage. Botnet judo: Fighting spam with itself. In *Proc. of the 17th Annual Network and Distributed System Security Symposium*, 2010.
- [33] K. Yadav, P. Kumaraguru, A. Goyal, A. Gupta, and V. Naik. Smsassassin: crowdsourcing driven mobile-based system for sms spam filtering. In *Proc. of the 12th Workshop on Mobile Computing Systems and Applications*, 2011.
- [34] G. Cormack, J. Hidalgo, and E. Sánchez. Feature engineering for mobile (sms) spam filtering. In *Proc. of the 30th international ACM SIGIR conference*, 2007.
- [35] H. Tan, N. Goharian, and M. Sherr. \$100,000 Prize Jackpot. Call now! Identifying the pertinent features of SMS spam. In *Proc. of the 35th Annual ACM SIGIR Conference*, 2012.
- [36] E. Wustrow, M. Karir, M. Bailey, F. Jahanian, and G. Huston. Internet background radiation revisited. In *Proc. of the 10th ACM Internet measurement conference*, 2010.
- [37] The honeynet project, 2012. <http://project.honeynet.org/>.
- [38] G. Dunlap, S. King, S. Cinar, M. Basrai, and P. Chen. Revirt: enabling intrusion analysis through virtual-machine logging and replay. In *Proc. of the 2nd USENIX Symposium on Operating Systems Design and Implementation*, 2002.
- [39] N. Jiang, J. Cao, Y. Jin, L. Li, and Z.-L. Zhang. Identifying suspicious activities through dns failure graph analysis. In *Proc. of the 8th IEEE International Conference on Network Protocols*, 2010.

Practical Comprehensive Bounds on Surreptitious Communication Over DNS

Vern Paxson^{◇*} Mihai Christodorescu[†] Mobin Javed[◇] Josyula Rao[‡] Reiner Sailer[‡]
Douglas Schales[‡] Marc Ph. Stoecklin[‡] Kurt Thomas[◇] Wietse Venema[‡] Nicholas Weaver^{*§}
[◇]UC Berkeley ^{*}ICSI [†]Qualcomm Research [‡]IBM Research [§]UC San Diego

Abstract

DNS queries represent one of the most common forms of network traffic, and likely the least blocked by sites. As such, DNS provides a highly attractive channel for attackers who wish to communicate surreptitiously across a network perimeter, and indeed a variety of tunneling toolkits exist [7, 10, 13–15]. We develop a novel measurement procedure that fundamentally limits the amount of information that a domain can receive surreptitiously through DNS queries to an upper bound specified by a site’s security policy, with the exact setting representing a tradeoff between the scope of potential leakage versus the quantity of possible detections that a site’s analysts must investigate.

Rooted in lossless compression, our measurement procedure is free from false negatives. For example, we address conventional tunnels that embed the payload in the query names, tunnels that repeatedly query a fixed alphabet of domain names or varying query types, tunnels that embed information in query timing, and communication that employs combinations of these. In an analysis of 230 billion lookups from real production networks, our procedure detected 59 confirmed tunnels. For the enterprise datasets with lookups by individual clients, detecting surreptitious communication that exceeds 4 kB/day imposes an average analyst burden of 1–2 investigations/week.

1 Introduction

Some of the most serious security threats that enterprises face concern the potential use of surreptitious communication (Figure 1). One such scenario takes the form of *exfiltration*, when an attacker with internal access aims to transmit documents or other substantive data out of the enterprise to a remote location [4]. Another scenario arises in the context of *interactive remote access*: an attacker who has patiently compromised a local system subsequently interacts with it over the network in order to assay the information it holds and employ it as an internal stepping stone for further probing of the enterprise.

DNS plays a pervasive role in Internet communication; indeed, the vast majority of *any* Internet communication ultimately begins with DNS queries. Even sites that are highly security-conscious will find that they still

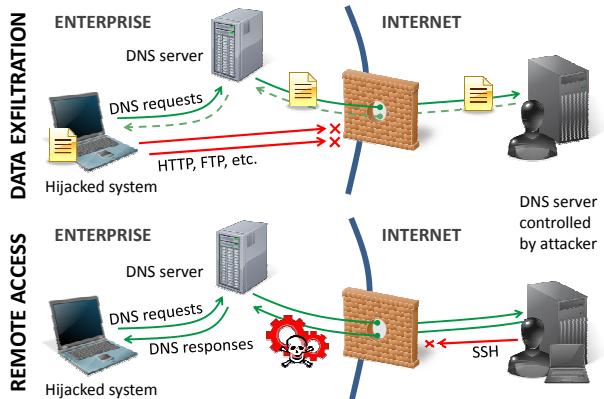


Figure 1: Two examples of surreptitious communication via DNS tunnels through perimeter firewalls.

must allow internal clients to issue DNS queries and receive the replies. Unless sites can restrict their systems to only intra-enterprise communication, some of these queries will necessarily reach external systems, giving attackers the opportunity to piggyback their actual communication over seemingly benign DNS traffic. Thus, DNS provides a highly-attractive target for attackers seeking a means of surreptitious communication.

We note that such communication fundamentally cannot be detected at the level of individual DNS queries. For example, an attacker could exfiltrate only one bit of information per day by having a local system under their control each day issue a single query for either `www.attacker.com` or `mail.attacker.com`, where the label used (`www` or `mail`) conveys either a 0 bit or a 1 bit.¹ It will prove intractable for a site’s security analysts (or any detection tool) to tell that such requests reflect adversarial activity, absent a great deal of additional information.

In this work we develop a principled means—rooted in assessments of information-theoretic entropy and free from false negatives—by which sites can analyze their

¹ We assume that the attacker controls the `attacker.com` DNS zone.

DNS activity and detect the presence of surreptitious communication whose volume exceeds a configurable bound. Simpler metrics, such as volume of DNS traffic, are not useful to distinguish tunnels from normal query traffic, because large-scale traffic naturally exhibits a high degree of diversity (§ 5.2). Approaches that focus on specific syntactical patterns [29] will miss communication with different encodings. Our configurable bound on the volume of surreptitious communication over DNS allows sites to trade off analysis burden (detections requiring investigation) versus assurance that such communication does not exceed a considerably low level.

We formulate this detection problem as having three main components. The first concerns constructing a sound, fairly tight estimate of the amount of information potentially encoded in a stream of DNS queries. Here we need to comprehensively identify all potential *information vectors*, i.e., aspects of DNS queries that can encode information. The second regards ensuring that we can compute such estimates with reasonable efficiency in the face of very high volumes of DNS activity (tens of millions of lookups per day). Finally, we need to assess to what degree benign DNS query streams encode significant amounts of information, and formulate effective ways of minimizing the burden that such benign activity imposes on a site’s security assessment process.

Thus, we conceptualize our overall goal as providing a site’s security analysts with a high-quality, tractable set of domains for which the corresponding DNS lookups potentially reflect surreptitious communication. We view it as acceptable that the analyst then needs to conduct a manual assessment to determine which of the candidates actually reflects a problem, *provided* that we keep the set small and the process of eliminating a benign candidate does not require much attention.

This work makes the following contributions:

- We introduce a principled means of detecting the presence of surreptitious communication over DNS, parameterized by a (configurable) bound on the amount of information conveyed.
- Our approach is *comprehensive* because we root our estimates of information conveyed in DNS lookups in lossless compression of entire query streams.
- We perform an in-depth empirical analysis of mostly-benign DNS traffic on an extensive set of traces comprising 230 billion queries observed across a variety of sites. For enterprise datasets with lookups by individual clients, we find that a bound of 4 kB/day per client and registered domain name imposes an operationally viable analysis burden. Thus, we argue that our procedure proves practical for real-world operational use.

After a summary of our information measurement procedure, we define the threat model in § 3. In § 4 we

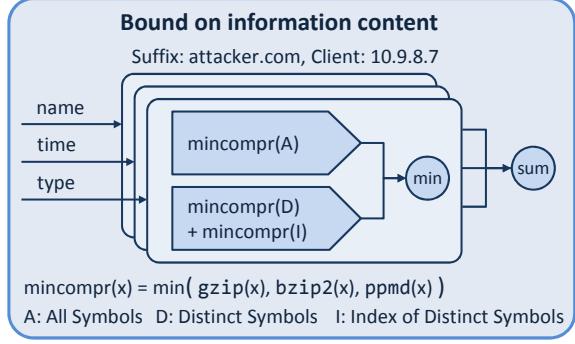


Figure 2: The information measurement procedure, summarized in § 2. Figure 5 shows the full detection procedure.

present the extensive datasets used in our study. We discuss information vectors potentially present in DNS queries and ways to estimate their volume in § 5, and explore implementation issues, including filtering techniques for reducing the resources required, in § 6. We evaluate the efficacy of our procedure in § 7, present a real-time detector in § 8, discuss findings, limitations, and future work in § 9, and review prior work in § 10.

2 Summary of the information measurement procedure

As explained in § 6.3, we analyze DNS queries per client and per registered domain name. For example, we aggregate queries with names ending in site₁.com, site₂.co.uk, and so on. We also aggregate PTR queries, but ignore them here for clarity.

We measure the information in query name, time and record-type sequences separately (§ 5.3). For example, we transform a sequence of query names *A* to a sequence of indices *I* into a table with distinct names *D*, and then compress *I* and *D* with *gzip*. The size of the output then gives us a measurement of the information in the input sequence.

The key insight is that *we will never under-estimate the information in a query sequence as long as the transformation and compression are reversible*, i.e., we can recover the original input sequence. Taking advantage of this insight, we subject each query attribute sequence to multiple (transformation, compressor) alternatives and use the minimal result as the tightest (upper) bound.

Figure 2 illustrates our measurement procedure. For each client and registered domain we compress both the original and transformed query name sequences with *gzip*, *bzip2* and *ppmd* [23], and take the size of the smallest output. We apply the same procedure to the record-type sequences and 32-bit inter-query arrival time distances, and from these compute a combined score.

Site	Features	Vantage point	Notes	Time span	Daily statistics: Average (Daily peak)		
					Clients	Total lookups	Distinct lookups
INDLAB	L,N,Q,T	I		1,212 d	10k (16k)	47M (164M)	310k (2.4M)
LBL	N,Q	I		2,776 d	6.8k (11k)	28M (154M)	867k (2.2M)
NERSC	N,Q	I		1,642 d	1.3k (3.3k)	9M (59M)	44k (114k)
UCB	N,Q,T	E	^a	45 d	2.1k (5.1k)	38M (52M)	3.3M (4.4M)
CHINA	N,Q,T	I/E	^b	5 d	61k (101k)	13.9M (15.7M)	468k (670k)
SIE	N	A	^{c,d}	53 d	123* (123*)	1.45B (1.84B)	110M (129M)

Table 1: Summary of data sources. The available features are: 0x20-encoding [27] (C), caching lifetime derived from reply time-to-live (L), query name (N), query type (Q), and timing (T). Sensor vantage points are: aggregated across multiple sites (A), external to site (E), lookups associated with individual clients as seen at internal name servers (I), a mixture of these last two (I/E).

^a The raw UCB dataset includes resolvers that employ 0x20-encoding [27] as well as a single system conducting high-volume DNS lookups for research purposes. We preprocessed this dataset by removing lookups from the research system (totaling more than 250M) and downcasing lookups from 0x20-resolvers (cf. § 5.3). (Note that the dataset has 3 days with only partial information.)

^b This dataset’s first day starts at 7AM local time rather than midnight. The other days are complete.

^c “Clients” in the SIE dataset instead reflect site resolvers, each with potentially thousands of actual clients.

^d As discussed later, we omit from our evaluation the PTR reverse lookups in this data, which comprise about 10% of the lookups.

3 Threat Model

Our basic model assumes that the attacker controls both a local system and a remote name server. The local system will communicate with the remote name server solely by issuing lookups for DNS names that the site’s resolver will ultimately send to the attacker’s name server. The attacker inspects both the content of these queries (i.e., the names and the associated query type, such as TXT or AAAA) and their arrival timing.

We further assume that the internal system under the attacker’s control makes standard queries, either because the site’s firewalling requires internal systems to use the site’s own resolvers, or because non-standard queries made directly to the public Internet could expose the communication’s anomalous nature.

For the investigation we develop in this work, we focus on communication outbound from local systems. (We briefly discuss inbound communication encoded in DNS replies in § 9.) We view the outbound direction as the most apt when concerned about exfiltration threats. In addition, for the interactive communication scenario, the outbound direction corresponds to the replies generated by a local login server in response to keystrokes sent by a remote client. The outbound traffic volume to the login client is typically 20 times larger than the incoming traffic [21], making DNS queries embedding outbound traffic the larger target for that scenario, too.

We do not consider here communication that an attacker spreads across multiple remote domains or multiple remote name servers (such as `attacker1.com`, ..., `attackern.com`), nor spread thinly across multiple local clients. We discuss these and other evasion issues in § 9.

4 The Data

For our analysis we draw upon datasets that together comprise 230 billion queries. The data was collected at multiple locations across the US and China, with vantage points ranging from internal DNS servers to network perimeters. We summarize each dataset and its daily traffic statistics in Table 1.

INDLAB: an industrial research laboratory. Collected with a network sniffer near an internal DNS server, this dataset contains queries from internal clients, the reply time-to-live, and microsecond-resolution time stamps.

LBL: a national research laboratory. This dataset contains DNS queries from local clients received by several internal DNS servers. Covering a time span of 7.5 years, this is the largest data set in our analysis.

NERSC: a super-computer center. The dataset contains queries from local clients to the site’s DNS servers.

UCB: a university campus. This data was collected on a perimeter network, providing an aggregate view of (outbound) DNS query traffic. This site includes servers that use 0x20 encoding [27], which nearly doubles the number of distinct lookup names.

CHINA: a caching server for several university networks in China, with visibility of individual client IP addresses.

SIE: the Security Information Exchange of the Internet Software Consortium [24]. In this collaboration of infrastructure providers, law enforcement, security companies and researchers, participants² mirror their DNS reply traffic from name servers across the Internet. (Note that each reply contains a copy of the query.)

With a combined average of 1.5 billion replies a day, SIE has by far the highest data rate in our collection.

²Heavily dominated by a single large U.S. ISP.

However, we note that we use it as a means of assessing to what degree our detection procedure indeed can find actual instances of surreptitious communication over DNS; we do *not* claim our procedure is tenable for actual operational use in this environment, which is hugely aggregated across (likely) millions of actual clients.

5 Establishing Communication Bounds

In this section we develop a principled approach for bounding the amount of information possibly conveyed by local systems to remote name servers. The next section then presents a number of filtering steps that reduce the resources required for detecting communication that exceeds these bounds.

5.1 Information Vectors

We first frame the basic communication mechanisms an attacker could employ. In general terms, we consider an attacker who wishes to communicate a significant quantity of information by sending DNS queries to a remote domain (say D.com) whose name server(s) the attacker controls. Such queries provide a number of *information vectors* that the attacker can exploit to surreptitiously embed data within the stream of queries.

We note that attackers can potentially employ multiple vectors at the same time. We emphasize that our detection scheme does *not* presume use of particular encodings for a given vector; the encodings we give here are just meant to illustrate the possibilities.

Query name-content vector. A conceptually straightforward way to embed data is for the attacker to devise a data encoding that conforms with the requirements imposed on DNS labels, limiting each to no more than 63 bytes in length, and complete DNS names to no more than 255 bytes [18]. For example, one could use Base-64 encoded data strings as such as VVNFQwo.D.com.

To our knowledge, all available tunneling-over-DNS tools reflect this style of approach.

Query name-codebook vector. Rather than using each DNS query to reflect a message many bytes long, attackers can encode messages using a fixed alphabet of symbols and then transmit those symbols one at a time using a series of queries. For example, to convey the bit-string 00101111 one bit at a time, a client could issue the queries: z.D.com, z.D.com, o.D.com, z.D.com, o.D.com, o.D.com, o.D.com, o.D.com. They could of course also use larger alphabets to obtain greater efficiency.

Encodings using this vector will in general generate many more lookups of the same names over time compared to those using the query name-content vector.

Query type vector. Along with the query name, clients include in their requests the *type* of DNS Resource Record they wish to resolve, such as PTR for reverse-IP-address-to-hostname mappings, or AAAA to look up

IPv6 addresses. Attackers can encode a modest amount of information per query using this 16-bit field.

Query timing vector. A more subtle information vector exists in the specific timing of queries. For example, if the attacker can resolve the arrival times of queries to 1 sec precision, then the attacker can use the number of seconds between successive queries as a means of conveying information.³

A key issue for timing vectors concerns clock precision. With an extremely precise clock (and sufficiently low jitter), intervals between queries can convey several bytes of information without requiring very large inter-query delays. For example, transmitting one query every second using a clock with 1 msec precision can convey $\lg 10^3$ bits per query, totaling more than 108 KB per day.

Other information vectors. Inspecting the DNS query format reveals several additional fields possibly available for communicating information: query identifiers, a number of flags, options, the query count, and the 16-bit address class field included in each query. We argue that none of these provide a reliable end-to-end information vector for an attacker, given the assumption in our threat model that the attacker’s client must relay its queries via a site’s standard (non-cooperative) resolver. Such relayed queries will not preserve query identifiers. The flags either do not survive the relaying process (e.g., Recursion Desired) or will appear highly anomalous if they vary (e.g., requesting DNSSEC validation), and likewise DNS options (EDNS0) do not survive relaying, as unknown options return an error [26], and the current options themselves are generally implemented on a hop-by-hop basis. Similarly, query counts other than 1 would appear highly anomalous and likely fail to actually propagate through the site’s name server. Likewise, use of any address class value other than IN (*Internet*) would be readily detectable as anomalous.

5.2 Challenge: Diversities Seen in Practice

A natural starting point when attempting to detect surreptitious DNS communication is to posit that the encodings used for the communication will stand out as strikingly different than typical DNS activity. If so, we can target the nature of the encoding for our detection.

What we find, however, is that while potential encodings may differ from *typical* DNS activity, they do not sufficiently stand out from the *diverse range* of benign activity. When we monitor at a large scale—such as analyzing the traffic from the 1000s of systems in an enterprise—we observe a striking degree of fairly extreme forms of DNS lookups.

³In addition, the specific query received after the given interval could also convey additional information using one of the previously described vectors.

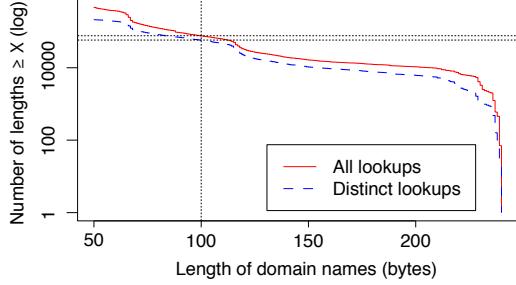


Figure 3: Distribution of the lengths of all individual (solid) or all distinct (dashed) domain name prefixes queried during a sample day of data from LBL. The horizontal lines mark that 76K (all) and 58K (distinct) lookups were ≥ 100 bytes. Lengths do not include the *registered domain* targeted by the lookup. Note that the plot shows the upper 1% of all queries, but the upper 18% of all distinct queries.

To illustrate, we consider DNS activity observed on a sample day in 2011 at LBL. It includes 35M queries issued from 9.4k hosts. These queries in total span 1.2M distinct names, and if we discard the first component of each name, 620K distinct subdomains. These subdomains are themselves rooted in 137K distinct *registered domains* (i.e., one level under com or co.uk).

One natural question concerns the frequency with which operational DNS traffic exhibits peculiarly long query names, since many natural encodings for surreptitious communication will aim to pack as much information into each query as possible. Figure 3 shows the distribution of domain name prefix lengths ≥ 50 bytes (i.e., characters) looked up in our sample day. We see that queries with names even larger than 100 bytes occur routinely: while rare in a relative sense (only 0.2% of query names are this large), 76,523 such queries occurred on that day. Restricting our analysis to distinct names (dashed line) does not appreciably lower this prevalence.

For concreteness, here are some examples of what such queries look like:

```
JohnsonHouse\032Officejet....sonhouse1.members.mac.com
www.10.1.2.3.static.becau....orant.edu.za.research.edu
awyvrvccataaaegdid5tmar7ete....ilu.license.crashplan.com
g63uar2ejiq5tlrk3zezf2fk....emc6pi88tz.er.spotify.com
5.1o19sr00rs95qo0p73415p....7rn92.i.02.s.sophosx1.net
```

where we have elided between 63 characters (first example) and 197 characters (last example). See Appendix A for the complete names.

Thus, simply attempting to detect queries that include unusually large names does not appear viable. Similarly, the examples above illustrate that benign traffic already includes DNS queries that use opaque encodings, so we do not see a promising angle to pursue with regard to recognizing surreptitious communication due to the syntax of its encoding.

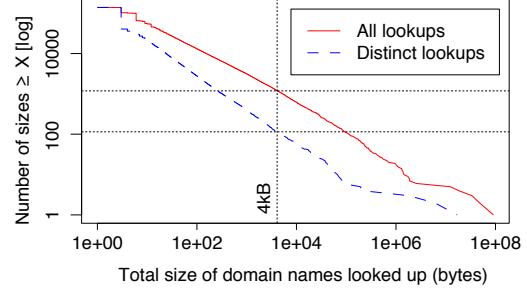


Figure 4: Distribution of the total length of domain name prefixes sent to different registered domains, computed as the sum of all names (solid) or distinct names (dashed). The horizontal lines mark that 1,186 registered domains received ≥ 4 kB of names, while 114 received ≥ 4 kB of distinct names.

A different perspective we might pursue is that if only a small number of remote name servers receive the bulk of the site’s queries, then we might be able to explicitly examine each such set of traffic. Figure 3, however, shows that large volumes of queries are spread across numerous remote name servers. The plot shows how many registered domains received a given total size of queries (the sum of the lengths of all of the prefixes sent to that domain). If we restrict our view to the total size of *distinct* queries that a registered domain receives, more than 100 registered domains each received in excess of 4kB of query names. If we include prefixes for repeated lookups, the figure is ten times higher.

Surprising query diversity also manifests in other dimensions. For example, surreptitious communication that leverages the transmission of repeated queries in a codebook-like fashion requires using low-TTL answers to prevent local caching from suppressing queries. However, we find that in benign traffic, low TTLs are not unusual: in a day of queries for external names that we examined, a little under 1% of the answers had TTLs of 0 or 1, and 38% are ≤ 60 sec. We also find instances of large numbers of repeated queries arising from benign activity such as misconfigurations and failures.

In summary, the variations we find operationally are surprisingly rich—enough so to illustrate that our problem domain will not lend itself to conceptually simple approaches due to the innate diversity that benign DNS lookups manifest when observed *at scale*.

To illustrate the difficulty, we evaluated the performance of a naive detector that simply sums up the volume of lookups sent to each domain, alerting on any client sending the domain more than 4,096 bytes in one day. In steady-state (using the same methodology as in § 7, including the *Identified Domain List* discussed below), this detector produces 200x more alerts than our actual procedure. If we alter the detector to only sum the volume of distinct lookups, we still must abide 5x more

alerts (and lose the ability to detect codebook-style encodings). We emphasize that because our actual procedure has no false negatives, all of these additional alerts represent false positives.

5.3 Establishing Accurate Bounds on Query Stream Information Content

Given that simple heuristic detection approaches will not suffice due to the innate diversity of DNS queries, we now pursue developing principled, direct assessments of upper bounds on the volume of data a given client potentially transmits in its queries.

A key observation is that—provided we do not *underestimate* the potential data volume—we can avoid any false negatives; our procedure will indeed identify any actual surreptitious communication of a given size over DNS. Given this tenet, the art then becomes formulating a sufficiently *tight* upper bound so we do not erroneously flag lookups from a client to a given domain as reflecting a significantly larger volume of information than actually transmitted.

We can obtain tight bounds by quantifying the size of carefully chosen *representations* of a client’s query stream. If we obtain these representations in a *lossless* fashion (i.e., we can recover the original query stream from the representation), then the bound is necessarily conservative in the sense of never underestimating the true information content of the queries. At the same time, the representation must be compact enough to reduce any redundancy from the query stream as efficiently as possible in order to obtain a tight estimate. Thus, the task we face is to determine a representation of the query stream that efficiently captures its elements, but does so in a reversible fashion. In general, we seek forms of *lossless compression* with high compression ratios.

Conceptually, the **heart of our approach** is to take encoded query streams and feed them to compression algorithms such as *gzip*, using the size of the compressor’s output as our estimate. While simple in abstract terms, pursuing this effectively requires (1) care in encoding the streams to try to achieve as tight a bound as possible, and (2) structuring the analysis procedure to execute efficiently given a huge volume of data to process.

For the rest of this section, we address the first of these issues. We then discuss execution efficiency in § 6.

Character casing. The first question regarding encoding query streams concerns the most obvious source of variation, namely the particular names used in the queries. For these, one significant encoding issue concerns *casing*. While the DNS specification states that names are treated in a case-insensitive manner, in practice resolvers tend to forward along names with whatever casing a client employs when issuing the query to the resolver.

Together, these considerations mean that, for example,

a query for `foo.D.com` and `FoO.D.COM` will both arrive at the same `D.com` name server, with the casing of the full query name preserved. Accordingly, we must downcase query name suffixes in order to correctly group them together (i.e., to account for the fact that the same name server will receive them), but preserve casing in terms of computing information content, since indeed the attacker can extract one bit of information per letter in a query (including the domain itself) depending on its casing.

0x20-encoding. Preserving casing in queries can raise a difficulty for formulating tight bounds on information content due to the presence of 0x20-encoding [27], which seeks to artificially increase the entropy in DNS queries to thwart some forms of blind-spoofing attacks. While the presence of arbitrary casing due to use of 0x20-encoding does indeed reflect an increase in the actual information content of a stream of queries, this particular source of variation is not of use to the attacker; they cannot in fact extract information from it.

We found that unless we take care, our UCB dataset, which includes queries from a number of resolvers that employ 0x20-encoding, will indeed suffer from significant overestimates of query stream information content. The presence of such resolvers however means that their clients cannot exploit casing as an information vector, since the resolver will destroy the client’s original casing. Accordingly, we developed a robust procedure (details omitted due to limited space) for identifying queries emanating from resolvers that employ 0x20-encoding. For those query sources we downcase the queries to accurately reflect that casing does not provide any information.

This procedure identified 205 clients in the UCB dataset. Other than those clients, we left casing intact.

Employing codepoints. General compressors such as *gzip* do not make any assumptions about the particular structure of the data they process. However, our particular problem domain has certain characteristics that can improve the compression process if we can arrange to leverage them. In particular, we know that DNS query streams often repeat at the granularity of entire queries. We can expose this behavior to a general compressor by constructing *codepoints*, as follows. We preprocess a given client’s query stream, replacing each distinct query with a small integer reflecting an index into a table that enumerates the distinct names. For example, this would reduce a query stream of `foo.X.com`, `bar.X.com`, `bar.X.com`, `foo.X.com`, `bar.X.com` to the stream 1, 2, 2, 1, 2, plus a dictionary that maps 1 to `foo.X.com` and 2 to `bar.X.com`. The particular encoding we use employs 24-bit integers (we take care in our information-content estimation to include the dictionary size).

Representing query types. For datasets that include query types, we construct a separate, parallel compres-

sion stream for processing the corresponding 16-bit values, i.e., we do not intermingle the query types with the query names.

Representing timing. Individual query timings offer only quite limited information content. Thus, for an attacker to make effective use of timing, they will need to send a large number of queries. This means that we likely will benefit from capturing not absolute timestamps but intervals between queries. We compute such intervals as 32-bit integers representing multiples of \mathfrak{R} , our assumed lower bound on the timing resolution the attacker can achieve. Again we construct a separate, parallel compression stream for processing these.

Clearly, the value of \mathfrak{R} can significantly affect the amount of information the attacker can extract from the timing of queries; but \mathfrak{R} will be fundamentally limited by network jitter. To formulate a defensible value of \mathfrak{R} , we asked the authors of [17] regarding what sort of timing variation their measurements found for end systems conducting DNS queries. Using measurements from about a quarter million distinct IP addresses, they computed the maximum timing difference seen for each client in a set of 10 DNS queries it issued. The median value of this difference across all of the clients was 32 msec. Only a quarter of the clients had a difference under 10 msec. Accordingly, for our study we have set \mathfrak{R} to 10 msec.

Constructing unified estimates. As described above, we separately process the query names, types, and timing. Formulating a final estimated bound on a query stream’s information content then is simply a matter of adding the three corresponding estimates. We note, though, that by tracking each separately, we can identify which one contributes the most significantly (per Figure 6 below).

Bakeoffs. Finally, as outlined above we have several potential choices to make in formulating our upper-bound information estimates: which compressor should we employ? Should we use codepoints or allow the compressor to operate without them (thus not imposing the size of the dictionary)? We note that we do not in fact have to make particular choices regarding these issues; we can *try each option separately*, and then simply choose the one that happens to perform best (generates the lowest information estimates) in a given context. Such “bakeoffs” are feasible since we employ lossless techniques to construct our estimates; we know that each estimate is sound, and thus the lowest of a set is indeed the tightest upper bound we can obtain.

The drawback with trying multiple approaches, of course, is that it requires additional computation. In the next section we turn to how to minimize the computation we must employ to formulate our estimates.

6 Implementation

The previous section described our approach to developing an accurate bounds on the amount of information conveyed using DNS queries to a given domain’s name server(s). Computing these estimates and acting upon their corresponding detections, however, raises a number of issues with regards to reducing the resources required for employing this approach.

In this section we discuss practical issues that arise when implementing our detection approach. One significant set of these concern *filtering*: either restricting the DNS queries we examine in order to conserve computing (or memory) resources, or reducing the burden that our detection imposes on a site’s security analysts. The key property of these filtering stages is their efficacy in concert, which is crucial for the scalability of our approach. Figure 5 shows the different stages of processing in our detection procedure and how they pare down in several steps the volume of both the queries that we must examine and the number of domain name suffixes to consider.

We describe our detection procedure as implemented for off-line analysis here, and discuss our experiences with a real-time detector in § 8.

6.1 Cached Query Filter

A query from a DNS client system cannot exfiltrate information unless it is forwarded by the recursive resolver. Thus a highly useful optimization for the internal vantage point (as discussed in § 4) is to model the recursive resolver’s cache and not consider any query where the resolver obtained the result from its cache.

We can accomplish this by observing the replies with the TTL field. We maintain a shadow cache based on the query attributes (contained in the reply) and the reply TTL values, and do not consider later queries until their information expires from the shadow cache.

The result of this filtering is to eliminate the disadvantage of the internal vantage point, as this filter ensures that later stages only process uncached requests. With the INDLAB dataset, this reduces the number of detections by about 2x for the timing vector, and about 10% for query names. Unfortunately not all of our datasets support this filtering.

6.2 Uninteresting Query Filter

We remove lookups that target domain names within the local organization itself, or within closely-related organizations. Due to their relatively high volume, we find that such lookups can result in a large number of detections, but the likelihood that someone will actually use a DNS tunnel between such domains will be negligible. Likewise, we remove lookups of PTR (address-to-name) records for local and reserved network address ranges.

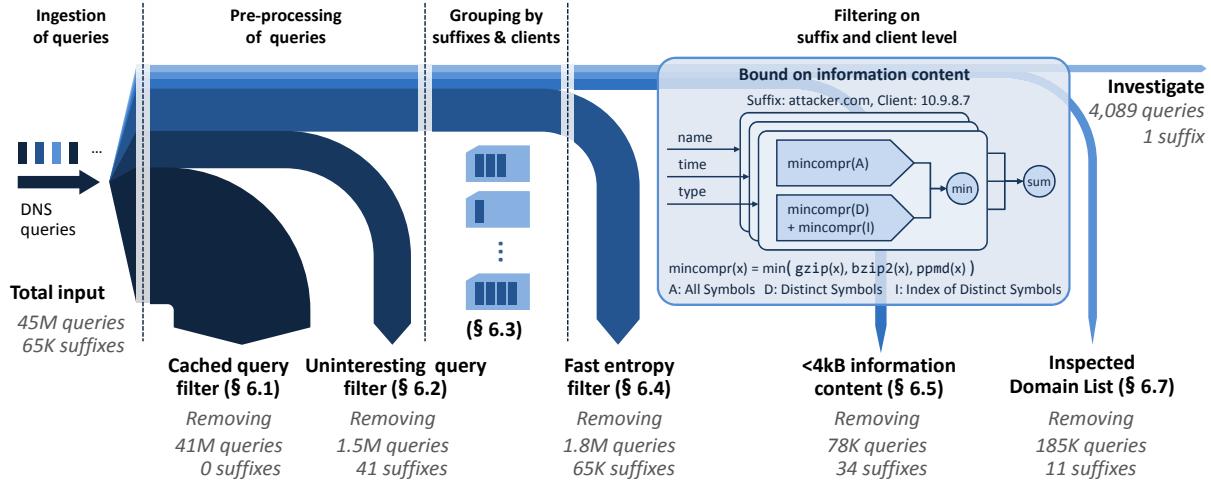


Figure 5: The full detection procedure. The numbers (grey) reflect a day at the INDLAB network for which the detection procedure flagged a new domain name (a relatively rare event).

Finally, we exclude names without a valid global top-level domain. This eliminates numerous queries from systems that are misconfigured or confused.

6.3 Grouping by Suffix and Client

In this stage of our detection procedure, we compute statistics per (lookup name suffix, client)-pair that will serve as input to the lightweight filter described in § 6.4.

Due to the voluminous nature of our data, we aggregate these statistics at the level of registered domain names (e.g., one level under com or co.uk). With IPv4 PTR lookups we aggregate at two and three labels under in-addr.arpa (corresponding with /16 or /24 network ranges), and with IPv6 PTR lookups we aggregate at 12 labels under ip6.arpa (corresponding with /48 network ranges). The reasoning behind these choices is that shorter PTR suffixes will in general represent large blocks that are parents to multiple organizations; thus, the presence of tunneling associated with such suffixes would require compromise of a highly sensitive infrastructure system. In our results for PTR lookups we find no indications of surreptitious communication.

We then compute for each query suffix and client the numbers of unique and distinct lookup names including that suffix, as well as the combined length of those lookup names. We group suffixes in a case-insensitive manner, but count as distinct any lookup names that differ only in case (cf. § 5.3).

6.4 Fast Filtering of Non-Tunnel Traffic

The very high volume of DNS queries means we can obtain significant benefit from considering additional measures for pre-filtering the traffic before we compute the principled bounds described in § 5. For each domain suffix, we use computationally lightweight metrics that overestimate the information content present in the in-

formation vectors described in § 5.1. We then compare the sum of these metrics across all information vectors against a *minimum-information content threshold*, \mathcal{T} . If the sum total (guaranteed to not underestimate) lies below the threshold, the traffic for the corresponding domain suffix cannot represent communication of interest. This approach allows us to short-circuit the detection process and eliminate early on numerous domain suffixes.

Fast filter for the query name vector. We consider the following quantities from a sequence of lookups made by some host during one day: the total number of lookups L , the number of distinct query names D_{name} in those lookups, and the total number of bytes C_{name} in those distinct query names. We remark that we can determine all three quantities with minimal computational and memory overhead.

Query name tunnels encode information in terms of the characters and the repetition patterns of the names looked up. Each character in a name may convey up to 1 byte of information, contributing up to C_{name} bytes in total. According to Shannon's law, the number of bits conveyed per lookup amounts to at most $\log_2 D_{\text{name}}$. Therefore the combined upper bound on information conveyed in bytes by such a tunnel amounts to:

$$I_{\text{name}} = C_{\text{name}} + L \cdot \frac{\log_2 D_{\text{name}}}{8}$$

Fast filter for the query type vector. We filter the query type vector similarly. Again, we consider a sequence of DNS lookups with a given suffix made by some host during one day. If we use D_{type} to denote the number of distinct query types in those lookups and C_{type} the total number of bytes in those distinct query types, we have:

$$I_{\text{type}} = C_{\text{type}} + L \cdot \frac{\log_2 D_{\text{type}}}{8}$$

Fast filter for the query timing vector. The timing vector is more complicated because we need to discretize the time information and create symbols representing the encoded data as it appears in the timing vector. We parametrize this process by the *time resolution* \mathfrak{R} that the network environment affords to the attacker.

Intuitively, for a given number of lookups L observed over a day, the amount of potential information encoded in time is maximal when the number of distinct inter-arrival times, k , is maximal. This is due to the fact that, without knowing the distribution of inter-arrival times, the empirical entropy from the inter-arrival times may be upper-bounded by $L \cdot \log_2 k$, where $\log_2 k$ is the number of bits encoded by a single lookup.

As a consequence, to assess the upper-bound on the information content for a fixed L and an assumed time-slot size (expressed as time resolution \mathfrak{R}), we need to determine into how many distinct inter-arrival times k we can partition one day into, while imposing as uniform a distribution of inter-arrival times as possible (i.e., leading to maximal entropy).

By maximizing k subject to the constraint that the distribution of distinct inter-arrival times is uniform (omitting details for brevity), and upper-bounding k by $L - 1$ (the number of intervals), we find that we can express the upper bound on the information amount in the timing vector as:

$$I_{\text{time}} = L \cdot \log_2 \left(\min \left(L - 1, \left\lfloor \frac{2M}{L - 1} \right\rfloor + 1 \right) \right)$$

where $M = \frac{86,400}{\mathfrak{R}}$ denotes the number of time slots with resolution \mathfrak{R} over one day (86,400 seconds).

Unified fast filter. From the above equations, we can now formulate the following unified test condition to handle all types of information vectors:

If $I_{\text{name}} + I_{\text{type}} + I_{\text{time}} < \mathfrak{J}$, the suffix is not a candidate tunnel.

We then eliminate from further detailed analysis the name suffixes that are not candidate tunnels.

Choosing the thresholds. The fast filter relies on two parameters, the information content threshold \mathfrak{J} and the time resolution \mathfrak{R} . In order to select security-relevant values for these parameters, we measured their impact on the analyst’s workload. (Note that in § 5.3 we also framed empirical evidence that $\mathfrak{R} = 10$ msec appears fairly conservative.) It is clear that both reducing the information content threshold and reducing the time resolution can increase the false positive rate, and relatedly the analyst’s workload.

Figure 6 shows how varying these parameters affects the analyst for INDLAB data. One can see, for example, that decreasing the information content threshold \mathfrak{J}

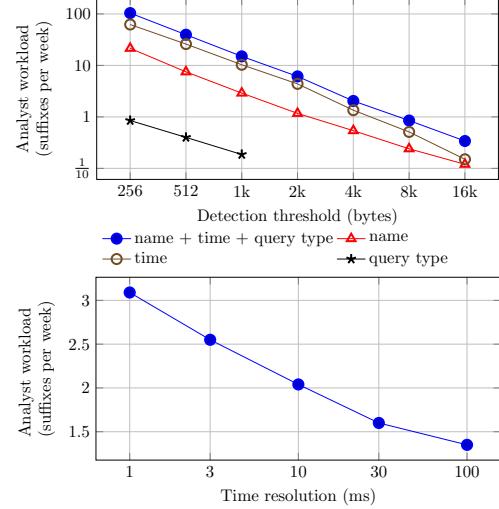


Figure 6: The impact of the information content threshold \mathfrak{J} and the time resolution \mathfrak{R} on the number of suffixes to validate manually per week for the INDLAB dataset. The top chart reflects a value $\mathfrak{R} = 10$ msec, and the bottom chart $\mathfrak{J} = 4,096$ bytes.

from 4,096 to 256 bytes (and potentially increasing security) would increase the number of domain name suffixes passed to the analyst for manual inspection 50-fold. The plot also shows a clear power-law relationship between analyst workload and \mathfrak{J} , with the former scaling as approximately $x^{-1.38}$ in the latter.

Setting the information content threshold \mathfrak{J} to 4,096 bytes and the time resolution \mathfrak{R} to 10 ms thus provides a good balance between analyst workload and potential detections. Sites might of course revisit these parameters based on their particular threat models and networking environments.

6.5 Bounding Information Content

For each (suffix, client)-pair that remains after the preceding filter steps, we compute the size of *gzip*, *bzip2* and *ppmd* [23] compression for the series of all corresponding lookup names, selecting the lowest value. We also assess a codepoint-oriented analysis (§ 5.3), computing the *gzip*, *bzip2* and *ppmd* compression sizes for the series of distinct (unique) lookup names, selecting the lowest value, and adding the lowest value of the *gzip*, *bzip2* and *ppmd* compression sizes for the corresponding distinct lookup name indices. Given these two assessments, we choose the smaller as the best (tightest) upper bound on the amount of information potentially transferred through lookup names to the given domain suffix (cf. box ‘‘Bound on Information Content’’ in Figure 5).

Next, we apply the same procedure to the corresponding inter-query arrival times (in $\mathfrak{R} = 10$ msec units) and query record types, if this information is available. Fi-

nally, we add up the results from the lookup name, time and type information vectors, and if their sum lies below \mathcal{I} , we discard the (suffix, client)-pair.

6.6 Inspected Domain List

We expect sites to employ our analysis procedure over an extended period of time. For example, once a site sets it up, it might run as a daily batch job to process the last 24 hours of lookups. An analyst inspects the traffic associated with any domains flagged by the procedure and renders a decision regarding whether the activity appears benign or malicious.

An important observation is that the same benign domains will often reappear day after day, due to the basic nature of their lookups. However, the analyst needn’t reexamine such domains, as the verdict will prove the same. (See § 9 for further discussion of this point.) Given this, we presume the use of an *Inspected Domain List* (IDL) that accumulates previous decisions regarding domains over time. For a given day’s detections, we omit flagging for the analyst any that already appear on the IDL. Once populated, such a dynamic list can greatly reduce the ongoing burden that our detection procedure places on a site’s analysts.

A final issue regarding the IDL concerns its granularity. For example, if our procedure flags s1.v4.ipv6-exp.l.google.com and we put that precise domain on the IDL, then this will not spare the analyst from having to subsequently investigate i2.v4.ipv6-exp.l.google.com.⁴ However we note that the analyst’s decision process will focus heavily on *registered domains*. In this example, the analyst will likely quickly decide to mark the detection as benign because for it to represent an actual problem would require subversion of some of Google’s name servers, which would represent an event likely significantly more serious than an attacker communicating surreptitiously out of the site. In addition, the analyst will reach this conclusion simply by inspecting the registered domain google.com, rather than studying all of the sub-domains in depth.

Accordingly, once an analyst inspects a detection, we place on the IDL the corresponding registered domain, which we compute by consulting Mozilla’s *Effective TLD Names* list [20]. In this example, com appears on the list (meaning that any domain directly under it will reflect a registration), so we add google.com to the IDL. Any subsequent matching against the IDL likewise employs trimming of names using the same procedure.

We note that we could implement the IDL with finer granularity than described above. In particular, we could frame it in terms of per-client filtering, or using custom entropy thresholds. We leave exploring these refinements for future work.

⁴ Both of these are actual detections.

Exfiltration Scenario	Estimated Data Volume			
	Total	Name	Timing	Type
Query name-content	111%	110%	0.4%	0.01%
Query name-codebook	109%	103%	5.6%	0.1%
Timing	105%	0.8%	104%	0.2%
Query type	111%	0.6%	6.8%	104%

Table 2: Estimates of data volumes produced by our procedure measured against specific exfiltration scenarios, showing the total estimate, and the individual contributions from the query name, timing, and type information estimation.

7 Evaluation

In this section we evaluate the efficacy of our detection procedure in terms of assuring that it can detect explicit instances of communication tunneled over DNS (§ 7.1) and investigating its performance on data from production networks (§ 7.2). For this latter, we assess both the procedure’s ability to find actual surreptitious communication, and, just as importantly, what sort of burden it imposes on security analysts due to the events generated.

7.1 Validating on Synthetic Data

To validate our procedure’s ability to accurately measure communication embedded in DNS queries, we assessed what sort of estimates it produces for scenarios under which we fully control the DNS communication used for exfiltration. Table 2 summarizes the results, comparing the information vector used for exfiltration vs. the estimates of the volume of data present in the corresponding lookups, both in total and when restricted to just considering a single information vector. All values are percentages of the actual exfiltration size, so a value of 105 indicates an estimate that was 105% of the true size (i.e., the estimate was 5% too high). Naturally, estimators that focus on information vectors different from those used in a given exfiltration scenario can greatly underestimate the data volume if used in isolation, highlighting the need to combine such estimators into a final comprehensive sum.

Regarding the scenarios reflected in the table, to assess tunnels based on encoding information directly in query names, we recorded Iodine [10] queries while sending a 99,438-byte compressed file with *scp*. The 11 % difference (shown in the “Query name-content” row) between measured content and actual content is nearly all due to tunnel encapsulation overhead (SSH, TCP/IP headers, Iodine framing). As we are not aware of any available tunneling tools that leverage repeated (codebook-style) queries, timing, or varying query types, we wrote simple proof-of-principle implementations for testing purposes. The codebook-style implementation used 16 distinct names that each convey four data bits per query,

Type Of Activity \ Dataset	INDLAB	LBL	NERSC	UCB	CHINA	SIE	SIE ^{UNIQ}
Lookups (days)	57B (1,212)	73B (2,565)	12B (1,642)	1.7B (45)	69M (5)	77B (53)	12B (53)
Detection threshold	4kB	4kB	4kB	10kB	10kB	10kB	10kB
Confirmed DNS channel	0	2	0	0	0	57	57
Benign use	286	306	29	200	41	4,815	1,088
Malware	2	2	0	5	2	74	73
Misconfiguration	49	62	5	126	8	310	182
IPv4 PTR	11	29	4	26	3	N/A	N/A
IPv6 PTR	0	5	0	1	0	N/A	N/A
<i>Unknown</i>	14	27	0	13	13	1	1
Total	362	433	38	371	67	5,256	1,401
Domains flagged (first week)	16	5	3	199	(67+)	3,002	798
Domains flagged (typical week)	2.0	1.1	0.15	32	N/A	358	97

Table 3: Number of domains flagged in each dataset, broken out by the type of activity that the use of the domain represents. The INDLAB, UCB and CHINA analyses cover all information vectors; LBL and NERSC incorporate query names and types, but not timing; SIE considers only query names; and SIE^{UNIQ} only the contents of query names (not repetitions). SIE and SIE^{UNIQ} analyses includes additional considerations discussed in the Appendix.

while the timing-interval implementation used one name and 16 distinct time intervals spaced 10 ms apart. The query-type implementation used one name and 16 distinct query types. In addition, these tunnels used five distinct query names for command and control. We exfiltrated a 10,000-byte compressed file and found that the difference between the estimated exfiltration volume and the actual size ranged from 5–11 %.

These results confirm that our procedure can readily detect information that is encoded into query names, timing, or query record types, and that it can provide meaningful upper bounds.

7.2 Evaluation on Operational Data

We now turn to evaluating our detection procedure as applied to the extensive datasets we gathered, comprising 230 billion lookups from the networks listed in Table 1.

A key question for whether our detector is operationally viable concerns the combination of (1) how many domains it flags for analysis, coupled with (2) how quickly an analyst can identify the common case of a flagged domain not in fact posing a threat.

The filtering steps in § 6 aim to address the first issue. Regarding the second issue, as we briefly discussed in § 6.6 we find that often analysts can rely on *fate-sharing* to quickly determine they needn’t further investigate a candidate domain. For example, a site’s analyst can reason that a detection of google.com or mcafee.com is safe to ignore, because if indeed an attacker has control over those domains’ name servers, the site has (much) bigger problems than simply the presence of surreptitious communication to the sites.

Table 3 summarizes the findings across all of the datasets. For each dataset, the row in bold gives the total

number of different domains flagged by our detector (many appear in more than one day), and the bottom row reflects the “steady state” burden on an analyst investigating detections for the given environment. We partition the datasets into two groups. The logs for INDLAB, LBL and NERSC include individual per-client lookups, and thus these sites represent the sort of environments for which we target our detection, using a threshold of 4 kB/day. The lookups recorded for UCB, CHINA and SIE, on the other hand, are primarily aggregated across many clients, and thus for these datasets we cannot perform per-client analysis. We do not aim to treat these datasets as operational environments for our detection procedure, but rather to assess what sort of surreptitious communication the procedure *can* detect in real traffic. For them, we use a higher threshold of 10 kB/day to limit our own analysis burden in assessing the resulting detections. Finally, the SIE dataset introduces some additional complexities, as discussed in Appendix B.

We classified the detections based on manual analysis to assign each to one of six general categories, as follows.

Confirmed DNS channel reflects domains for which we could amass strong evidence that indeed the detection represents surreptitious communication over DNS. For LBL, both flagged domains correspond to tunnels that staff members acknowledge having set up to obtain free Internet access in WiFi hotspots that allow out DNS traffic without requiring payment. One used DNStunnel [8], the other NSTX [13].

For SIE, we identified 3 types of tunnels. One type (responsible for 42 domains) corresponds to a product offered by Dynamic Internet Technology, a company that builds tools to evade censorship [9]. These tunnels encode most requests in two 31-character labels, using only

alphanumerics, followed by an identifier that appears to identify the tunnel itself. Another 10 domains all have whois information leading to MMC Networks Limited (of Gibraltar), a company that provides a program offering “Free WiFi” using tunneling [28]. The tunneling technology used for these is a variant of Iodine, with the main difference being use of only alphanumeric characters for the encoding. We also found 5 domains that use Iodine, for reasons we have not been able to identify.

Finally, we examined an addition 150 billion DNS records captured in a separate 259 days of monitoring from SIE. Due to monitoring gaps, this expanded data is unsuitable for analyzing long-term analyst burden. But in it (using a somewhat higher detection threshold) we detected 42 new tunnel instances, including a new tunnel type belonging to vpnoverdns.com.

Benign use encompasses a number of different scenarios that we believe would lead an analyst to fairly quickly decide that the corresponding activity does not appear problematic. These scenarios include flagging of: (1) a well-known site (e.g., google.com), for which a name server breach would reflect a catastrophe, so very likely has not occurred (*fate-sharing*). (2) A sister site (e.g., a partner institute), where a similar argument holds. (3) ISPs, for which sometimes local systems look up many hostnames corresponding to end-user systems. For example, in LBL we observe queries for numerous names such as 201-11-50-242.mganm703.dsl.brasiltelecom.net.br. (4) Directory-style services offered over DNS, including blocklists, user-generated content, and catalogs. (5) Software license servers. (6) Cloud-based antivirus services.

Malware indicates lookups associated with malware activity or sites flagged (for example, by McAfee’s *SiteAdvisor* service) as malicious. For SIE these also include lookups such as p9b-8-na-5w-2z3-djmu-...-njx-2es.info, i.e., 62-character labels consisting of letters or numbers separated by dashes. We concluded that these lookups reflect malware activity because names following the same pattern appeared in a trace generated by a researcher running bots within a contained environment.

Misconfiguration generally reflect clients making large volumes of lookups due to configuration problems that lead to repeated failures. For example, in one LBL instance we observed more than 60,000 lookups of 33 different names within a single domain, such as _ldap._tcp.standardname-...isi.fhg.de. Other problems we observed include lookups apparently based on email addresses, such as itunes@new-music.itunes.com; subdomains appearing to be IP addresses; repeated failures of names with narrow, rigid structures; and domains in search paths that have lookups encapsulating a client’s entire stream of queries sent to other domains.

IPv4 PTR and IPv6 PTR reflect lookups under the in-

addr.arpa and ip6.arpa zones, respectively. These zones provide a decentralized mapping from numeric IP addresses to domain names. As discussed in § 6.2, we do not flag PTR lookup suffixes that correspond to address ranges that are local to the organization, or that are reserved. As noted in § 6.3, for IPv4 PTR lookups we only flag suffixes corresponding to /16 or /24 netblocks, and for IPv6, /48 netblocks.

Unknown reflects domains for which we could not arrive via manual analysis at a confident determination regarding how to classify the activity. For example, one striking instance concerns a number of domains (primarily seen in CHINA traffic, but also SIE) that issue thousands of lookups such as:

`wojn1befrhpfumrupmsn.0ule365.net`

`jnr1ciinsszxahnfrvxe.0ule365.net`

`okgjeqckeqrxdigktkua.0ule365.net`

Here, the domain (0ule365.net) is associated with a Chinese gaming site. Other instances following the same pattern appear to be associated with *phishing* sites related to such gaming sites.

Domains flagged in first week and in typical week reflect the two extreme behaviors of our *Inspected Domain List* approach (§ 6.6). In the first week of operation our detector reports a peak number of domains; once the list is primed, it flags domains at a much lower rate. (We special-case the figure for CHINA because that entire dataset spans less than a week.)

Finally, the main conclusion we highlight regarding the **Total** row is the low number of events that analysts would have to inspect. (Even for SIE, the average load aggregated across the more than 100 participating sites comes to about 50 detections per day, given \mathcal{I} increased from 4 kB to 10 kB.)

8 Real-Time Operation

As developed so far, our analysis procedure operates in an offline fashion, processing full days as a single unit. While this suffices to enable analysts to detect DNS exfiltration on a daily basis, real-time detection would enable immediate identification of such activity and thus much quicker response. In this section we explore the viability of adapting our scheme for such detection.

Our real-time variant uses *gzip* and *bzip2* as the compression functions. We can adapt both the cached query filter and the “uninteresting query” filter to streaming operation, with the only consideration being that we modify the cached query filter to actively flush all cache entries as their TTLs expire to minimize statekeeping.

Adapting the fast filter and the compression-based filters takes more consideration, since they naturally process entire sets of activity as a unit. In addition, if we try to use a compressor in a stream fashion, we must deal with the compressor’s destructive operation: if we add

data to a stream and call `flush()` to obtain the size of the compressed result, the `flush()` operation changes the compressor’s internal state—adding more data and calling `flush()` again can produce a larger output than simply compressing all of the data at once.

Our approach combines the fast filter and the compression measurement for each (domain, client) pair as follows. Initially, for each pair we only track the uncompressed input. Upon receiving new input, we check whether the total message length plus maximum possible entropy contribution from the timing, and query, and query type could possibly lead to the pair generating an alert. If not, we simply append the new information to the list of previously seen queries.

If the total could cross our threshold, we allocate compressors, feed them all of the recorded input, and invoke `flush()`. If the resulting entropy lies below the alert threshold, we simply update the uncompressed data threshold that could possibly generate an alert, discard the compressed data, and continue. Otherwise, we generate an alert, create new compressors, feed them all the previous data, and pass all subsequent data to them as it arrives. These new compressors allow us to compute a full 24-hour entropy total for the (domain, client) pair to aid the analyst. After 24 hours we generate a summary for each pair and discard the associated state.

For good performance we parallelized this approach, running the cached-query and uninteresting-query filters in a single process that dispatches each (suffix, client) pair to one of 15 distinct child processes. We verified that the implementation produces a consistent analysis by processing the same day of INDLAB data using both the original batch implementation (with only `gzip` and `bzip2`) and the real-time variant (70M DNS queries, 36M non-empty replies). They fully agreed, with the real-time implementation requiring 28 minutes and 4.5GB of RAM to process the day of traffic. The execution totaled 53 CPU-core-minutes on a dual processor Intel Xeon X5570 system. Given these results, we conclude that real-time operation is quite viable.

9 Discussion

This paper demonstrates how we can comprehensively measure the information content of an outbound DNS query stream. Our lossless compression-based procedure measures all information that an attacker can effectively send via names, types, and timing, regardless of the actual encoding used. This procedure also has only two tuning parameters, the threshold of detection and the timing precision.

Some minor DNS features remain that we have not included in our analysis procedure. We have omitted these for simplicity, since in their usual (benign) use, they appear almost always to have a single value for a

given client. These information vectors include requesting DNSSEC information (single bit) and the query’s *class* (which for modern traffic is almost always type `TN`, “Internet”). Similarly, future EDNS0 extensions could appear that recursive resolvers will forward intact, providing a new information vector. For all such features, we can simply employ an additional compressor optimized with the use of a very low-cost special case of using a single bit to indicate that for a given client, the feature never changes.

Attackers can tunnel information in DNS replies as well as in queries, and indeed existing tunnels do so. Since replies can include domain names (returned for example in CNAME records) or unstructured byte strings (e.g., `TXT` records), replies can potentially convey large volumes of data. (We remind the reader that in this work we have focused on analyzing DNS queries rather than responses since for the scenarios of particular interest—exfiltration or remote interactive access—the query streams will generally carry the bulk of the data.)

Attackers who can successfully mimic the appearance of benign data-rich query streams (such as block-list lookup services) can trick analysts into deeming their surreptitious communication as harmless. Similarly, an attacker who compromises a previously benign domain can encode their traffic using the same style of lookups as the domain originally used. These problems are orthogonal to the question of *flagging* the activity.

Attackers aware of our detection procedure can in addition design their tunnels to keep the information content below the 4 kB per day threshold. Given that we aggregate information content metrics per domain, a simple evasion strategy would be to spread the traffic across $K > 1$ domains, and then send < 4 kB per day to each, but in aggregate communicate K times that volume. A possible detection approach we envision pursuing consists of analyzing each client’s lookups in their entirety, rather than on a per-destination-domain basis. Coupled with an expanded Inspected Domain List (§ 6.6) to remove the major contributors to DNS traffic, we would aim with this approach to compute a bound on the total information content each client communicates via all of its external DNS queries.

Finally, attackers could spread their exfiltration across multiple compromised clients, so that each client’s query stream remains below the detection threshold. Our experiences with external vantage points such as UCB indicates that we still might be able to find the activity of groups of clients, since that vantage point already aggregates multiple clients into a single apparent source. However, a combination of using multiple compromised clients and K external name servers might prove exceedingly difficult to detect for the sort of thresholds we have employed in this work.

10 Related Work

Four areas of prior work have particular relevance to our study: covert communication; designing ways of tunneling communication over DNS traffic; detecting such tunneling; and establishing bounds on the volume of covert communication.

We adopt Moskowitz and Kang’s classification of covert communication channels [19]. In particular, a storage channel is a covert channel where the output alphabet consists of different responses all taking the same time to be transmitted, and a timing channel is a covert channel where the output alphabet is made up of different time values corresponding to the same response. Accordingly, we treat covert communication via DNS query content (name, type and other attributes) as a storage channel, and covert communication via query timing as a timing channel.

Conventional DNS tunnels are similar in construction: they are bi-directional, directly embedding the outbound information flow in query names, and the inbound flow in server responses. In the absence of outbound data, the client sends low-frequency queries to poll the tunnel server for any pending data. The functionality of these tunnels ranges from a simple client-to-server virtual circuit to full IP-level connectivity. Examples are NSTX [13], dns2tcp [7], Iodine [10], OzymanDNS [15], tcp-over-dns [25], and Heyoka [14]. DNS exfiltration has also been a tool in the attacker’s toolbox for a number of years (per [22] and the references therein).

Beyond query names, the DNS message format contains a variety of fields that could be used for embedding data (as we detail in § 5.1). In addition to the DNS-specific message fields, timing (e.g., the timing of queries) provides a rich vector for embedding data. This is not unique to DNS traffic, but present in all Internet traffic, allowing any message to be encoded in the inter-arrival times between packets. Gianvecchio et al. [12] showed how to automatically construct timing channels that mimic the statistical properties of legitimate network traffic to evade detection. Our detection technique avoids such complication by measuring information content rather than particular statistical properties.

One approach for detecting covert communication over DNS examines the statistical properties of DNS traffic streams. Karasaridis et al. propose DNS tunnel detection by computing hourly the Kullback-Leibler distance between baseline and observed DNS packet-size distributions [16]. To defeat such temporal statistical anomaly detectors, Butler et al. propose stealthy half-duplex and full-duplex DNS tunneling schemes [5]. They also propose the use of Jensen-Shannon divergence of per-host byte distributions of DNS payloads to detect tunneled traffic. Their detection technique only flags whether the aggregate traffic contains tunneled communication; it

does not identify the potential tunneled domains. In addition, the detection rate depends to a large extent on the ratio of tunneled traffic to normal traffic. In [3], the authors show that domain names in legitimate DNS queries have 1-, 2-, and 3-gram fingerprints following Zipf distributions, which distinguishes them from the higher-entropy names used in DNS tunneling. The evaluations in these works do not particularly address practicality for operational use, however, since the authors validate their hypotheses on short, low-volume benign and synthetic tunneled traces collected using free DNS tunneling tools. As we discuss in § 5.2, large-scale DNS traffic often exhibits extensive diversity in multiple dimensions, which likely will exacerbate issues of false positives.

Our work overlaps with work on algorithmically-generated domain names by Yadav et al. [29]. The most salient difference is that their algorithm assumes a specific model of name construction (distributions of letters and bigrams). Instead of focusing on specific name patterns and missing communication that uses different encodings, we measure the aggregate information content of a query stream regardless of how encodings are generated for the query name, type or timing.

Detection of timing channels has been studied before, and we mention here only a few recent results. Cabuk et al. [6] observe that timing-based tunnels often introduce artificial regularity in packet inter-arrival times and present detection methods based on this characteristic. More generally, Gianvecchio and Wang [11] identify timing-based tunnels in general Internet traffic (not just DNS) by using conditional entropy measures to identify the subtle distortions introduced by the tunnel in packet inter-arrival time distributions. These works use time intervals of 20 msec or more; we use a more conservative 10 msec timing resolution, and do not assume the presence of detectable distortions.

While the general problem of surreptitious communication has received extensive examination in the literature of covert channels and steganography, more closely related to our work is previous research on bounding the volume of surreptitious communication in other protocols. Borders et al. studied this problem for HTTP, observing that covert communication is constrained to the user-generated part of an outgoing request [1, 2]. By removing fixed protocol data and data derived from inbound communication, the authors show how to determine a close approximation to the true volume of information flows in HTTP requests. An analogous approach for our problem domain would be to track the domain names a system receives from remote sources (such as web pages and incoming email), and to exclude lookups for these names as potentially conveying information. Such tracking, however, appears infeasible without requiring extensive per-system monitoring.

11 Summary

We have presented a comprehensive procedure to detect stealthy communication that an adversary transmits via DNS queries. We root our detection in establishing principled bounds on the information content of entire query streams. Our approach combines careful encoding and filtering stages with the use of lossless compression, which provides guarantees that we never underestimate information content regardless of the specific encoding(s) an attacker employs.

We demonstrated that our procedure detects conventional tunnels that encode information in query names, as well as previously unexplored tunnels that repeatedly query names from a fixed alphabet, vary query types, or embed information in query timing. We applied our detection procedure to 230 billion lookups from a range of production networks and addressed numerous challenges posed by anomalous-yet-benign DNS query traffic. In our assessment we found that for datasets with lookups by individual clients and a threshold of detecting 4 kB/day of exfiltrated data per client and domain, the procedure typically flags about 1–2 events per week for enterprise sites. For a bound of 10 kB, it typically flags 50 per day for extremely aggregated logs at the scale of a national ISP. In addition, buried within this vast number of lookups our procedure found 59 confirmed tunnels used for surreptitious communication.

Acknowledgments

Our thanks to Partha Bannerjee, Scott Campbell, Haixin Duan, Robin Sommer, and James Welcher for facilitating some of the data and processing required for this work. Our thanks too to Christian Rossow and the anonymous reviewers for their valuable comments.

This work would not have been possible without the support of IBM’s *Open Collaboration Research* awards program. In addition, elements of this work were supported by the U.S. Army Research Office under MURI grant W911NF-09-1-0553, and by the National Science Foundation under grants 1161799, 1223717, and 1237265. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the sponsors.

References

- [1] BORDERS, K., AND PRAKASH, A. Towards Quantification of Network-Based Information Leaks via HTTP. In *Proceedings of the 3rd USENIX Workshop on Hot Topics in Security* (2008), USENIX Association.
- [2] BORDERS, K., AND PRAKASH, A. Quantifying Information Leaks in Outbound Web Traffic. In *Proceedings of the IEEE Symposium on Security and Privacy* (2009), USENIX Association.
- [3] BORN, K., AND GUSTAFSON, D. Detecting DNS Tunnels Using Character Frequency Analysis. In *Proceedings of the 9th Annual Security Conference* (2010).
- [4] BROMBERGER, S. DNS as a Covert Channel Within Protected Networks. http://energy.gov/sites/prod/files/oeprod/DocumentsandMedia/DNS_Exfiltration_2011-01-01_v1.1.pdf, 2011.
- [5] BUTLER, P., XU, K., AND YAO, D. Quantitatively analyzing stealthy communication channels. In *Proceedings of International Conference on Applied Cryptography and Network Security* (2011).
- [6] CABUK, S., BRODLEY, C. E., AND SHIELDS, C. Ip covert timing channels: design and detection. In *Proceedings of the 11th ACM conference on Computer and communications security* (New York, NY, USA, 2004), CCS ’04, ACM, pp. 178–187.
- [7] DEMBOUR, O. DNS2tcp. <http://www.hsc.fr/ressources/outils/dns2tcp/index.html.en>.
- [8] DNStunnel. <http://www.dnstunnel.de/>.
- [9] Dynamic Internet Technology. <http://www.dit-inc.us/>.
- [10] EKMAN, E., AND ANDERSSON, B. Iodine, tunnel IPv4 over DNS. <http://code.kryo.se/iodine/>, 2011.
- [11] GIANVECCHIO, S., AND WANG, H. An entropy-based approach to detecting covert timing channels. *Dependable and Secure Computing, IEEE Transactions on* 8, 6 (Nov/Dec. 2011), 785–797.
- [12] GIANVECCHIO, S., WANG, H., WIJESEKERA, D., AND JAJOEDIA, S. Model-based covert timing channels: Automated modeling and evasion. In *Proceedings of the 11th international symposium on Recent Advances in Intrusion Detection* (Berlin, Heidelberg, 2008), RAID ’08, Springer-Verlag, pp. 211–230.
- [13] GIL, T. NSTX (IP-over-DNS). <http://thomer.com/howtos/nstx.html>.
- [14] Heyoka. <http://heyoka.sourceforge.net/>.
- [15] KAMINSKY, D. OzyManDNS.
- [16] KARASARIDIS, A., MEIER-HELLSTERN, K., AND HOEFLIN, D. Detection of DNS anomalies using flow data analysis. In *Global Telecommunications Conference (GLOBECOM)* (2006).
- [17] KREIBICH, C., WEAVER, N., NECHAEV, B., AND PAXSON, V. Netalyzr: Illuminating the edge network. In *Proceedings of the ACM Internet Measurement Conference (IMC)* (Melbourne, Australia, November 2010), pp. 246–259.
- [18] MOCKAPETRIS, P. Domain names—implementation and specification. RFC 1035, Internet Engineering Task Force, Nov. 1987.
- [19] MOSKOWITZ, I. S., AND KANG, M. H. Covert channels - here to stay? In *Proceedings of the Ninth Annual Conference on Computer Assurance* (1994), pp. 235–244.
- [20] MOZILLA. Public Suffix List. Published online at <http://publicsuffix.org/>. Last accessed on May 4, 2012.

- [21] PAXSON, V. Empirically-Derived Analytic Models of Wide-Area TCP Connections. *IEEE/ACM Transactions on Networking* 2, 4 (Aug. 1994), 316–336.
- [22] RICKS, B. DNS Data Exfiltrationa Using SQL Injection. <http://www.defcon.org/images/defcon-16/dc16-presentations/defcon-16-ricks.pdf>, 2008.
- [23] SHKARIN, D. PPMD. <http://www.compression.ru/ds/ppmdj1.rar>, 2006.
- [24] Security Information Exchange. <http://sie.isc.org/>.
- [25] tcp-over-dns. <http://analogbit.com/software/tcp-over-dns>.
- [26] VIXIE, P. Extension Mechanisms for DNS (EDNS0). RFC 2671 (Proposed Standard), Aug. 1999.
- [27] VIXIE, P., AND DAGON, D. Use of Bit 0x20 in DNS Labels to Improve Transaction Identity. Work in progress, Internet Engineering Task Force, 2008.
- [28] Wi-Free. <http://wi-free.com/>.
- [29] YADAV, S., REDDY, A. K. K., REDDY, A. N., AND RANJAN, S. Detecting algorithmically generated malicious domain names. In *Proceedings of the 10th annual conference on Internet measurement* (2010), IMC ’10, ACM, pp. 48–61.

A Full Names for Examples

For completeness, Figure 7 lists the full names of various DNS lookups that in the main body of the text we elided portions for readability. Note that for some names we introduced minor changes for privacy considerations.

B Issues Evaluating the SIE Dataset

The SIE data’s extreme volume and qualitatively different nature necessitated several changes to our analysis procedure. Our access to the data was via a Hadoop cluster, requiring coding of our algorithms in the Pig and Scala languages. These provide efficient support for only a subset of the functionality we employed when analyzing the other datasets. A significant difference in this regard was that we were confined to only using *gzip* for compression; *bzip2* and *ppmd* were not available.

Another important difference concerns the definition of “client”. A single large American ISP dominates the SIE data, representing roughly 90% of the traffic. This ISP uses clusters of resolvers to process requests. Thus, a single abstract resolver manifests as multiple “client IP addresses”, which we determined come from the same /28 address prefix. Therefore we treat query source IP addresses equivalent in their top 28 bits as constituting a single source.

This extreme aggregation leads to significant increases in detections, as we are now measuring the information volume for queries aggregated across potentially hundreds of thousands of clients. One particular increase in benign alerts arises due to popular names with short TTLs (e.g., www.google.com). With so many clients, every popular name becomes immediately refetched whenever its TTL expires, leading to a steady stream of closely-spaced lookups. This very high level of aggregation also generates such a large volume of detections for

```
5.1o19sr00ors95qo0p73415p3r8r8q777634r5o86osn295ss2rqos
s3r9601ro3.1rlp7r4719o34393648s2345nn60qnqoop45psos37n
551s002n80850sr2r8n3.r1105qqq28r7pn82843rp76383qr6344q
qpq7rpnrp63o957687r980r.rrq656p04pn614q6n76o97883op73
r0p787rn92.i.02.s.sophosx1.net
```

```
g63uar2ejig5t1rkng3zezf2fksjrxpxyviro4ce5yz65udnjn.dagbuu
5pkocwcaxkntmxzwvkb1hg3q1j6ho7jwobeddjgqv.vgepxfdwfhu7
6on6gza2nkringxp35e6g3ftpqlp1h6uofgo.kukjy4jvybu7jhr1
hrgxe7es3lmkxdixmpb41g7wmbygjg7.gef2uoemc6pi8tz.er.s
potify.com
```

```
awyvrvccataaaegdiid5tmr7eteje2kst35frnnr3kupbf6hr.gq3dey
4qnjvgtoltoj2dq5bxnmamaaaeaiaaeg7xa4ut3ilu.license.cra
shplan.com
```

```
www.10.1.2.3.static.because.dul.is.rfc.ignorant.edu.za.
static.because.dul.is.rfc.ignorant.edu.za.research.edu
JohnsonHouse\032Officejet\032J6400\032\032The\032Johnso
n\032MacBook..ipp..tcp.johnsonhouse1.members.mac.com
```

(a) Example DNS names with more than 100 bytes in length (cf. § 5.2).

```
1751913.86c0ade0d13143ab83d7e4f60cbd204c.00000000.xello
.xobni.com
```

```
1753942.86c0ade0d13143ab83d7e4f60cbd204c.00000000.xello
.xobni.com
```

```
1756950.86c0ade0d13143ab83d7e4f60cbd204c.00000000.xello
.xobni.com
```

```
1758762.86c0ade0d13143ab83d7e4f60cbd204c.00000000.xello
.xobni.com
```

(b) Example DNS names with little variation between consecutive queries.

```
p9b-8-na-5w-2z3-djmu-7pk-qy-0-bok-re9-ym-v9h-av-njx-2es
.info
```

(c) Example DNS name reflecting malware activity (cf. § 7.2).

```
_ldap..tcp.standardname-des-ersten-standorts..sites.dc.
_msdcis.isi26.isi.fhg.de
```

(d) Example DNS name originating from client misconfiguration (cf. § 7.2).

Figure 7: Full names of examples used in the main text. We line-break each name at 54/55 characters.

reverse lookups that we excluded them from the SIE analysis, which removes about 10% of the queries.

As previously discussed in § 4, we emphasize that the role of the SIE dataset for our evaluation is simply to give us a (huge) target environment in which to validate that we can find actual tunnels. We do not envision our procedure as operationally viable for this environment; nor does such an environment strike us as making sense in terms of conforming with our threat model, which focuses on tightly controlled enterprises, rather than wide-open ISPs.

Given this perspective, to keep our own manual analysis tractable, for SIE we used a detection threshold \mathcal{I} of 10 kB rather than the 4 kB value we use for the other datasets.

We also explored the effects of other analysis changes. First, we investigated conducting our analysis on the SIE queries reduced to distinct, sorted names. This transformation removes our opportunity of assessing *query name-codebook* information vectors, but preserves our ability to estimate data conveyed through the *query name-content* vector—the only type of encoding employed by known DNS tunneling tools. Table 2 shows this version of the SIE data as SIE^{UNIQ}. The reduction in analyst load is quite significant, more than a factor of three.

Let Me Answer That For You: Exploiting Broadcast Information in Cellular Networks

Nico Golde, Kévin Redon, Jean-Pierre Seifert

Technische Universität Berlin and Deutsche Telekom Innovation Laboratories

{nico, kredon, jpseifert}@sec.t-labs.tu-berlin.de

Abstract

Mobile telecommunication has become an important part of our daily lives. Yet, industry standards such as GSM often exclude scenarios with active attackers. Devices participating in communication are seen as trusted and non-malicious. By implementing our own baseband firmware based on OsmocomBB, we violate this trust and are able to evaluate the impact of a rogue device with regard to the usage of broadcast information. Through our analysis we show two new attacks based on the paging procedure used in cellular networks. We demonstrate that for at least GSM, it is feasible to hijack the transmission of mobile terminated services such as calls, perform targeted denial of service attacks against single subscribers and as well against large geographical regions within a metropolitan area.

1 Introduction

While past research on *Global System for Mobile Communications (GSM)* mainly focused on theoretical research [17, 18], a very recent research direction challenged the fundamental GSM security assumptions with respect to the practical availability of *open* GSM equipment. The assumptions have been made on both sides of the radio part of the cellular network. One side of the radio link is the *Base Station System (BSS)* consisting of the *Base Transceiver Station (BTS)* and the *Base Station Controller (BSC)*, while the other side of the radio part is the modem or the so-called baseband of a cellular phone. Traditionally, both radio stacks have been carefully kept out of reach for any kind of malicious activities.

But a booming market for used telecommunication equipment, cheap software defined radios, leakage of some hardware specifications, and a well-trained open source community finally broke up this closed cellular world. The overall community work culminated in three open source projects: OpenBSC, OpenBTS, and Osmo-

comBB [20, 25, 45]. These open source projects constitute the long sought and yet *publicly available* counterparts of the *previously closed* radio stacks. Although all of them are still constrained to 2G network handling, recent research provides open source software to tamper with certain 3G base stations [24]. Needless to say that those projects initiated a whole new class of so far unconsidered and practical security investigations within the cellular communication research, [28, 30, 34].

Despite the recent roll-out of 4G networks, GSM remains the dominant cellular standard in many countries. Moreover, as most new LTE devices are backwards compatible to GSM, this older standard will not vanish soon at all, but rather complement 3G and LTE connectivity in areas with pure GSM coverage. Several other reasons such as worse indoor coverage and the lower number of deployed UMTS and LTE base stations contribute to this. Additionally, telecommunication providers have already begun to reuse their existing GSM infrastructure within non-voice scenarios which require a much slower data communication than modern network technologies are capable of. This is especially the case for *Machine to Machine (M2M)* or so-called *Internet of Things (IoT)* communications over GSM. Corresponding applications will soon become parts of our daily life and will make us more dependent than ever on GSM, cf. [19, 35]. Given this pervasive GSM usage, it is very important to evaluate the security offered by a standard which is more than 20 years old and is based on assumptions, many of which no longer hold true.

This paper continues the challenge of the mobile security assumption that *certain active attacks can be safely excluded* from the threat model. Towards this goal we show novel attacks against mobile terminated services. While the root cause also exists in newer standards such as UMTS or LTE, we demonstrate the impact of it in commercially deployed GSM networks. To the best of our knowledge, the limitations of currently available hard- and software would make it very difficult

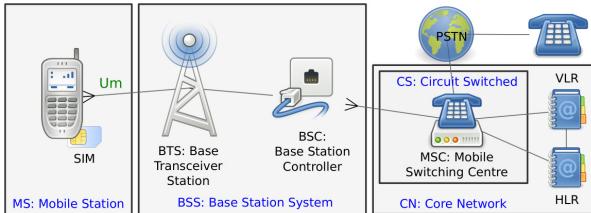


Figure 1: Simplified GSM network infrastructure.

to test these attacks in UMTS and LTE networks. Prior to publishing this research, we responsibly notified the respective standard organisations via a carrier of our research results.

In summary, we make the following main contributions:

- We present the paging response attack, a novel and practical attack against mobile terminated services.
- We show the feasibility and the implementation of a mobile phone firmware which is capable to steal a short message over-the-air and to perform denial of service attacks against mobile terminated services in GSM networks. Furthermore, we evaluated these attacks to be present in major European operator networks.
- We eventually assess the boundary conditions for a large-scale paging response attack in order to cause denial of service conditions within a large geographical area of a major city.

The remainder of the paper is structured as follows. Section 2 provides an overview of the 3GPP GSM network infrastructure, as well as details about logical channels and paging protocol procedures required to understand our attacks; Section 3 details our novel attack that exploits the paging procedure as used in GSM; Section 4 describes characteristics of location areas in a large metropolitan area and the respective requirements to perform a large-scale denial of service attack against these regions; Section 5 discusses two different countermeasures to address the attacks; Section 6 provides an overview of related research; Section 7 concludes our research.

2 Background and Overview

This section briefly describes the GSM cellular network infrastructure. We continue to explain the important types and functions of logical channels. Furthermore, we depict the protocol details required to understand the basis of our attack.

2.1 GSM Infrastructure

Despite the complexity of a complete GSM mobile network architecture [3], only a few entities are relevant to this work. In the following paragraph, we provide the necessary background on the infrastructure components of relevance to this research. Figure 1 illustrates the architecture and connections between these components:

- *BTS*: The Base Transceiver Station is a phone’s access point to the network. It relays radio traffic to and from the mobile network and provides access to the network over-the-air. A set of BTSs is controlled by a Base Station Controller (BSC) and is part of a Base Station System (BSS).
- *MS*: The Mobile Station is the mobile device interacting with the mobile operator network. It comprises hardware and software required for mobile communication (baseband processor, SIM card, and a GSM stack implementation). The MS interacts with the BTS over a radio link, also known as the U_m interface. In this paper, the mobile phone of a victim is often referred to as MS. We will also use the term MS, user, subscriber, phone, and mobile device interchangeably.
- *MSC*: The Mobile Switching Center [6] is a core network entity responsible for routing services, such as calls and short messages, through the network. It utilizes components from BSSs to establish connections to mobile devices, organizes hand-over procedures and connects the cellular network to the Public Switched Telephone Network (PSTN).
- *VLR*: The Visitor Location Register maintains location and management data for mobile subscribers roaming in a specific geographical area handled by an MSC. It acts as a local database cache for various subscriber information obtained from the central Home Location Register (HLR), e.g., the mobile identity. A subscriber can only be present in one VLR at a time. Each of the areas served has an associated unique identifier, the Location Area Code (LAC) [3, 8]. As soon as a phone leaves a certain geographical area called *Location Area (LA)*, it has to perform the Location Update procedure [4] to notify the network of this event.

2.2 GSM Logical Channels

The available GSM frequencies are shared among a number of mobile carriers. Each of the GSM frequency bands is divided into multiple carrier frequencies by means of Frequency Division Multiple Access (FDMA). A BTS

serves at least one associated carrier frequencies identified by the Absolute Radio-Frequency Channel Number (ARFCN). The ARFCN provides a dedicated pair of uplink and downlink frequencies for receiving and transmitting data over the U_m interface [10]. Because the radio frequency is shared among a number of subscribers, GSM uses Time Division Multiple Access (TDMA) as channel access method and divides physical channels provided by the ARFCN into 8 time slots. A sequence of 8 consecutive time slots is called a TDMA frame. Multiple TDMA frames form a multiframe. It consists either of 51 or 21 TDMA frames (respectively control frames or traffic frames). Multiframes are further partitioned to provide logical channels.

The two categories of logical channels in GSM are *control channels* and *traffic channels* [5]. Control channels provide means for signaling between the network and the MS. Because our attack is solely based on signaling, we focus on the details of control channels. There are three categories of control channels:

- *BCH*: Broadcast Channels provide a point-to-multipoint, unidirectional channel from the BTS to mobile stations (transmitted on the downlink frequency). Among other functionalities, they act as beacon channels and include logical channels for frequency correction (FCCH), synchronization (SCH), and information about the cell configuration and identity (BCCH) [5, 7].
- *CCCH*: Common Control Channels are used for signaling between the BTS and MS, both on the uplink and downlink. They are used by the MS to request radio resources and to access the mobile network.
- *DCCH*: Dedicated Control Channels carry signaling messages related to handover procedures or connection establishment, e.g., during call setups.

For our attack, we are mainly interested in logical channels that are part of the CCCH and DCCH categories. These categories consist of several logical channels. The logical channels of interest are as follows:

- *PCH*: The Paging Channel is used by the BTS to inform an MS about an incoming service (via paging request messages on the downlink channel). The PCH, which is part of the CCCH, will be monitored by any MS in idle mode unless it is currently using a dedicated channel.
- *RACH*: The Random Access Channel provides a shared uplink channel utilized by the MS to request a dedicated channel from the BTS. Placing a phone

call or receiving an incoming service always requires a phone to setup a dedicated signaling channel beforehand.

- *AGCH*: The Access Grant Channel provides a downlink channel used by the BTS to transmit assignment messages that notify mobile stations of assigned channel details. A successful channel request on the RACH will result in an Immediate Assignment message on the AGCH. These assignment messages contain the required configuration parameters that enable the MS to tune to the requested channel.
- *SDCCH*: The Standalone Dedicated Control Channel is used on both uplink and downlink. It is employed for call setup and signaling between BTS and MS. Furthermore, it can be utilized to transmit short messages to the MS.

It is important to note that both the BCH and CCCH channel types are point-to-multipoint channels. This implies that information on the logical downlink channels is broadcasted to all subscribers served by a specific BTS. Throughout this work we will see how this can be abused to model new attacks.

2.3 Mobile Terminated Service Procedures

The GSM specifications differ between traffic originating or terminating at a mobile phone. This is referred to as Mobile Originated (MO) and Mobile Terminated (MT) traffic. As outlined previously, we aim to attack MT services, such as phone calls or SMS. Thus, in the following we concentrate on the underlying protocol procedures associated with MT services [4].

In order to deliver a service to a phone, the MSC needs to determine the location of the respective subscriber. This has to be done for two reasons. First, mobile phones will be idle most of the time to save battery power and so will not be in constant contact with the network. Thus, the operator does not always know the specific BTS that provides the best reception level to the MS. Therefore, it must broadcast this signal of an incoming service through at least the entire location area. Second, broadcasting this information through the whole operator network would impose a huge performance overhead and possibly overload the paging channel [42].

In a first step, the core network determines the responsible MSC/VLR for the target subscriber with the help of the HLR. Next, the MSC obtains the location information for the destination subscriber from the VLR and sends a *paging message* to all BSCs in the subscriber's location area. This message includes a list of cell identifiers/base stations serving the specific

location area [13]. The message also contains the mobile identity of the subscriber, which is usually either a *International Mobile Subscriber Identity* (IMSI) or a *Temporary Mobile Subscriber Identity* (TMSI). We illustrate the remaining protocol logic using a successful MT phone call as depicted in Figure 2.

1. The BSC sends a *paging command* message which includes the subscriber identity to all base stations within the location area. All base stations re-encapsulate the mobile identity and transmit it as part of a *paging request* message on the downlink PCH.
2. When receiving a paging request on the PCH, each MS compares the Mobile Identity (MI) included in the request with its own. The result determines whether the message is addressed to itself or a different subscriber.
3. In case of an identity match, the MS needs to acquire access to Radio Resources (RR) in order to receive the MT service. To do so, it sends a *channel request* including a random reference number on the uplink RACH.
4. Upon receipt of the channel request, the network allocates radio resources and a dedicated channel. Next, it acknowledges the request and sends details of the allocated channel to the MS in an *immediate assignment* message on the AGCH downlink. To allow the MS to identify its assignment, the message contains the random reference of the requester.
5. The AGCH is a shared downlink channel. Therefore, an MS receiving an assignment message compares the included reference with the one sent in the request. If the reference matches, the MS tunes to the dedicated signaling channel included in the assignment.
6. After this step succeeded, the Mobile Station establishes a signaling link, usually over the SDCCH, by sending a GSM Layer 2 *SABM* frame containing a Layer 3 *paging response* message.
7. Following this, the MS and BTS undergo an authentication, ciphering and service setup procedure. Details of this procedure are not relevant for our attack. We skip these details here.

The GSM standard specifies [4] three types of paging requests – type 1, 2, and 3. The type stipulates the number of subscribers that can be addressed with the paging request. Type 1 can page one or two subscribers,

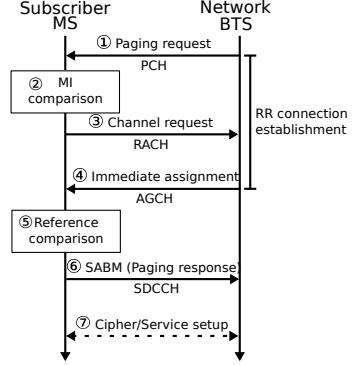


Figure 2: Mobile Terminated (MT) paging procedure.

type 2 two or three subscribers, and type 3 paging requests are directed towards four subscribers at once. A recent study [30] suggests that in real operator networks the vast majority of paging requests is of type 1. During our experiments, we verified that 98% of all paging requests that we observed are type 1 requests. Therefore, we ignore type 2 and type 3 paging requests in our study.

3 Attack Description

In this section, we will provide the theoretical background of our attack, introduce our experimental setup and elaborate on the feasibility of such an attack.

3.1 The Two Threat Models

Denial of Service Attacks. The first threat comprises an active attacker, interested in significantly *disturbing* mobile terminated services within a specific geographical area, e.g., a district or a part of a city. In certain situations it is desirable to ensure that a person or a device is not reachable via mobile telephony. For example a third-party may want to prevent a specific call from reaching the victim. The effect would be similar to the ability of selectively jamming incoming services for a set of subscribers. This includes individuals and groups of individuals. Such an attack would also have considerable business ramifications. While it would not compromise the general operation of the carrier, it would affect their revenue. The inability to receive a phone call will not only leave angry customers, it further impacts the generated billing as subscribers are charged when a call is connected. If any subscriber is able to place phone calls, but nobody is able to receive services, no profit is created. An exception here are short messages, as SMS operates in store-and-forward fashion and does not create billing on delivery of a message, but on its submission.

Mobile Terminated Impersonation. The second threat considers an attacker who aims to *hijack* a mobile terminated service. As a result, the service would be delivered to the attacker instead of the victim. This turns a passive adversary, who is able to observe air traffic, into an active attacker who can accept the mobile terminated service and impersonate the victim. For example an attacker could be interested in hijacking the delivery of an SMS message. Consequently, it is possible to read its content and at the same time prevent its submission to the victim. In practice this could, for example, allow an attacker to steal a mobile TAN (mTAN), which is often used as two-factor authentication for online banking, or any other valuable secret from the message. We also consider an attacker who wants to impersonate a victim that is being called. By hijacking the MT call setup, it is almost impossible for the calling person to verify the callee's identity by means other than the voice.

3.2 Paging Response Attack Description

Our attack is inspired by two specific properties of GSM networks and its protocols.

Network State: GSM networks involve complex state machines [4] and face high amounts of traffic while operating on tight radio resource constraints. Consequently, it is desirable to keep states as short as possible.

Broadcast Information: the paging procedure is initiated on a broadcast medium, namely the PCH portion of the CCCH, and more importantly is performed before any authentication or cipher setup takes place. This implies that any subscriber, including an adversary phone, is able to observe paging requests for other subscribers, plus the inherent inability of the network to distinguish between a fake paging response and a genuine one.

As a net result, it is possible to exploit these aspects to send paging response messages on behalf of a victim being paged. The network stack can under no circumstances determine which of the replies is the legitimate paging response by the intended subscriber.

Denial of Service. The GSM documents do not specify the network behavior in such a situation. Therefore, the behavior of such a race condition is implementation dependent and may be exploitable. However, the state machine nature of GSM protocols suggest that if an attacker is able to answer a paging request faster than the intended subscriber, it will no longer be in a state in which it expects a paging response and thus will ignore the message of a victim. Consequently, the victim will receive a channel release message from the network. Next, the service setup will not succeed if the attacker does not provide the correct cryptographic keys required to complete authentication and cipher setup. Accordingly, the service setup

cannot proceed and for example, a call will be dropped. The result is a novel and powerful denial of service attack against MT services that 1. does not rely on frequency jamming; 2. does not rely on resource exhaustion; and 3. is very hard to detect.

We verified that it is indeed possible to win the race for the fastest paging response time, as we will demonstrate. We were able to carry out such an attack in all major German operator networks including O2, Vodafone, T-Mobile, and E-Plus.

MT Session Hijacking. Exploiting the paging procedure does not only allow to disturb communication. It is important to note that in certain network configurations, this attack could be abused beyond performing denial of service attacks. Not all countries properly authenticate each service and use encryption. For example, only under 20% of the networks analyzed by the gsmmap project [41] authenticate mobile terminated phone calls 100% of the time. 50% of the tested networks only authenticate 10% of the services [28].

In such a network, an adversary can effectively take over any MT service that is not authenticated and impersonate a victim. We assume a network without encryption and insufficient authentication as above. If the attacker is able to successfully exploit the race condition on the air interface, it is possible to directly hijack an MT service by following the protocol specifications. The paging response attack proceeds as in the DoS scenario. However, in this case, by winning the race, an attacker can accept, e.g., a victim's phone call or short message.

The victim of such an attack is thus faced with two consequences. For a mobile terminated call, it is not safe to assume that the called party is indeed the desired person. For short messages this implies that a message may not reach the victim, but additionally also that its contents cannot be considered secret.

Even if the network is configured to use encryption, an attacker is merely required to perform an additional step. In an encrypted network without proper authentication, the paging procedure is followed by the cipher setup. During this process to create an encrypted channel, the network sends a *cipher mode command* message to notify the MS of the encryption algorithm to be used. The *cipher mode complete* response from the MS indicates a completion of the cipher setup. In a network that uses encryption, this response has to be encrypted using the session key K_c as input to the A5 encryption algorithm. This session key is derived from a secret key K_i that is stored on the SIM card issued by the operator and a random challenge *RAND* sent from the network to the MS. Due to the lack of perpetual authentication, an attacker can fully impersonate the victim after cracking the session key K_c and sending the *cipher mode complete* mes-

sage. The cracked session key then allows to decrypt the subsequent communication that follows the cipher setup.

In practice, essentially both commonly used GSM cipher algorithms, A5/2 and A5/1, have been broken and demonstrated to be cryptographically weak [17, 18, 23, 39]. The session key can be acquired before hijacking the service by sniffing air traffic and using the kraken tool [40]. Also, some networks are configured to still use A5/0 [26], which does not provide any encryption. This further simplifies such an attack in those commercially deployed networks. Furthermore, for the subsequent paging response attack, an attacker does not even require physical proximity to a victim, because, as explained earlier, the carrier network is paging throughout an entire location area. In order to exploit this, an attacker requires a mobile device that enables him to observe traffic on the air interface and send arbitrary messages to the network. Additionally, a practical attack requires the fake response to arrive prior to the victim’s message. Therefore, the attack is significantly challenging in terms of timing.

We successfully implemented both, the MT service hijacking and the denial of service attack. For the sake of simplicity, we obtained the session key through the SIM browser in the engineering mode of a Blackberry phone. Nevertheless, as outlined before this step, it can be trivially obtained by a 3rd party by using a tool like kraken [40]. Cracking of K_c is merely a step that has to be performed prior to our attack, but is not part of the problem itself, which is the race condition. Given a known K_c , our code to take over an MT session, can hijack the transmission of a short message delivery in a real network.

It is important to note that the main reason for evaluating the paging race condition in GSM was the availability of freely modifiable hardware and software. However, modern telecommunication standards such as UMTS or LTE are making use of exactly the same paging procedure principles [11, 14, 15]. Insufficient cryptography and authentication further escalate the problem, but the root cause does not only pertain to GSM.

We will continue to examine the requirements, boundary conditions, and feasibility of mounting such an attack in practice.

3.3 Experimental Setup

Launching such an attack requires hardware and software to interact with GSM base stations. More precisely, the attack relies on a device which allows us to modify its baseband (BB) implementation in order to control its radio communication. Traditionally this has been very difficult due to the closed nature of the GSM industry

(phone manufacturers, baseband vendors, infrastructure equipment suppliers). For many years there existed no freely modifiable radio communication hardware with GSM stack implementations. While the GSM specifications are publicly available (very comprehensive though, over 1000 PDF documents), there are very few manufacturers of GSM equipment who have released any public documentation.

However, this situation has changed in the last years with the availability of inexpensive hardware such as the Universal Software Radio Peripheral (USRP) [22] and various software implementations around the Osmocom [45] project. Additionally, in 2004 the source code of the Vitelcom TSM30 mobile phone was uploaded to a Sourceforge project [37] which allowed a broader audience to study a GSM phone stack for the first time.

Hardware Selection. There are basically three possible choices when it comes to the hardware selection of our desired radio device: *USRP*, *Vitelcom TSM30*, and certain *TI Calypso chipset based phones*. All of these devices can be utilized as GSM radio transceivers with software modifications. Yet some of these come with intrinsic disadvantages. First, for the USRP there is currently no GSM baseband implementation that allows the device to be used as a handset. While we could have implemented this, it would have been a very demanding task. Second, even though available, the TSM30 source code is a full-featured baseband implementation, which is too complex for our needs. Moreover, the availability of TSM30 devices is sparse and they are not easy to obtain.

Instead we used Motorola C123 and Motorola C118 phones, which are based on the TI Calypso chipset. These phones are inexpensive (around 20 Euros), easy to obtain in quantity, and more importantly can be used in combination with the Free Software baseband implementation OsmocomBB [47]. This enables us to receive over-the-air traffic and send arbitrary GSM frames.

Implementation. OsmocomBB implements a simplified version of the GSM stack. The GSM physical layer (L1) firmware runs on the phone, while the data-link layer (L2) and Layer 3 (L3) run on a computer as an application (layer23). L1 and layer23 communicate with each other via a UART serial connection. Layer 2 implements a modified version of the Link Access Protocol for the D channel (LAPD) used in ISDN, called Link Access Protocol on the Dm channel (LAPDm). Layer 3 comprises three sublayers: Radio Resource management, Mobility Management, and Connection Management. As our attack is based on paging, which is part of Layer 3, we required a modified version of the layer23 application.



Figure 3: Experimental setup: Motorola C1XX phones with custom firmware, GPS receivers, and a laptop for serial communication.

In practice our attack is particularly time critical, because we have to win a race condition on the air interface. It became evident that a layer23 implementation that runs on a computer is far too slow to win the race given the bottlenecks such as queueing between multiple layers, scheduling, and the use of UART serial communication. Consequently, we reimplemented a minimal version of LAPDm and Layer 3 directly in the L1 firmware to allow it to run solely on the phone. Specifically this includes the paging protocol, which is part of the radio resource sublayer.

Figure 3 shows our experimental setup consisting of a notebook and several OsmocomBB phones. The serial cables are required in order to flash the firmware. Using this implementation we can camp on specific ARFCNs, observe paging requests within a location area, and send arbitrary GSM layer2/layer3 messages in a timely manner. Additionally, we used OpenBTS [20] in combination with a USRP as a BTS to test our setup and perform various measurements as later described in Section 3.5.

3.4 Targeted Attacks

Attacking individual persons requires our OsmocomBB phone to observe air traffic and respond to specific paging requests. In particular paging requests that contain the victim's mobile identity. For privacy reasons, most network operators use TMSIs as mobile identities rather than the static IMSI. The TMSI is only valid within a location area and is subject to frequent changes [9]. Therefore, we need to determine the presence and the TMSI of a victim in a given location area.

For this we implemented the method proposed by Kune et al. to reveal the mapping between TMSI and subscriber [30]. We modified OsmocomBB's layer23

mobile application and introduced functionality that issues n (where n is 10-20) phone calls in a row. Next, the application terminates the connection before the target phone is ringing, but late enough so that the network generates a paging request. The victim phone does not ring during this early stage of the protocol flow, because it does not know yet what type of service is incoming. In our tests we empirically determined that, e.g., a time of 3.7 seconds after the *CC-Establishment confirmed* state has the desired effect in the O2 network. The exact timing may differ slightly, depending on the network that is used to initiate the call and the network in which the victim resides.

At the same time, a second phone is monitoring the PCH of any BTS within the target location area for paging requests. All TMSIs contained in the observed paging requests are logged together with a precise timestamp of the event. It makes sense to choose the ARFCN with the best signal reception to minimize errors and possible delays. By first limiting the resulting log to time ranges in which our calls were initiated, we can extract a number of candidate TMSIs. Further filtering the result set for TMSIs occurring in repeating patterns that reflect our call pattern yields to a very small set of candidate TMSIs or even single TMSIs. This process can be repeated to narrow down the set of candidate TMSIs to a manageable number. If the network uses IMSIs for identification, then an attacker could use the same process to determine the subscriber's identity. Alternatively, an attacker could use a Home Location Register query service to obtain the IMSI directly [1].

By default, the monitoring phone does not react to any paging request. After obtaining the victim TMSI, we transfer the TMSI via HDLC over the serial connection to the monitoring phone. This also changes the phone's role from a solely passive listener to an attacker. It starts to compare TMSIs contained in paging request with the supplied victim TMSI. On every match, the attacking phone promptly initiates the previously introduced paging protocol procedure to respond first. As a result, the paging response by the victim will be ignored and the call will be dropped unless we fully accept the service. At this point, it is not possible to reach the victim anymore. To block MT services over a longer period of time, the subscriber identification procedure needs to be reissued due to TMSI reallocations over time [4].

3.5 Feasibility

The success of such an exploit depends essentially on the response time of the attacker and victim devices. To achieve maximum impact, an attacker phone needs to respond faster than the “average” customer device. The response time of the phone depends on a number of fac-

Table 1: List of tested phones, baseband chipset, and baseband vendor.

Phone model	BB chipset	BB vendor
Blackberry Curve 9300	Marvell PXA930	Marvell
iPhone 4s	MDM6610	Qualcomm
Samsung Galaxy S2	XMM 6260	Infineon
Nokia N900	TI Rapuyama	Nokia
Nokia 3310	TI MAD2WDI	Nokia
Motorola C123	TI Calypso	OsmocomBB ¹
SciPhone Dream G2	MT6235	Mediatek
Sony Ericsson W800i	DB2010	Ericsson
Sony Xperia U	NovaThor U8500	ST-Ericsson

¹ Layer1 paging attack code and modified layer23 application.

tors that are difficult to measure. This includes signal quality, weather, network saturation, application processor operating system, GSM time slots, and others. Yet, most of these parameters only have very little impact on the overall response time.

As the baseband chipset and its GSM stack implementation handles all radio communication, including the upper layer GSM logic, we suspect it to be a key contributor to a fast response time. We validate this claim by measuring the timing of various phones with different baseband vendors. Referring to a market report [2], Qualcomm and Intel alone account for 60% of the baseband revenue in 2011. Yet, relevant baseband chips and stacks that are currently available in mobile phones on the market are Qualcomm, Intel (formerly Infineon), Texas Instruments, ST-Ericsson, Renesas (formerly Nokia), Marvell, and Mediatek. We tested timing behavior for different phones for each of these vendors. Additionally, we also tested the response time for the OsmocomBB layer23 application to back up our claim that this implementation is too slow to perform our attack. Table 1 lists the tested phone models, chipset names, and the corresponding baseband vendor.

Timing Measurements. It is not feasible to modify the tested devices itself for measurements, as we only have access to the operating system on the application processor, and not the baseband. Furthermore, the phone could only guess when its response hits the serving network. Thus, in order to estimate the paging response time, we operate our own test GSM BTS based on a USRP and OpenBTS [20]. OpenBTS implements a simplified GSM network stack running on commodity hardware while using the USRP device as a transceiver. We patched OpenBTS to obtain timing information for the different steps during the paging procedure. Specifically, we are interested in the time a phone needs to acquire a radio channel and to send the paging response. This includes two parts of the paging procedure, the time

between the initial paging request and the channel request, and the time between the initial paging request and the reception of the paging response. We log both of these timestamps for the relevant baseband vendors in nanoseconds using *clock_gettime(2)*. Additionally, we measure the same for an attack phone running our own lightweight, OsmocomBB-based baseband implementation. To trigger paging activity, we consecutively send 250 short messages, one after each channel teardown, to our test devices.

While we could have also used software like OpenBSC [25] in combination with a nanoBTS [27], we decided to utilize OpenBTS to be in full control over the transmission and reception. The nanoBTS is controlled over Ethernet, runs its own operating system, including scheduling algorithms, and cannot be modified. Thus, we used OpenBTS to minimize the deviation that may occur due to the nature of this BTS device.

Timing Observations. Figure 4 summarizes the results of our time measurements for each baseband vendor. It shows the elapsed time between the first paging request message sent to the phone, the arrival of the channel request message, and the occurrence of the paging response. Interestingly, the generation of the phone had little influence on the response timing. In our tests, a Nokia 3310, which is almost 10 years older than the tested Nokia N900, shows almost the same timing behavior. We do not have a definitive answer to explain this observation. However, a plausible explanation can be found in the age of GSM. GSM was developed in the 1980s and most of the mobile telephony stacks for GSM are of this

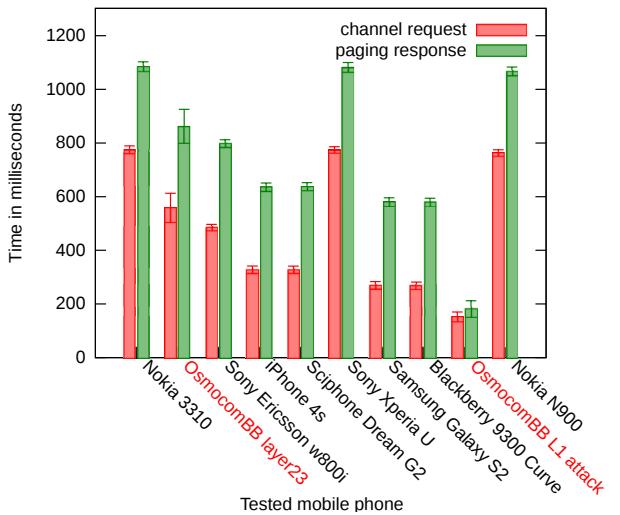


Figure 4: Time difference between initial paging request and subsequent channel request or paging response for different baseband vendors. Confidence interval: 95%.

era. As most baseband vendors nowadays concentrate their efforts on exploring the technical challenges of 3G and 4G telephony standards, we believe that GSM stacks have not been modified for a long time. We do not expect significant modifications of baseband stacks by the respective vendors nowadays. Thus, we assume that timing behavior across different phone platforms using the same baseband will show similar patterns.

The most important observation from Figure 4 is that on average, with a confidence interval of 95%, our minimal OsmocomBB-based implementation is the fastest in transmitting the channel request and paging response. For our implementation, there is roughly a 180 milliseconds delay between the paging request and the arrival of the paging response. Thus, on average our attack implementation is able to transmit the final paging response prior to all other major basebands and can be conducted within the duration of a single multiframe (235.4 ms). This includes the OsmocomBB layer23 mobile application, which is significantly slower than our self-contained layer1 attack software and shows similar timing performance as conventional phones.

Therefore, with a very high likelihood, our software is able to win the race. It is also noteworthy that our lightweight stack can transmit the paging response almost immediately after the channel request (and reception of the Immediate Assignment). The test devices show a gap of at least 200ms before the transmission of the paging response. We expect that this is related to internal scheduling algorithms and queuing mechanisms between different layers of the baseband implementation.

4 Attacking Location Areas

Besides attacking individual subscribers, we show that it is also possible to leverage this attack to disrupt network service in large geographical regions. As explained in Section 2, the serving network does not always have the knowledge of the exact location a subscriber resides in. As a consequence, it also does not know which BTS is currently within a good reception of the mobile device. The phone announces a change of the location area by performing the *Location Update* [4] procedure. By monitoring *System Information* [4] messages on the Broadcast Control Channel (BCCH), a phone can keep track of location areas served by the BTSs within reception. The aforementioned lack of knowledge is compensated by the network by distributing paging requests throughout all base stations in the location area. This implies that an adversary is able to observe and respond to paging requests not only transmitted by a single a BTS, but within a larger geographical region formed by the location area.

We already showed in Section 3.5 that we win the race for the paging response with high probability. Given that

an attacker is able to answer all paging requests that can be observed on the PCH, it is possible to perform a denial of service attack against all MT services within the location area. Depending on its size, the impact of this would be massive, e.g., breaking MT calls in areas as large as city districts or even bigger regions. However, in practice there are a few obstacles to consider.

Depending on the paging activity, it is unlikely that service in an entire geographical can be disrupted by a single attacker phone. In order to send the paging response, the MS has to tune to a dedicated channel. As a result, it would not be able to observe paging requests while being in dedicated mode. After sending the response, the attacker MS has to resynchronize with the BTS to observe CCCH/PCH traffic again. By logging timestamps for the various protocol steps, we measured the time for this procedure on the OsmocomBB side. On average we need 745 milliseconds to resynchronize in order to receive further paging requests after we sent the response. Furthermore, as shown in Figure 4, we need on average 180 milliseconds to transmit the paging response. This means that in ideal conditions, with a single phone, we are able to handle up to

$$\frac{60\text{s}}{745\text{ms} + 180\text{ms}} = 64.8 \text{ paging requests per minute.}$$

Depending on the network activity, this may or may not be enough to answer all paging requests. Additionally, we need to examine the different paging activities that can be seen in real operator networks. If the paging activity is very large, then the attacker may need to use multiple phones to perform the attack.

Finally, to get an understanding of the impact of such an attack, we need to determine the size of the geographical region covered by a location area.

4.1 Location Area Paging Activity

An attack against an entire location area, e.g., in a metropolitan area, requires an adversary to respond to all paging requests in that area. Consequently, the efficiency of a large-scale attack depends on the operator specific paging activities and the allocated resources on the attacker side.

For the purpose of estimating paging activity, we modified our OsmocomBB stack to log all TMSIs in combination with a time stamp of its appearance. Because the paging requests are broadcasted throughout a location area, camping on one operator BTS for that area is sufficient to observe all paging activity for that area on the CCCH/PCH. We recorded the TMSIs in paging requests for all major German operators in a metropolitan area over a time period of 24 hours. The logs were created at exactly the same location, at the same date and

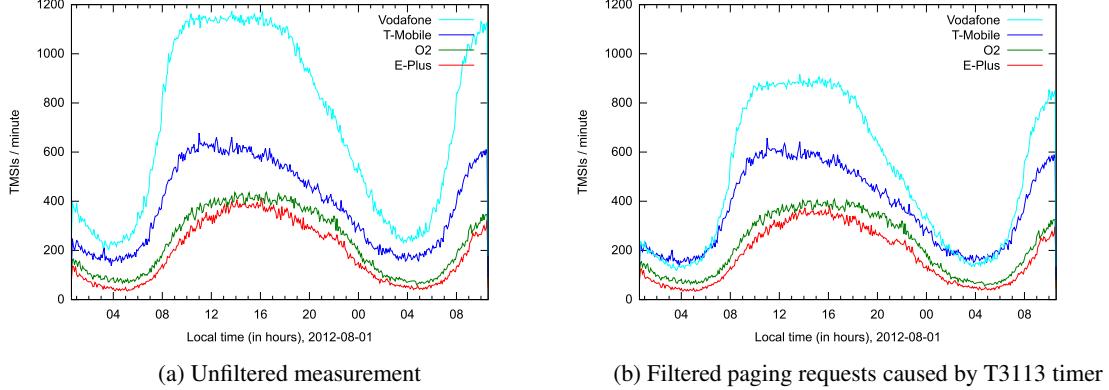


Figure 5: Number of TMSIs per minute contained in paging requests of four major German operators over 24 hours.

time. We observed that in some cases the network is not paging with the TMSI but with the IMSI. E.g., if the subscriber is marked as attached to the network but cannot be reached using the TMSI, the MSC starts paging using the IMSI. In this case, depending on the operator network configuration, paging may also be performed outside of the location area. However, this type of paging request is the minority and thus ignored in our measurements. Furthermore, assuming that a subscriber is present in the monitored location area, the network very likely already paged using the TMSI in this area. Obviously, it is simple to implement the attack in the case that network pages using IMSIs instead of TMSIs. In fact our code can also handle IMSIs.

Figure 5a summarizes the paging observations. The first observation to be made is that paging activity heavily varies throughout the time of the day. The observed pattern is not random, but rather reflects human activity during typical days. It is also interesting to note that the amount of paging requests heavily differs among the various tested operators. While for example E-Plus at peak times has a rate of roughly 415 TMSIs contained in paging requests per minute, Vodafone has almost 1200 in the same time period.

Such differences can be caused for example by the number of active subscribers in the network, or the size of the respective location area. During this measurement, we noticed several reoccurring TMSI patterns. Vodafone is actually always paging each TMSI at least two times. A second paging request is always issued two seconds after the initial paging request. This explains the massive amount of paging requests and we suspect this to be an attempt to improve the overall subscriber availability. Also, our logged data shows that some of these TMSIs are paged at regular intervals. We believe that these requests may partially be directed at M2M devices, e.g., for remote monitoring.

Figure 5b shows a filtered version of Figure 5a. Specifically, we filtered appearances of TMSIs contained in paging requests that we do not need to respond to. 3GPP TS 04.08 [4] specifies a timer, T3113, that is set on transmission of a paging request. If no paging response was received prior to the expiry of this timer, the network reissues the paging request by paging the mobile subscriber again. However, assuming that we are able to observe and respond to all paging requests, this retransmission would not occur during an attack. Therefore, these can be filtered from the result. By analyzing the logged TMSIs and the respective timestamps, we recorded the reappearance of each TMSI that was originally transmitted as part of a paging request. The vast majority of reappearances in time reach a common maximum which we assume is the timer value. A prevalent value seems to be five seconds. It is also reasonable that this is caused by a triggered timer. A normal call setup takes longer than five seconds [30] and short messages are queued at the SMS service center and likely transmitted over the same channel following one paging request.

As a result, the overall activity of relevance in practice is lower than the general amount of TMSIs contained in observed paging requests. The Vodafone measurements can be reduced by almost 22% during peak times and 33% during low activity times. However, due to the limited memory resources of the attacking phones, we cannot take this into account during an active attack.

4.2 A Randomized Attack Strategy using TMSIs

The measured data in Section 4.1 suggests that even in location areas with low paging activity an attacker needs more than a single phone to respond to all paging requests. Thus, paging requests need to be distributed across multiple attacking phones. While serial commu-

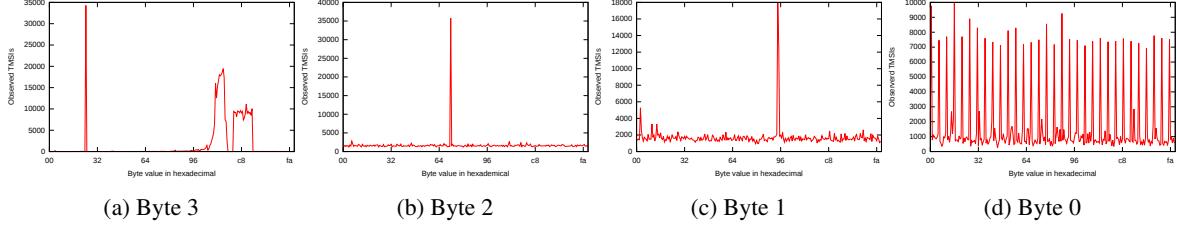


Figure 6: Statistical distribution of each TMSI byte contained in paging requests for O2. Based on 437734 TMSIs.

nication could be used to coordinate these efforts, it also poses a significant slowdown. Consequently, using serial communication would lower the chance to win the race. We therefore decided to not make use of any actual communication between attacking devices, but to use a probabilistic approach instead.

For this, we analyzed the TMSI values to determine the statistical distribution of each individual TMSI byte as contained in respective paging requests. Namely, to prevent the collection of mobile subscriber identities and thus enable tracking, mobile phones are in most cases identified by their TMSI instead of their IMSI. To provide strong anonymity, a network should therefore sufficiently randomize those short term identities to provide unlinkability. A statistically uniform distribution would ease randomly distributing the paging load across multiple phones. However, an analysis of collected TMSIs made it clear that not all bits of the TMSI are sufficiently random or at least uniformly distributed. This may be, because some parts of the TMSI can be related to, e.g., the time of its allocation [8]. We also observed that certain bytes of the TMSI appear more frequently in specific ARFCNs. Thus, we further analyzed the distribution of each individual of the four TMSI bytes, for all tested operators. We use O2 as an example operator here even though nearly identical patterns can be seen for other carriers.

Figure 6 shows for each possible byte value how of-

ten a specific value was found in TMSIs contained in paging requests that we logged. As visible, all byte values are not uniformly distributed. However, 6a, 6b, and 6c show a significantly different pattern from 6d. Not every possible value of the least significant byte (LSB) of the TMSI is encountered with equal frequency on the air interface. For example the value 0xff is not used at all. Some seem to be more likely than others. Nonetheless, 6d shows that value ranges are close to a uniform distribution. This becomes more plausible in Figure 7, which compares the cumulative distribution function for observed values of the LSB and the uniform distribution. We make use of this characteristic to delegate specific attack phones to dedicated TMSI LSB byte ranges. This way, we can distribute the immense amount of paging between several phones by simply using randomization and thus avoid coordination at all. Outliers for certain value ranges could be compensated by adding more phones to the specific range. To prevent recompilation of our OsmocomBB based firmware for distinct value range, we introduced a mechanism to configure the range at runtime. This mechanism is similar to the TMSI setting described in Section 3.4 and is based on a HDLC message over serial.

A similar distribution could be achieved by hashing TMSI values and assigning individual phones to specific hash prefixes. However for simplicity and to reduce the response time as much as possible we decided not to do this.

4.3 Mapping Location Areas

When performing a large-scale attack against a geographic region, we have to determine the size covered by the location area. Specifically, this knowledge enables an adversary to precisely plan the affected zone of such an attack. An attacker carefully selects the target location areas for specific regions and operators.

Location areas are not organized to cover an equally large area. As pointed out in Section 4.1, this impacts the paging activity that can be observed in a specific location area. Their size differs among operators and specifics of the covered environment. In fact, because of

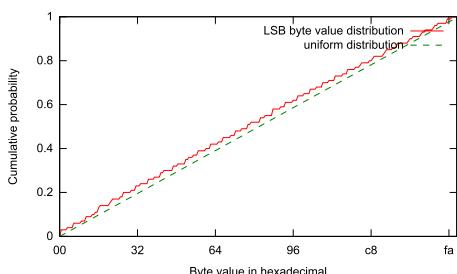


Figure 7: Cumulative distribution function for Byte 0 (LSB) of TMSIs contained in paging requests observed for O2.

its impact on mobility management, location area planning is an important aspect for mobile network operators. Its size manifests a trade-off between subscriber-induced and network-induced performance degradation. Small location areas can cause a significant signaling overhead in the core network due to frequent location updates. It has already been demonstrated, that this can lead to denial of service like conditions [36]. A large location area causes additional load due to the paging overhead.

The Location Area Code (LAC), which is part of the Location Area Identifier (LAI), is broadcasted by each BTS in regular intervals on the BCCH via a *System Information Type 3* message. To map location areas, we use a slightly modified version of the `cell_log` application from the OsmocomBB tool-chain. `cell_log` scans all ARFCNs in the assigned GSM frequency spectrum for a carrier signal. It then attempts to sync to these frequencies and logs decoded system information messages as broadcasted on the BCCH. In combination with off-the-shelf GPS receivers, we determine the geographic location of the observed LAC.

By slowly driving through the city in a car, we collected a number of waypoints and the respective GSM cells in sight. To minimize the loss due to driving speed, the scan process was performed simultaneously on eight OsmocomBB devices. In order to estimate the surface covered by a location area, we calculated the convex hull of points within the same LAC. The size of location areas in a metropolitan area such as Berlin varies from 100km^2 to 500km^2 . From our study, the average location area in Berlin covers around 200km^2 . Data from OpenCellID and Crowdflow [29, 31] indicate that outside of the city center location areas exist that cover over 1000km^2 . Figure 8 shows location areas that we mapped for one of the four major operators in Berlin.

Most location areas partially overlap with geographic regions that are part of a different area. These results provide a rough insight on dimensions of location areas in a metropolitan area. It also shows that a large-scale denial of service attack based on the paging procedure has a significant impact to a large number of subscribers.

4.4 Amplification of the Paging Response Attack

The attack procedure as introduced in Section 3 does prevent MT services from being delivered to a subscriber. However, it does not provide a persistent way to cause denial of service conditions. Access to mobile services is denied as long as an adversary is running the attack. Accordingly, calls reissued by subscribers to reach a person, have to be attacked again, which may further raise the paging load. To prevent this, we make use of another attack that has been publicized before. Munaut discov-

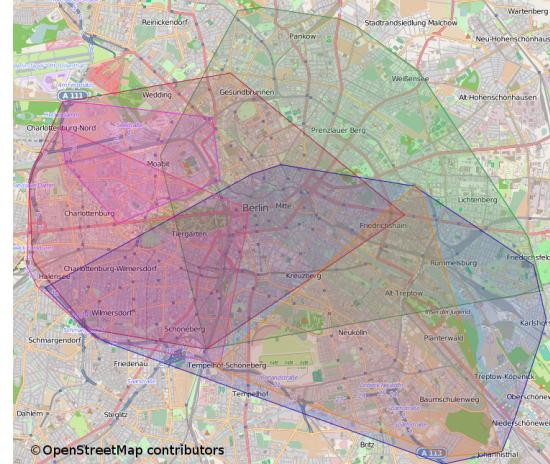


Figure 8: Location Areas of Vodafone Germany in Berlin.

ered that the *IMSI DETACH* message is not authenticated in GSM and 3G networks [34]. As a result, an attacker can easily craft detach messages on behalf of a victim. This message acts as an indication to the network that a subscriber is no longer marked as available for the carrier. As a result, the network marks the mobile station as detached and will no longer page the subscriber until it reassociates with the network. Consequently, this stops the network from delivering MT services. During normal operation, this message is generated by the phone and sent to the network, e.g., when it is being switched off.

The mobile identity contained in the detach indication message is not limited to IMSIs, but can also contain TMSIs. By combining the paging response attack with the IMSI detach attack, it is therefore possible to amplify its effect. After each paging response, our OsmocomBB implementation reuses the collected mobile identity to send a detach message. Accordingly, our attack ensures that an initial call to a subscriber will be terminated and that reissued services such as calls will not cause paging activity again. Thus, by doing so, we effectively reduce the paging load over time.

4.5 Large-scale Attack Feasibility

We continue to evaluate the feasibility of a large-scale attack using multiple phones against commercially deployed networks. The transmission of a large number of RACH bursts and SDCCH channel allocations may be limited due to radio resource bottlenecks. We therefore verify, whether or not a single cell provides enough resources, or an attack needs to be conducted in a distributed fashion.

Prerequisites. In the following, we denote the TMSI paging request activity as r_{request} and the number of re-

quired phones to handle this respectively as n_{phones} . As discussed in Section 4.2, the TMSI LSB value range is used to equally distribute the paging load across multiple attacking phones. Therefore, assigned phones need to wait for a range match. On average this requires $t_{\text{matching}} = r_{\text{request}}/n_{\text{phones}}$ seconds. On a match, the phone sends a channel request on the RACH and a paging response on an SDCCH in t_{response} seconds. It finally synchronizes back to the CCCH in t_{sync} seconds to be prepared for the next run. The required time for an attack is therefore $t_{\text{attack}} = t_{\text{matching}} + t_{\text{response}} + t_{\text{sync}}$ seconds. Thus, for a successful attack, the minimum number of phones an adversary requires is $n_{\text{phones}} \geq r_{\text{request}} \cdot t_{\text{attack}}$.

RACH Resource Constraints. The available resources provided by a single cell depend on its configuration. The GSM specifications defines a number of valid channel configurations [7]. Thus, an adversary is limited by the number of available RACH slots and the number of SDCCHs that a cell provides. In practice cells in metropolitan areas use the *BCCH+CCCH* or *FCCH+SCH+CCCH+BCCH* channel configurations on the first time slot. These are not combined with DCCHs and therefore allow all 51 bursts on the uplink of a 235.4 ms 51-multiframe to be used to transmit channel requests on the RACH. Because the RACH is a shared medium, collisions with requests of other subscribers may occur. According to Traynor et al. [36], the maximum resulting throughput is 37%. As a result, an attacker can transmit up to $r_{\text{RACH}} = 51/0.2354 \cdot 0.37 \approx 80$ channel requests per second in a single cell. Consequently, given that $n_{\text{phones}} \leq n_{\text{RACH}} = r_{\text{RACH}} \cdot t_{\text{attack}}$ is true, a single cell can fulfill the channel request requirements.

SDCCH Resource Constraints. Following the channel allocation, the adversary phone uses an SDCCH to send the paging response. Analogical to the RACH, SDCCHs in medium- or large-sized cells in a metropolitan area are provided on a separate time slot. A typical *SDCCH/8+SACCH/8* channel configuration comprises of 8 SDCCHs per 51-multiframe, in theory offering: $r_{\text{SDCCH}} = 8/0.2354 = 34$ SDCCH/second. Clearly this may make the signaling channel the major bottleneck for this attack. Accordingly, the occupation time of these channels needs to be taken into account. For example according to Traynor et al., a rough estimation of the occupation time of the channel for an Insert Call Forwarding operation is 2.7 seconds [36]. Compared to this, our attack occupies the channel for a very short duration, as shown in Section 3.5.

Similar to the RACH requirements, the maximum number of attacking phones per cell is therefore bounded by $n_{\text{phones}} \leq n_{\text{SDCCH}} = r_{\text{SDCCH}} \cdot t_{\text{attack}}$.

Example Computation. The following is an example, based on the peak values from our measurements gathered for the E-Plus network and as reflected in Ta-

Table 2: Example resource requirements for E-Plus.

Variable	Value	Reference
r_{request}	415 paging/min	Section 4.1
t_{response}	180 ms	Section 3.5
t_{sync}	745 ms	Section 4

ble 2. Based on the previous equations, at least $n_{\text{phones}} \approx 10.820$ phones are required to attack a typical location of E-Plus. Given the costs of the Motorola devices, this is a reasonably small amount. Each paging response attack lasts $t_{\text{attack}} \approx 1.564$ seconds. This allows up to $n_{\text{RACH}} \approx 125$ phones without a saturation of the RACH. For the SDCCH, the above formula yields to a maximum of $n_{\text{SDCCH}} \approx 53$ phones. It is also important to note that the number of phones is proportional to the impact. This means that half of the attacking phones would still be able disrupt service for half of the subscribers of a location area.

A single cell therefore provides enough resources to attack a complete location area of a considerably small operator. In practice these resources are shared with legit MO and MT traffic. The exact traffic patterns and the number of cells per location is unknown. Furthermore, a combination with the IMSI detach attack prevents phones that reside in the location area to generate further MT activity. As we cannot estimate these activities, we do not include this in our calculation. Nevertheless, the results indicate the required resources for a large-attack do not extensively exhaust the resources provided by a cell. Additionally, there is no technical limitation of distributing attacking phones across small number of different cells.

5 Countermeasures

In this section we present two countermeasures against the attacks we developed. Specifically, we propose different approaches to resolve both problems. A solution is required to not only fix the denial of service issue, but at the same time the MT service hijacking. Unlike the second prevention strategy, the first solution solves both issues at once, but requires a protocol change.

For the first solution, we propose a change to the paging protocol procedure [4]. To perform authentication, the network is sending a 128 bit random challenge (RAND) to the subscriber. Based on the secret key K_i that is only stored on the SIM card or in the authentication center of the network, the subscriber computes a 32 bit response value using the A3 algorithm. The so-called Signed Response (SRES) value is sent back to the

network. In the same fashion, the operator network computes SRES based on K_i as stored in the authentication center. If both SRES values match, the subscriber successfully authenticated itself to the network. However, as mandated in the GSM specification, the authentication is performed after the paging response is processed. The same principle applies to UMTS [12]. Therefore, the paging response itself is not authenticated. By adapting the protocol to include the RAND value in the paging request and SRES in the paging response, this can be changed. This implies that all of the paging responses are authenticated, which eliminates session hijacking. At the same time a paging response that includes authentication information can be used by the network to validate the response before changing the state to not expect further paging responses. Thus, also solving the denial of service attack. It is important to note that this requires a fresh RAND for every authentication to prevent replay attacks. This is similar to the protocol change proposed by Arapinis et al. [16], which encrypts the paging request using a shared session key called *unlikability key*. While they use this key to prevent tracking of subscribers via IMSI paging, the same modification also prevents our described attacks. Unfortunately, partly due to the difficulty of updating devices in the field, the industry is reluctant to apply new protocol changes to commercially deployed networks.

The second solution involves no protocol change, but has to dismantle each problem individually. MT session hijacking issue can be addressed, by enforcing authentication for each service request. This would also overcome MO session hijacking. In order to eliminate the denial of service attack, the MSC/VLR state machine needs to be changed. Specifically, the MSC/VLR has to be able to map all incoming paging responses to the correct service as long as no fully authenticated session exists. Accordingly, this circumvents the denial of service attack.

6 Related Work

In the last years, various attacks against cellular networks and their protocol stacks have been published. We separate related work into two parts. First, attacks that allow an adversary to impersonate a victim. Second, denial of service attacks in mobile networks that result in customers not being able to receive MT services.

Impersonation. In [28] Nohl and Melette demonstrated that it is possible to impersonate a subscriber for mobile originated services. By first sniffing a transaction over-the-air, cracking the session key K_c , and knowing a victim's TMSI, they were able to place a phone call on behalf of a victim. The authors of [24] used a femtocell device under their control in order to impersonate a subscriber that is currently booked into the femtocell. By

relaying authentication challenges to a victim, they were able to send SMS messages on behalf of the subscriber. Our work in this paper is different, as we do not attack MO services, but MT services. Thus, in our research, the considered victim is, e.g., the called party and not the caller. Contrary to attacking MO services, attacking MT services is time critical.

Denial of Service. We consider relevant types of denial of service attacks in mobile networks that affect MT services for subscribers. We determined three types of denial of service attacks that fulfill this requirement: attacks directly targeting the victim phone, attacks focusing on the network, and attacks affecting subscribers, but without direct communication.

The first type comprises DoS attacks that target mobile devices directly, most notably phones. These issues are usually baseband/phone specific and caused by implementation flaws. Several vulnerabilities have been discovered in mobile phones that can lead to code execution and denial of service conditions [33, 46]. Particularly, the Curse-of-Silence flaw enabled an adversary to disable the MT SMS functionality of specific Nokia devices [44]. In [38] Racic et al. demonstrate that it is possible to stealthily exhaust mobile phone batteries by repeatedly sending crafted MMS messages to a victim. Consequently, the phone battery will drain very fast, eventually the phone will switch off, and MT services can no longer be delivered to a victim. Our attack is inherently different from these kinds of attacks, because it is independent from the target device type and does not interact with the victim directly at all.

The second category consists of attacks that target the operator itself, and as a consequence also impact MT services for subscribers. These types are caused by design flaws. Spaar showed in [43] that it is feasible to exhaust channel resources of a base station by continuously requesting new channels on the RACH. Unlike our attack, this attack is limited to a single BTS and does not affect subscribers served by a different cell. Therefore, to attack a metropolitan area, an attacker needs to communicate with and attack every BTS in that area. Enck et al. [21] showed that it is practical to deny voice or SMS services within a specific geographical area, by sending a large number of short messages to subscribers in that area. Serror et al. [42] exhibit that similar conditions can be achieved in CDMA2000 networks by causing a significant paging load and delay of paging messages via Internet originating packets to phones. A comparable resource consumption attack for 3G/WiMax has been demonstrated by Lee et al. in [32]. As Traynor et al. outline [36], it is also possible to degrade the performance of large networks by utilizing a phone botnet and, e.g., repeatedly adding and deleting call forwarding settings. All of these attacks exhaust network resources mostly

due to generated signaling load. As a result, services can no longer be reliably offered to mobile subscribers, effectively causing denial of service conditions. This includes MT and MO services. We exploit a race condition in the MT paging procedure and do not attack the core network itself. Our attack does not intend to generate excessive signaling traffic in the network. As a result, it is not prevented by proposed mitigations for these kind of issues from previous research.

Our attack fits into the last of the three types of attacks that result in DoS for MT services. Most network attacks aim to abuse generated signaling to decrease the overall performance of the operator network. Attacks against mobile devices merely use the network as a bearer to deliver a specific payload to the phone. The third category is stipulated by attacks that target the mobile device itself, but do not send any payload to it. The aforementioned IMSI detach attack discovered by Munaut [34] can effectively cause that a service such as a call, will not result in paging requests by the network anymore. As described in section 4.4, this design flaw even supports our attack. Contrary to this vulnerability, the paging response attack allows us to precisely control when and where a victim can be reached or not. After sending a detach indication, an attacker cannot control anymore for how long this state is kept.

Our approach can be used either to hijack a session or to perform a denial of service attacks. We do attack mobile stations but neither by exhausting network resources, nor by directly communicating to the target device. We can target specific geographical areas, specific subscribers or a group of subscribers without the need to build a hit list of phone numbers residing in that area. Depending on the target, the attack can be either distributed or performed from a single phone. Additionally, the involved costs for this attack are as cheap as acquiring the required number of Motorola C1XX phones.

7 Conclusion

The trust in the security of cellular networks and specifically the widely used GSM standard has been shattered several times. Yet, attacks against mobile terminated services are a minority. The undisturbed operation of telecommunication networks is traditionally based on trust. The inherent trust that each subscriber and participant in communication plays by the rules. Nonetheless, due to several available and modifiable software and hardware projects for telecommunication, this trust relationship has to be considered broken. In this paper we showed how to exploit the trust in paging procedures on a broadcast medium. We demonstrated that it is possible to leverage a race condition in the paging protocol to a novel denial of service attack and the possibility to hi-

jacking mobile terminated services in GSM. Moreover, we showed that this attack can not only disturb communication for single subscribers, but can also greatly affect telephony in a larger geographical region formed by location areas. A motivated attacker can interrupt communication on a large scale by merely utilizing a set of inexpensive consumer devices that are available on the market. This is considerably more efficient compared to traditional radio jamming due to the broad frequency range of mobile carrier networks and the size of location areas. In order to mitigate these attacks, we propose two different countermeasures of which one does not require a protocol change. We strongly encourage future standards to consider threats caused by active attackers that tamper with user equipment and protocol stacks.

8 Acknowledgement

The authors would like to thank Dieter Spaar, Harald Welte, Holger Freyther, Benjamin Michéle, Alex Dent, and Dmitry Nedospasov for technical discussions and their help in reviewing this paper.

References

- [1] Routo Messaging. <http://www.routomessaging.com>.
- [2] Gartner Says Worldwide Smartphone Sales Soared in Fourth Quarter of 2011 With 47 Percent Growth. <http://www.gartner.com/it/page.jsp?id=1924314>, February 2012.
- [3] 3GPP. Digital cellular telecommunications system (Phase 2+); Network architecture (GSM 03.02 version 7.1.0 Release 1998). Tech. rep., 3rd Generation Partnership Project, 2000. 3GPP TS 03.02 V7.1.0.
- [4] 3GPP. Digital cellular telecommunications system (Phase 2+); Mobile radio interface layer 3 specification (3GPP TS 04.08 version 7.9.1 Release 1998). Tech. rep., 3rd Generation Partnership Project, 2001. 3GPP TS 04.08 V7.9.1.
- [5] 3GPP. Digital cellular telecommunications system (Phase 2+); Multiplexing and multiple access on the radio path (3GPP TS 05.02 version 8.9.0 Release 1999). Tech. rep., 3rd Generation Partnership Project, 2001. 3GPP TS 05.02 V8.9.0.
- [6] 3GPP. Digital cellular telecommunications system (Phase 2+); Base Station System - Mobile Services Switching Centre (BSS-MSC) Interface - Interface Principles (3GPP TS 08.02 version 8.0.1 Release 1999). Tech. rep., 3rd Generation Partnership Project, 2002. 3GPP TS 08.02 V8.0.1.
- [7] 3GPP. Digital cellular telecommunications system (Phase 2+); Mobile Station - Base Station System (MS - BSS) Interface Channel Structures and Access Capabilities (3GPP TS 04.03 version 8.0.2 Release 1999). Tech. rep., 3rd Generation Partnership Project, 2002. 3GPP TS 04.03 V8.0.2.
- [8] 3GPP. Digital cellular telecommunications system (Phase 2+); Numbering, addressing and identification (3GPP TS 03.03 version 7.8.0 Release 1998). Tech. rep., 3rd Generation Partnership Project, 2003. 3GPP TS 03.03 V7.8.0.
- [9] 3GPP. Digital cellular telecommunications system (Phase 2+); Security-related network functions (3GPP TS 03.20 version 8.6.0 Release 1999). Tech. rep., 3rd Generation Partnership Project, 2008. 3GPP TS 03.20 V8.6.0.

- [10] 3GPP. Digital cellular telecommunications system (Phase 2+); Radio transmission and reception (3GPP TS 45.005 version 9.1.0 Release 9). Tech. rep., 3rd Generation Partnership Project, 2010. 3GPP TS 45.005 V9.1.0.
- [11] 3GPP. Universal Mobile Telecommunications System (UMTS);Physical channels and mapping of transport channels onto physical channels (FDD)(3GPP TS 25.211 version 9.2.0 Release 9). Tech. rep., 3rd Generation Partnership Project, 2010. 3GPP TS 25.211 9.2.0.
- [12] 3GPP. Universal Mobile Telecommunications System (UMTS); LTE;3G security; Security architecture(3GPP TS 33.102 version 9.4.0 Release 9). Tech. rep., 3rd Generation Partnership Project, 2011. 3GPP TS 33.102 V9.4.0.
- [13] 3GPP. Digital cellular telecommunications system (Phase 2+); Mobile Switching Centre - Base Station system (MSC-BSS) interface; Layer 3 specification (3GPP TS 48.008 version 9.8.0 Release 9). Tech. rep., 3rd Generation Partnership Project, 2012. 3GPP TS 48.008 V9.8.0.
- [14] 3GPP. LTE;Evolved Universal Terrestrial Radio Access (E-UTRA); User Equipment (UE) procedures in idle mode(3GPP TS 36.304 version 9.9.0 Release 9). Tech. rep., 3rd Generation Partnership Project, 2012. 3GPP TS 36.304 V9.9.0.
- [15] 3GPP. Universal Mobile Telecommunications System (UMTS);User Equipment (UE) procedures in idle mode and procedures for cell reselection in connected mode(3GPP TS 25.304 version 9.8.0 Release 9). Tech. rep., 3rd Generation Partnership Project, 2012. 3GPP TS 25.304 V9.8.0.
- [16] ARAPINIS, M., MANCINI, L., RITTER, E., RYAN, M., GOLDE, N., REDON, K., AND BORGAONKAR, R. New Privacy Issues in Mobile Telephony: Fix and Verification. In *Proceedings of the 19th ACM Conference on Computer and Communications Security* (October 2012).
- [17] BARKAN, E., BIHAM, E., AND KELLER, N. Instant Ciphertext-Only Cryptanalysis of GSM Encrypted Communication. *J. Cryptol.* 21, 3 (Mar. 2008), 392–429.
- [18] Biryukov, A., Shamir, A., AND WAGNER, D. Real Time Cryptanalysis of A5/1 on a PC. In *Proceedings of the 7th International Workshop on Fast Software Encryption* (London, UK, UK, 2001), FSE '00, Springer-Verlag, pp. 1–18.
- [19] BOSWARTHICK, D., ELLOUMI, O., AND HERSENT, O. *M2M Communications: A Systems Approach*. Wiley, March 2012.
- [20] D. BURGESS ET AL. OpenBTS. <http://openbts.org>.
- [21] ENCK, W., TRAYNOR, P., McDANIEL, P., AND LA PORTA, T. Exploiting open functionality in SMS-capable cellular networks. In *Proceedings of the 12th ACM conference on Computer and communications security* (New York, NY, USA, 2005), CCS '05, ACM, pp. 393–404.
- [22] ETTUS. USRP. <http://www.ettus.com/products>, 2009.
- [23] FRANK A. STEVENSON. [A51] The call of Kraken. <http://web.archive.org/web/20100812204319/http://lists.lists.reflextor.com/pipermail/a51/2010-July/000683.html>, July 2010.
- [24] GOLDE, N., REDON, K., AND BORGAONKAR, R. Weaponizing Femtocells: The Effect of Rogue Devices on Mobile Telecommunications. In *Proceedings of the 19th Annual Network & Distributed System Security Symposium* (Feb. 2012).
- [25] H. WELTE. OpenBSC. <http://openbsc.osmocom.org>.
- [26] INFOSECURITY MAGAZINE. Indian company hacks GSM and usurps IMSI. <http://www.infosecurity-magazine.com/view/24680/indian-company-hacks-gsm-and-usurps-imsi/>, March 2012.
- [27] IP.ACCESS LTD. nanoBTS 1800. http://www.ipaccess.com/picocells/nanoBTS_picocells.php.
- [28] KARSTEN NOHL AND LUCA MELETTE. Defending mobile phones. <http://events.ccc.de/congress/2011/Fahrplan/events/4736.en.html>, December 2011.
- [29] KRELL, M. Crowdflow. <http://crowdflow.net>.
- [30] KUNE, D. F., KOELNDORFER, J., HOPPER, N., AND KIM, Y. Location leaks over the GSM air interface. In *Proceedings of the 19th Annual Network & Distributed System Security Symposium* (Feb. 2012).
- [31] LANDSPURG, T. OpenCellID. <http://opencellid.org>.
- [32] LEE, P. P. C., BU, T., AND WOO, T. On the detection of signaling DoS attacks on 3G/WiMax wireless networks. *Comput. Netw.* 53, 15 (2009), 2601–2616.
- [33] MULLINER, C., GOLDE, N., AND SEIFERT, J.-P. SMS of Death: From Analyzing to Attacking Mobile Phones on a Large Scale. In *Proceedings of the 20th USENIX Security Symposium* (San Francisco, CA, USA, August 2011).
- [34] MUNAUT, S. IMSI DETACH DoS. <http://security.osmocom.org/trac/ticket/2>, May 2010.
- [35] NOKIA SIEMENS NETWORKS. Nokia Siemens Networks promotes GSM for Machine to Machine applications. <http://www.nokiasiemensnetworks.com/news-events/press-room/press-releases/nokia-siemens-networks-promotes-gsm-for-machine-to-machine-applications>.
- [36] P. TRAYNOR, M. LIN, M. ONGTANG, V. RAO, T. JAEGER, T. LA PORTA, P. McDANIEL. On Cellular Botnets: Measuring the Impact of Malicious Devices on a Cellular Network Core. In *ACM Conference on Computer and Communications Security (CCS)* (November 2009).
- [37] PURPLELABS. Tsm30 firmware. <http://web.archive.org/web/20090325133430/http://sourceforge.net/projects/plabs>, November 2004.
- [38] RACIC, R., MA, D., AND CHEN, H. Exploiting MMS Vulnerabilities to Stealthily Exhaust Mobile Phone's Battery. In *Securecomm and Workshops, 2006* (28 2006-sept. 1 2006), pp. 1–10.
- [39] SECURITY RESEARCH LABS. A5/1 decryption project. <http://opensource.srlabs.de/projects/a51-decrypt>.
- [40] SECURITY RESEARCH LABS. Decrypting GSM phone calls. https://srlabs.de/decrypting_gsm/.
- [41] SECURITY RESEARCH LABS. GSM security map. <http://www.gsmap.org>.
- [42] SERROR, J., ZANG, H., AND BOLOT, J. C. Impact of paging channel overloads or attacks on a cellular network. In *Proceedings of the 5th ACM workshop on Wireless security* (New York, NY, USA, 2006), WiSe '06, ACM, pp. 75–84.
- [43] SPAAR, D. RACH flood DoS. <http://security.osmocom.org/trac/ticket/1>, November 2009.
- [44] T. ENGEL. Remote SMS/MMS Denial of Service - Curse Of Silence. <http://berlin.ccc.de/~tobias/cursesms.txt>, December 2008.
- [45] VARIOUS CONTRIBUTORS. Osmocom project. <http://osmocom.org>.
- [46] WEINMANN, R.-P. Baseband Attacks: Remote Exploitation of Memory Corruptions in Cellular Protocol Stacks. In *Proceedings of the 21st USENIX Workshop on Offensive Technologies* (Bellevue, WA, USA, August 2012).
- [47] WELTE, H., MUNAUT, S., EVERSBERG, A., AND OTHER CONTRIBUTORS. OsmocomBB. <http://bb.osmocom.org>.

Dowsing for overflows: A guided fuzzer to find buffer boundary violations

Istvan Haller

VU University Amsterdam

Asia Slowinska

VU University Amsterdam

Matthias Neugschwandtner

Vienna University of Technology

Herbert Bos

VU University Amsterdam

Abstract

Dowser is a ‘guided’ fuzzer that combines taint tracking, program analysis and symbolic execution to find buffer overflow and underflow vulnerabilities buried deep in a program’s logic. The key idea is that analysis of a program lets us pinpoint the right areas in the program code to probe and the appropriate inputs to do so.

Intuitively, for typical buffer overflows, we need consider only the code that accesses an array in a loop, rather than all possible instructions in the program. After finding all such candidate sets of instructions, we rank them according to an estimation of how likely they are to contain interesting vulnerabilities. We then subject the most promising sets to further testing. Specifically, we first use taint analysis to determine which input bytes influence the array index and then execute the program symbolically, making only this set of inputs symbolic. By constantly steering the symbolic execution along branch outcomes most likely to lead to overflows, we were able to detect deep bugs in real programs (like the `nginx` webserver, the `ircd` IRC server, and the `ffmpeg` videoplayer). Two of the bugs we found were previously undocumented buffer overflows in `ffmpeg` and the `poppler` PDF rendering library.

1 Introduction

We discuss *Dowser*, a ‘guided’ fuzzer that combines taint tracking, program analysis and symbolic execution, to find buffer overflow bugs buried deep in the program’s logic.

Buffer overflows are perennially in the top 3 most dangerous software errors [12] and recent studies suggest this will not change any time soon [41, 38]. There are two ways to handle them. Either we harden the software with memory protectors that terminate the program when an overflow occurs (at runtime), or we track down the vulnerabilities before releasing the software (e.g., in the testing phase).

Memory protectors include common solutions like shadow stacks and canaries [11], and more elaborate compiler extensions like WIT [3]. They are effective in preventing programs from being exploited, but they do not remove the overflow bugs themselves. Although it is better to crash than to allow exploitation, crashes are undesirable too!

Thus, vendors prefer to squash bugs beforehand and typically try to find as many as they can by means of fuzz testing. Fuzzers feed programs invalid, unexpected, or random data to see if they crash or exhibit unexpected behavior¹. As an example, Microsoft made fuzzing mandatory for every untrusted interface for every product, and their fuzzing solution has been running 24/7 since 2008 for a total of over 400 machine years [18].

Unfortunately, the effectiveness of most fuzzers is poor and the results rarely extend beyond shallow bugs. Most fuzzers take a ‘blackbox’ approach that focuses on the input format and ignores the tested software target. Blackbox fuzzing is popular and fast, but misses many relevant code paths and thus many bugs. Blackbox fuzzing is a bit like shooting in the dark: you have to be lucky to hit anything interesting.

Whitebox fuzzing, as implemented in [18, 7, 10], is more principled. By means of symbolic execution, it exercises all possible execution paths through the program and thus uncovers all possible bugs – although it may take years to do. Since full symbolic execution is slow and does not scale to large programs, it is hard to use it to find complex bugs in large programs [7, 10]. In practice, the aim is therefore to first cover as much unique code as possible. As a result, bugs that require a program to execute the same code many times (like buffer overflows) are hard to trigger except in very simple cases.

Eventual completeness, as provided by symbolic execution, is both a strength and a weakness, and in this paper, we evaluate the exact opposite strategy. Rather

¹See <http://www.fuzzing.org/> for a collection of available fuzzers

than testing all possible execution paths, we perform *spot checks* on a small number of code areas that look likely candidates for buffer overflow bugs and test each in turn.

The drawback of our approach is that we execute a symbolic run for each candidate code area—in an iterative fashion. Moreover, we can discover buffer overflows only in the loops that we can exercise. On the other hand, by homing in on promising code areas directly, we speed up the search considerably, and manage to find complicated bugs in real programs that would be hard to find with most existing fuzzers.

Contributions The goal we set ourselves was to develop an efficient fuzzer that actively *searches* for buffer overflows *directly*. The key insight is that careful analysis of a program lets us pinpoint the right places to probe and the appropriate inputs to do so. The main contribution is that our fuzzer directly zooms in on these buffer overflow candidates and explores a novel ‘spot-check’ approach in symbolic execution.

To make the approach work, we need to address two main challenges. The first challenge is *where* to steer the execution of a program to increase the chances of finding a vulnerability. Whitebox fuzzers ‘blindly’ try to execute as much of the program as possible, in the hope of hitting a bug eventually. Instead, *Dowser* uses information about the target program to identify code that is most likely to be vulnerable to a buffer overflow.

For instance, buffer overflows occur (mostly) in code that accesses an array in a loop. Thus, we look for such code and ignore most of the remaining instructions in the program. Furthermore, *Dowser* performs static analysis of the program to *rank* such accesses. We will evaluate different ranking functions, but the best one so far ranks the array accesses according to complexity. The intuition is that code with convoluted pointer arithmetic and/or complex control flow is more prone to memory errors than straightforward array accesses. Moreover, by focusing on such code, *Dowser* prioritizes bugs that are complicated—typically, the kind of vulnerabilities that static analysis or random fuzzing cannot find. The aim is to reduce the time wasted on shallow bugs that could also have been found using existing methods. Still, other rankings are possible also, and *Dowser* is entirely agnostic to the ranking function used.

The second challenge we address is *how* to steer the execution of a program to these “interesting” code areas. As a baseline, we use *concolic* execution [43]: a combination of concrete and symbolic execution, where the concrete (fixed) input starts off the symbolic execution. In *Dowser*, we enhance concolic execution with two optimizations.

First, we propose a new path selection algorithm. As we saw earlier, traditional symbolic execution aims

at code coverage—maximizing the fraction of individual branches executed [7, 18]. In contrast, we aim for *pointer value* coverage of *selected* code fragments. When *Dowser* examines an interesting pointer dereference, it steers the symbolic execution along branches that are likely to alter the value of the pointer.

Second, we reduce the amount of symbolic input as much as we can. Specifically, *Dowser* uses dynamic taint analysis to determine which input bytes influence the pointers used for array accesses. Later, it treats only these inputs as symbolic. While taint analysis itself is not new, we introduce novel optimizations to arrive at a set of symbolic inputs that is as *accurate* as possible (with neither too few, nor too many symbolic bytes).

In summary, *Dowser* is a new fuzzer targeted at vendors who want to test their code for buffer overflows and underflows. We implemented the analyses of *Dowser* as LLVM [23] passes, while the symbolic execution step employs S2E [10]. Finally, *Dowser* is a *practical* solution. Rather than aiming for all possible security bugs, it specifically targets the class of buffer overflows (one of the most, if not the most, important class of attack vectors for code injection). So far, *Dowser* found several real bugs in complex programs like `nginx`, `ffmpeg`, and `inspircd`. Most of them are extremely difficult to find with existing symbolic execution tools.

Assumptions and outline Throughout this paper, we assume that we have a test suite that allows us to reach the array accesses. Instructions that we cannot reach, we cannot test. In the remainder, we start with a big picture and the running example (Section 2). Then, we discuss the three main components of *Dowser* in turn: the selection of interesting code fragments (Section 3), the use of dynamic taint analysis to determine which inputs influence the candidate instructions (Section 4), and our approach to nudge the program to trigger a bug during symbolic execution (Section 5). We evaluate the system in Section 6, discuss the related projects in Section 7. We conclude in Section 8.

2 Big picture

The main goal of *Dowser* is to manipulate the pointers that instructions use to access an array in a loop, in the hope of forcing a buffer overrun or underrun.

2.1 Running example

Throughout the paper, we will use the function in Figure 1 to illustrate how *Dowser* works. The example is a simplified version of a buffer underrun vulnerability in the `nginx-0.6.32` web server [1]. A specially crafted

A buffer underrun vulnerability in nginx

```

[1] int ngx_http_parse_complex_uri(ngx_http_request_t *r)
[2] {
[3]     state = sw_usual;
[4]     u_char* p = r->uri_start; // user input
[5]     u_char* u = r->uri.data; // store normalized uri here
[6]     u_char ch = *p++;
[7] 
[8]     while (p <= r->uri_end) {
[9]         switch (state) {
[10]             case sw_usual:
[11]                 if (ch == '/') {
[12]                     state = sw_slash; *u++ = ch;
[13]                     else /* many more options here */
[14]                         ch = *p++; break;
[15] 
[16]             case sw_slash:
[17]                 if (ch == '/') {
[18]                     *u++ = ch;
[19]                     else if (ch == '=') {
[20]                         state = sw_dot; *u++ = ch;
[21]                         else /* many more options here */
[22]                             ch = *p++; break;
[23] 
[24]             case sw_dot:
[25]                 if (ch == '=') {
[26]                     state = sw_dot_dot; *u++ = ch;
[27]                     else /* many more options here */
[28]                         ch = *p++; break;
[29] 
[30]             case sw_dot_dot:
[31]                 if (ch == '=') {
[32]                     state = sw_slash; u -= 4;
[33]                     while (*u-1 != '/') u--;
[34]                     else /* many more options here */
[35]                         ch = *p++; break;
[36] 
[37]         }
[38]     }
}

```

Fig. 1: A simplified version of a buffer underrun vulnerability in nginx.

input tricks the program into setting the `u` pointer to a location outside its buffer boundaries. When this pointer is later used to access memory, it allows attackers to overwrite a function pointer, and execute arbitrary programs on the system.

Figure 1 presents only an excerpt from the original function, which in reality spans approximately 400 lines of C code. It contains a number of additional options in the `switch` statement, and a few nested conditional `if` statements. This complexity severely impedes detecting the bug by both static analysis tools and symbolic execution engines. For instance, when we steered S2E [10] all the way down to the vulnerable function, and made solely the seven byte long uri path of the HTTP message symbolic, it took over 60 minutes to track down the problematic scenario. A more scalable solution is necessary in practice. Without these hints, S2E did not find the bug at all during an eight hour long execution.² In contrast, *Dowser* finds it in less than 5 minutes.

The primary reason for the high cost of the analysis in S2E is the large number of conditional branches which depend on (symbolic) input. For each of the branches, symbolic execution first checks whether either the condition or its negation is satisfiable. When both branches are feasible, the default behavior is to examine both. This

Nginx is a web server—in terms of market share across the million busiest sites, it ranks third in the world. At the time of writing, it hosts about 22 million domains worldwide. Versions prior to 0.6.38 had a particularly nasty vulnerability [1].

When nginx receives an HTTP request, the parsing function `nginx_http_parse_complex_uri`, first normalizes a uri path in `p=r->uri_start` (line 4), storing the result in a heap buffer pointed to by `u=r->uri.data` (line 5). The `while-switch` implements a state machine that consumes the input one character at a time, and transform it into a canonical form in `u`.

The source of the vulnerability is in the `sw_dot_dot` state. When provided with a carefully crafted path, nginx wrongly sets the beginning of `u` to a location somewhere below `r->uri.data`. Suppose the uri is "`//.../foo`". When `p` reaches "`/foo`", `u` points to `(r->uri.data+4)`, and `state` is `sw_dot_dot` (line 30). The routine now decreases `u` by 4 (line 32), so that it points to `r->uri.data`. As long as the memory below `r->uri.data` does not contain the character `/`, `u` is further decreased (line 33), even though it crosses buffer boundaries. Finally, the user provided input ("`foo`") is copied to the location pointed to by `u`.

In this case, the overwritten buffer contains a pointer to a function, which will be eventually called by nginx. Thus the vulnerability allows attackers to modify a function pointer, and execute an arbitrary program on the system.

It is a complex bug that is hard to find with existing solutions. The many conditional statements that depend on symbolic input are problematic for symbolic execution, while input-dependent indirect jumps are also a bad match for static analysis.

procedure results in an exponentially growing number of paths.

This real world example shows the need for (1) focusing the powerful yet expensive symbolic execution on the most interesting cases, (2) making informed branch choices, and (3) minimizing the amount of symbolic data.

2.2 High-level overview

Figure 2 illustrates the overall *Dowser* architecture.

First, it performs a data flow analysis of the target program, and ranks all instructions that access buffers in loops ①. While we can rank them in different ways and *Dowser* is agnostic as to the ranking function we use, our experience so far is that an estimation of complexity works best. Specifically, we rank calculations and conditions that are more complex higher than simple ones. In Figure 1, `u` is involved in three different operations, i.e., `u++`, `u--`, and `u-=4`, in multiple instructions inside a loop. As we shall see, these intricate computations place the dereferences of `u` in the top 3% of the most complex pointer accesses across nginx.

In the second step ②, *Dowser* repeatedly picks high-ranking accesses, and selects test inputs which exercise them. Then, it uses dynamic taint analysis to determine which input bytes influence pointers dereferenced in the candidate instructions. The idea is that, given the for-

²All measurements in the paper use the same environment as in Section 6.

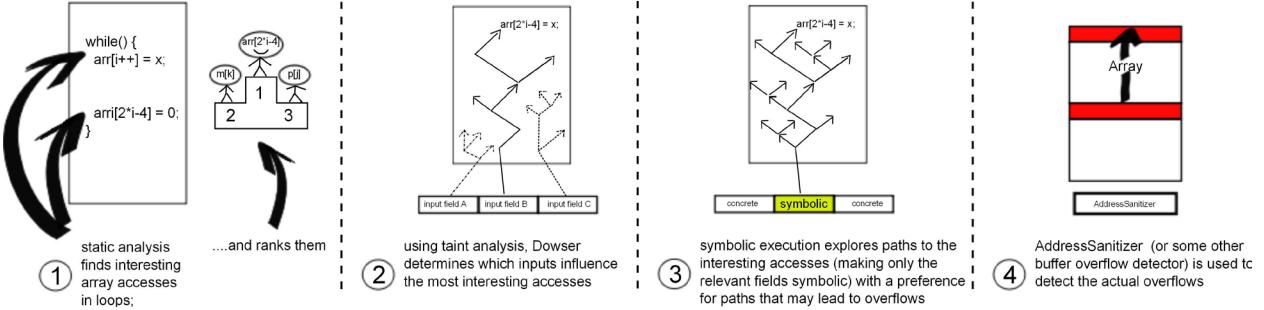


Fig. 2: *Dowser*—high-level overview.

mat of the input, *Dowser* fuzzes (i.e., treats as symbolic), only those fields that affect the potentially vulnerable memory accesses, and keeps the remaining ones unchanged. In Figure 1, we learn that it is sufficient to treat the uri path in the HTTP request as symbolic. Indeed, the computations inside the vulnerable function are independent of the remaining part of the input message.

Next ③, for each candidate instruction and the input bytes involved in calculating the array pointer, *Dowser* uses symbolic execution to try to nudge the program toward overflowing the buffer. Specifically, we execute symbolically the loop that contains the candidate instructions (and thus should be tested for buffer overflows)—treating only the relevant bytes as symbolic. As we shall see, a new path selection algorithm helps to guide execution to a possible overflow quickly.

Finally, we detect any overflow that may occur. Just like in whitebox fuzzers, we can use any technique to do so (e.g., Purify, Valgrind [30], or BinArmor [37]). In our work, we use Google’s AddressSanitizer [34] ④. It instruments the protected program to ensure that memory access instructions never read or write so called, “poisoned” red zones. Red zones are small regions of memory inserted inbetween any two stack, heap or global objects. Since they should never be addressed by the program, an access to them indicates an illegal behavior. This policy detects sequential buffer over- and underflows, and some of the more sophisticated pointer corruption bugs. This technique is beneficial when searching for new bugs since it will also trigger on silent failures, not just application crashes. In the case of nginx, AddressSanitizer detects the underflow when the `u` pointer reads memory outside its buffer boundaries (line 33).

We explain step ① (static analysis) in Section 3, step ② (taint analysis) in Section 4, and step ③ (guided execution) in Section 5.

3 Dowsing for candidate instructions

Previous research has shown that software complexity metrics collected from software artifacts are helpful in finding vulnerable code components [16, 44, 35, 32]. However, even though complexity metrics serve as useful indicators, they also suffer from low precision or recall values. Moreover, most of the current approaches operate at the granularity of modules or files, which is too coarse for the directed symbolic execution in *Dowser*.

As observed by Zimmermann et al. [44], we need metrics that exploit the unique characteristics of vulnerabilities, e.g., buffer overflows or integer overruns. In principle, *Dowser* can work with any metric capable of ranking groups of instructions that access buffers in a loop. So, the question is how to design a good metric for complexity that satisfies this criterion? In the remainder of this section, we introduce one such metric: a heuristics-based approach that we specifically designed for the detection of potential buffer overflow vulnerabilities.

We leverage a primary pragmatic reason behind complex buffer overflows: convoluted pointer computations are hard to follow by a programmer. Thus, we focus on ‘complex’ array accesses realized inside loops. Further, we limit the analysis to pointers which evolve together with loop induction variables, i.e., are repeatedly updated to access (various) elements of an array.

Using this metric, *Dowser* ranks buffer accesses by evaluating the complexity of data- and control-flows involved with the array index (pointer) calculations. For each loop in the program, it first statically determines (1) the set of all instructions involved in modifying an array pointer (we will call this a pointer’s *analysis group*), and (2) the conditions that guard this analysis group, e.g., the condition of an `if` or `while` statement containing the array index calculations. Next, it labels all such sets with scores reflecting their complexity. We explain these steps in detail in Sections 3.1, 3.2, and 3.3.

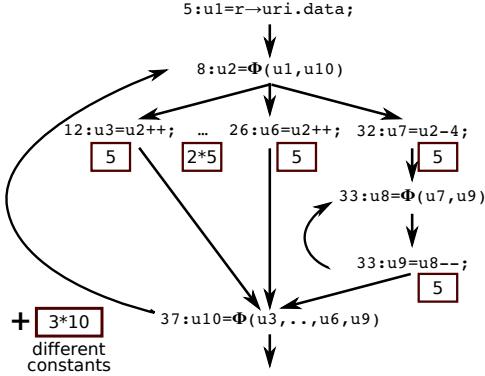


Fig. 3: Data flow graph and analysis group associated with the pointer p from Figure 1. For the sake of clarity, the figure presents pointer arithmetic instructions in pseudo code. The PHI nodes represent locations where data is merged from different control-flows. The numbers in the boxes represent points assigned by *Dowser*.

3.1 Building analysis groups

Suppose a pointer p is involved in an “interesting” array access instruction acc_p in a loop. The *analysis group* associated with acc_p , $\text{AG}(\text{acc}_\text{p})$, collects all instructions that influence the value of the dereferenced pointer during the execution of the loop.

To determine $\text{AG}(\text{acc}_\text{p})$, we compute an intraprocedural data flow graph representing operations in the loop that compute the value of p dereferenced in acc_p . Then, we check if the graph contains cycles. A cycle indicates that the value of p in a previous loop iteration affects its value in the current one, so p depends on the loop induction variable.

As mentioned before, this part of our work is built on top of the LLVM [23] compiler infrastructure. The static single assignment (SSA) form provided by LLVM translates directly to data flow graphs. Figure 3 shows an example. Observe that, since all dereferences of pointer u share their data flow graph, they also form a single analysis group. Thus, when *Dowser* later tries to find an illegal array access within this analysis group, it tests all the dereferences at the same time—there is no need to consider them separately.

3.2 Conditions guarding analysis groups

It may happen that the data flow associated with an array pointer is simple, but the value of the pointer is hard to follow due to some complex control changes. For this reason, *Dowser* ranks also control flows: the conditions that influence an analysis group.

Say that an instruction manipulating the array pointer p is *guarded* by a condition on a variable var , e.g., `if(var<10){*p++=0;}`. If the value of var is diffi-

cult to keep track of, so is the value of p . To assess the complexity of var , *Dowser* analyzes its data flow, and determines the analysis group, $\text{AG}(\text{var})$ (as discussed in Section 3.1). Moreover, we recursively analyze the analysis groups of other variables influencing var and p inside the loop. Thus, we obtain a number of analysis groups which we rank in the next step (Section 3.3).

3.3 Scoring array accesses

For each array access realized in a loop, *Dowser* assesses the complexity of the analysis groups constructed in Sections 3.1 and 3.2. For each analysis group, it considers all instructions, and assigns them points. The more points an AG cumulatively scores, the more complex it is. The overall rank of the array access is determined by the maximum of the scores. Intuitively, it reflects the most complex component.

The scoring algorithm should provide roughly the same results for semantically identical code. For this reason, we enforce the optimizations present in the LLVM compiler (e.g., to eliminate common subexpressions). This way, we minimize the differences in (the amount of) instructions arising from the compiler options. Moreover, we analyzed the LLVM code generation strategies, and defined a powerful set of *equivalence rules*, which minimize the variation in the scores assigned to syntactically different but semantically equivalent code. We highlight them below.

Table 1 introduces all types of instructions, and discusses their impact on the final score. In principle, all common instructions involved in array index calculations are of the order of 10 points, except for the two instructions that we consider risky: pointer casts and functions that return non-pointer values used in pointer calculation.

The absolute penalty for each type of instruction is not very important. However, we ensure that the points reflect the difference in complexity between various code fragments, instead of giving all array accesses the same score. That is, instructions that complicate the array index contribute to the score, and instructions that complicate the index a lot also score very high, relative to other instructions. In Section 6, we compare our complexity ranking to alternatives.

4 Using tainting to find inputs that matter

Once *Dowser* has ranked array accesses in loops in order of complexity, we examine them in turn. Typically, only a small segment of the input affects the execution of a particular analysis group, so we want to search for a bug by modifying solely this part of the input, while keeping the rest constant (refer to Section 5). In the current section, we explain how *Dowser* identifies the link

Instructions	Rationale/Equivalence rules	Points
Array index manipulations		
Basic index arithmetic instr., i.e., addition and subtraction	GetElemPtr, that increases or decreases a pointer by an index, scores the same. Thus, operations on pointers are equivalent to operations on offsets. An instruction scores 1 if it modifies a value which is not passed to the next loop iteration.	1 or 5
Other index arithmetic instr. e.g., division, shift, or xor	These instructions involve more complex pointer calculations than the standard add or sub. Thus, we penalize them more.	10
Different constant values	Multiple constants used to modify a pointer make its value hard to follow. It is easier to keep track of a pointer that always increases by the same value.	10 per value
Constants used to access fields of structures	We assume that compilers handle accesses to structures correctly. We only consider constants used to compute the index of an array, and not the address of a field.	0
Numerical values determined outside the loop	Though in the context of the loop they are just constants, the compiler cannot predict their values. Thus they are difficult to reason about and more error prone.	30
Non-inlined functions returning non-pointer values	Since decoupling the computation of a pointer from its use might easily lead to mistakes, we heavily penalize this operation.	500
Data movement instructions	Moving (scalar or pointer) data does not add to the complexity of computations.	0
Pointer manipulations		
Load a pointer calculated outside the loop	It denotes retrieving the base pointer of an object, or using memory allocators. We treat all <i>remote</i> pointers in the same way - all score 0.	0
GetElemPtr	An LLVM instruction that computes a pointer from a base and offset(s). (See add.)	1 or 5
Pointer cast operations	Since the casting instructions often indicate operations that are not equivalent to the standard pointer manipulations (listed above), they are worth a close inspection.	100

Table 1: Overview of the instructions involved in pointer arithmetic operations, and their penalty points.

between the components of the program input and the different analysis groups. Observe that this result also benefits other bug finding tools based on fuzzing, not just *Dowser* and concolic execution.

We focus our discussion on an analysis group $AG(acc_p)$ associated with an array pointer dereference acc_p . We assume that we can obtain a test input I that exercises the potentially vulnerable analysis group. While this may not always be true, we believe it is a reasonable assumption. Most vendors have test suites to test their software and they often contain at least one input which exercises each *complex* loop.

4.1 Baseline: dynamic taint analysis

As a basic approach, *Dowser* performs dynamic taint analysis (DTA) [31] on the input I (tainting each input byte with a unique color, and propagating the colors on data movement and arithmetic operations). Then, it logs all colors and input bytes involved in the instructions in $AG(acc_p)$. Given the format of the input, *Dowser* maps these bytes to individual fields. In Figure 1, *Dowser* finds out that it is sufficient to treat uri as symbolic.

The problem with DTA, as sketched above, is that it misses *implicit flows* (also called *control dependencies*) entirely [14, 21]. Such flows have no direct assignment of a tainted value to a variable—which would be propagated by DTA. Instead, the value of a variable is completely determined by the value of a tainted variable in a condition. In Figure 1, even though the value of u in

line 12 is dependent on the tainted character ch in line 11, the taint does not flow directly to u , so DTA would not report the dependency. Implicit flows are notoriously hard to track [36, 9], but ignoring them completely reduces our accuracy. *Dowser* therefore employs a solution that builds on the work by Bao et al. [6], but with a novel optimization to increase the accuracy of the analysis (Section 4.2).

Like Bao et al. [6], *Dowser* implements *strict control dependencies*. Intuitively, we propagate colors only on the most informative (or, information preserving) dependencies. Specifically, we require a direct comparison between a tainted variable and a compile time constant. For example, in Figure 1, we propagate the color of ch in line 11 to the variables $state$ and u in line 12. However, we would keep $state$ and u untainted if the condition in line 11 for instance had been either "*if(ch != '/')*" or "*if(ch < '/')*". As implicit flows are not the focus of this paper we refer interested readers to [6] for details.

4.2 Field shifting to weed out false dependencies

Improving on the handling of strict control dependencies by Bao et al. [6], described above, *Dowser* adds a novel technique to prevent overtainting due to false dependencies. The problems arise when the order of fields in an input format is not fixed, e.g., as in HTTP, SMTP (and the commandline for most programs). The approach from [6] may falsely suggest that a field is dependent on all fields that were extracted so far.

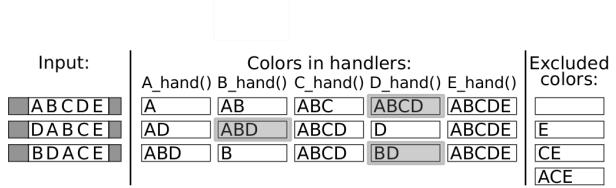


Fig. 4: The figure shows how *Dowser* shuffles an input to determine which fields really influence an analysis group. Suppose a parser extracts fields of the input one by one, and the analysis group depends on the fields B and D (with colors B and D, respectively). *Colors in handlers* show on which fields the subsequent handlers are strictly dependent [6], and the shaded rectangle indicates the colors propagated to the analysis group. *Excluded colors* are left out of our analysis.

For instance, `lighttpd` reads new header fields in a loop and compares them to various options, roughly as follows:

```
while () {
    if(cmp(field, "Content") == 0)
    ...
    else if(cmp(field, "Range") == 0)
    ...
    else exit (-1);
    field = extract_new_header_field();
}
```

As the parser tests for equivalence, the implicit flow will propagate from one field to the next one, even if there is no real dependency at all! Eventually, the last field appears to depend on the whole header.

Dowser determines which options really matter for the instructions in an analysis group by *shifting* the fields whose order is not fixed. Refer to Figure 4, and suppose we have run the program with options A, B, C, D, and E, and our analysis group really depends on B and D. Once the message gets processed, we see that the AG does not depend on E, so E can be excluded from further analysis. Since the last observed color, D, has a direct influence on the AG, it is a true dependence. By performing a circular shift and re-trying with the order D, A, B, C, E, *Dowser* finds only the colors corresponding to A, B, D. Thus, we can leave C out of our analysis. After the next circular shift, *Dowser* reduces the colors to B and D only.

The optimization is based on two observations: (1) the last field propagated to the AG has a direct influence on the AG, so it needs to be kept, (2) all fields beyond this one are guaranteed to have no impact on the AG. By performing circular shifts, and running DTA on the updated input, *Dowser* drops the undue dependencies.

Even though this optimization requires some minimal knowledge of the input, we do not need full understanding of the input grammar, like the contents or effects of fields. It is sufficient to identify the fields whose order is not fixed. Fortunately, such information is available for many applications—especially when vendors test their own code.

5 Exploring candidate instructions

Once we have learnt which part of the program input influences the analysis group $\text{AG}(\text{acc}_p)$, we fuzz this part, and we try to nudge the program toward using the pointer p in an illegal way. More technically, we treat the interesting component of the input as symbolic, the remaining part as fixed (concrete), and we execute the loop associated with $\text{AG}(\text{acc}_p)$ symbolically.

However, since in principle the cost of a complete loop traversal is exponential, loops present one of the hardest problems for symbolic execution [19]. Therefore, when analyzing a loop, we try to select those paths that are most promising in our context. Specifically, *Dowser* prioritizes paths that show a potential for knotty pointer arithmetic. As we show in Section 6, our technique significantly optimizes the search for an overflow.

Dowser's loop exploration procedure has two main phases: learning, and bug finding. In the *learning phase*, *Dowser* assigns each branch in the loop a weight approximating the probability that a path following this direction contains new pointer dereferences. The weights are based on statistics on the variety of pointer values observed during an execution of a short symbolic input.

Next, in the *bug finding phase*, *Dowser* uses the weights determined in the first step to filter our uninteresting parts of the loop, and prioritize the important paths. Whenever the weight associated with a certain branch is 0, *Dowser* does not even try to explore it further. In the vulnerable `nginx` parsing loop from which Figure 1 shows an excerpt, only 19 out of 60 branches scored a non-zero value, so were considered for the execution. In this phase, the symbolic input represents a real world scenario, so it is relatively long. Therefore, it would be prohibitively expensive to be analyzed using a popular symbolic execution tool.

In Section 5.1, we briefly review the general concept of concolic execution, and then we discuss the two phases in Sections 5.2 and 5.3, respectively.

5.1 Baseline: concrete + symbolic execution

Like DART and SAGE [17, 18], *Dowser* generates new test inputs by combining concrete and symbolic execution. This technique is known as *concolic* execution [33]. It runs the program on a concrete input, while gathering symbolic constraints from conditional statements encountered along the way. To test alternative paths, it systematically negates the collected constraints, and checks whether the new set is satisfiable. If so, it yields a new input. To bootstrap the procedure, *Dowser* takes a test input which exercises the analysis group $\text{AG}(\text{acc}_p)$.

As mentioned already, a challenge in applying this approach is how to select the paths to explore first. The

classic solution is to use depth first exploration of the paths by backtracking [22]. However, since doing so results in an exponentially growing number of paths to be tested, the research community has proposed various heuristics to steer the execution toward unexplored regions. We discuss these techniques in Section 7.

5.2 Phase 1: learning

The aim of the learning phase is to rate the `true` and `false` directions of all conditional branches that depend on the symbolic input in the loop L . For each branch, we evaluate the likelihood that a particular outcome will lead to unique pointer dereferences (i.e., dereferences that we do not expect to find in the alternative outcome). Thus, we answer the question of how much we expect to gain when we follow this path, rather than the alternative. We encode this information into *weights*.

Specifically, the weights represent the likelihood of unique *access patterns*. An access pattern of the pointer p is the sequence of all values of p dereferenced during the execution of the loop. In Figure 1, when we denote the initial value of u by u_0 , then the input " $//\dots/$ " triggers the following access pattern of the pointer u : $(u_0, u_0+1, u_0+2, u_0-2, \dots)$.

To compute the weights, we learn about the effects of individual branches. In principle, each of them may (a) directly affect the value of a pointer, (b) be a precondition for another important branch, or (c) be irrelevant from the computation’s standpoint. To distinguish between these cases, *Dowser* analyzes all possible executions of a *short* symbolic input. By comparing the sets of p ’s access patterns observed for both outcomes of a branch, it discovers which branches do not influence the diversity of pointer dereferences (i.e., are irrelevant).

Symbolic input In Section 4, we identified which part of the test input I we need to make symbolic. We denote this by I_S . In the learning phase, *Dowser* executes the loop L exhaustively. For performance reasons, we therefore further limit the amount of symbolic data and make only a short fragment of I_S symbolic. For instance, for Figure 1, the learning phase makes only the first 4 bytes of `uri` symbolic (not enough to trigger the bug), while scaling up to 50 symbolic bytes in the bug finding phase.

Algorithm *Dowser* exhaustively executes L on a short symbolic input, and records how the decisions taken at conditional branch statements influence pointer dereference instructions. For each branch b along the execution path, we retain the access pattern of p realized during this execution, $AP(p)$. We informally interpret it as “if you choose the `true` (respectively, `false`) direction of the branch b , expect access pattern $AP(p)$ (respectively, $AP'(p)$)”. This procedure results in two sets of access patterns for each branch statement, for the taken

and non-taken branch, respectively. The final weight of each direction is the fraction of the access patterns that were unique for the direction in question, i.e., were not observed when the opposite one was taken.

The above description explains the intuition behind the learning mechanism, but the full algorithm is more complicated. The problem is that a conditional branch b might be exercised multiple times in an execution path, and it is possible that all the instances of b influence the access pattern observed.

Intuitively, to allow for it, we do not associate access patterns with just a single decision taken on b (`true` or `false`). Rather, each time b is exercised, we also retain which directions were previously chosen for b . Thus, we still collect “expected” access patterns if the `true` (respectively, `false`) direction of b is followed, but we augment them with a precondition. This way, when we compare the `true` and `false` sets to determine the weights for b , we base the scores on a deeper understanding of how an access pattern was reached.

Discussion It is important for our algorithm to avoid false negatives: we should not incorrectly flag a branch as irrelevant—it would preclude it from being explored in the bug finding phase. Say that `instr` is an instruction that dereferences the pointer p . To learn that a branch directly influences `instr`, it suffices to execute it. Similarly, since branches retain full access patterns of p , the information about `instr` being executed is also “propagated” to all its preconditions. Thus, to completely avoid false negatives, the algorithm would require full coverage of the instructions in an analysis group. We stress that we need to exercise all instructions, and not all paths in a loop. As observed by [7], exhaustive executions of even short symbolic inputs provide excellent instruction coverage in practice.

While false positives are undesirable as well, they only cause *Dowser* to execute more paths in the second phase than absolutely necessary. Due to the limited path coverage, there are corner cases, when false positives can happen. Even so, in `nginx`, only 19 out of 60 branches scored a non-zero value, which let us execute the complex loop with a 50-byte-long symbolic input.

5.3 Phase 2: hunting bugs

In this step, *Dowser* executes symbolically a real-world sized input in the hope of finding a value that triggers a bug. *Dowser* uses the feedback from the learning phase (Section 5.2) to steer its symbolic execution toward new and interesting pointer dereferences. The goal of our heuristic is to avoid execution paths that do not bring any new pointer manipulation instructions. Thus, *Dowser* shifts the target of symbolic execution from traditional *code coverage* to *pointer value coverage*.

Dowser's strategy is explicitly dictated by the weights. As a baseline, the execution follows a depth-first exploration, and when *Dowser* is about to select the direction of a branch b that depends on the symbolic input, it adheres to the following rules:

- If both the `true` and `false` directions of b have weight 0, we do not expect b to influence the variety of access patterns. Thus, *Dowser* chooses the direction randomly, and does not intend to examine the other direction.
- If only one direction has a non-zero weight, we expect to observe unique access patterns only when the execution paths follows this direction, and *Dowser* favors it.
- If both of b 's directions have non-zero weights, both the `true` and `false` options may bring unique access patterns. *Dowser* examines both directions, and schedules them in order of their weights.

Intuitively, *Dowser*'s symbolic execution tries to select paths that are more likely to lead to overflows.

Guided fuzzing This concludes our description of *Dowser*'s architecture. To summarize, *Dowser* helps fuzzing by: (1) finding “interesting” array accesses, (2) identifying the inputs that influence the accesses, and (3) fuzzing intelligently to cover the array. Moreover, the targeted selection procedure based on pointer value coverage and the small number of symbolic input values allow *Dowser* to find bugs quickly and scale to larger applications. In addition, the ranking of array accesses permits us to zoom in on more complicated array accesses.

6 Evaluation

In this section, we first zoom in on the running example of `nginx` from Figure 1 to evaluate individual components of the system in detail (Section 6.1). In Section 6.2, we consider seven real-world applications. Based on their vulnerabilities, we evaluate our dowsing mechanism. Finally, we present an overview of the attacks detected by *Dowser*.

Since *Dowser* uses a ‘spot-check’ rather than ‘code coverage’ approach to bug detection, it must analyze each complex analysis group separately, starting with the highest ranking one, followed by the second one, and so on. Each of them runs until it finds a bug or gets terminated. The question is when we should terminate a symbolic execution run. Since symbolic execution of a single loop is highly optimized in *Dowser*, we found each bug in less than 11 minutes, so we execute each symbolic run for a maximum of 15 minutes.

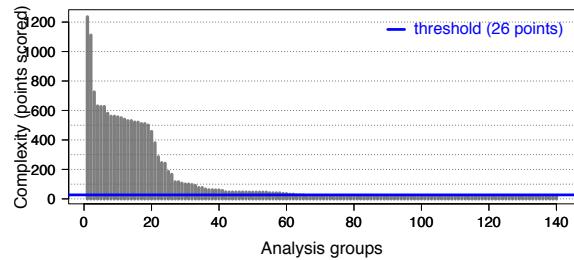


Fig. 5: Scores of the analysis groups in `nginx`.

Our test platform is a Linux 3.1 system with an Intel(R) Core(TM) i7 CPU clocked at 2.7GHz with 4096KB L2 cache. The system has 8GB of memory. For our experiments we used an OpenSUSE 12.1 install. We ran each test multiple times and present the median.

6.1 Case study: Nginx

In this section, we evaluate each of the main steps of our fuzzer by looking at our case study of `nginx` in detail.

6.1.1 Dowsing for candidate instructions

We measure how well *Dowser* highlights potentially faulty code and filters out the uninteresting fragments.

Our first question is whether we can filter out all the simple loops and focus on the more interesting ones. This turns out to be simple. Given the complexity scoring function from Section 3, we find that across all applications all analysis groups with a score less than 26 use just a single constant and at most two instructions modifying the offset of an array. Thus, in the remainder of our evaluation, we set our cut-off threshold to 26 points.

As shown in Table 2, `nginx` has 517 outermost loops, and only 140 analysis groups that access arrays. Thus, we throw out over 70% of the loops immediately³. Figure 5 presents the sorted weights of all the analysis groups in `nginx`. The distribution shows a quick drop after a few highly complex analysis groups. The long tail represents the numerous simple loops omnipresent in any code. 55.7% of the analysis groups score too low to be of interest. This means that *Dowser* needs to examine only the remaining 44.3%, i.e., 62 out of 140 analysis groups, or at most 12% of all loops. Out of these, the buffer overflow in Figure 1 ranks 4th.

6.1.2 Taint analysis in context of hunting for bugs

In Section 4 we mentioned that ‘traditional’ dynamic taint analysis misses implicit flows, i.e., flows that have

³In principle, if a loop accesses multiple arrays, it also contains multiple access groups. Thus, these 140 analysis groups are located in fewer than 140 loops.

no direct assignment of a tainted value to a variable. The problem turns out to be particularly serious for `nginx`. It receives input in text format, and transforms it to extract numerical values or various flags. As such code employs conditional statements, DTA misses the dependencies between the input and analysis groups.

Next, we evaluate the usefulness of field shifting. First, we implement the taint propagation exactly as proposed by Bao et al. [6], without any further restrictions. In that case, an index variable in the `nginx` parser becomes tainted, and we mark all HTTP fields succeeding the `uri` field as tainted as well. As a result, we introduce more symbolic data than necessary. Next, we apply field shifting (Section 4.2) which effectively limits taint propagation to just the `uri` field. In general, the field shifting optimization improves the accuracy of taint propagation in all applications that take multiple input fields whose order does not matter. On the other hand, it will not help if the order is fixed.

6.1.3 Importance of guiding symbolic execution

We now use the `nginx` example to assess the importance of guiding symbolic execution to a vulnerability condition. For `nginx`, the input message is a generic HTTP request. Since it exercises the vulnerable loop for this analysis group, its `uri` starts with “//”. Taint analysis allows us to detect that only the `uri` field is important, so we mark only this field as symbolic. As we shall see, without guidance, symbolic execution does not scale beyond very short `uri` fields (5-6 byte long). In contrast, *Dowser* successfully executes 50-byte-long symbolic `uris`.

When S2E [10] executes a loop, it can follow one of the two search strategies: depth-first search, or maximizing code coverage (as proposed in SAGE [18]). The first one aims at complete path coverage, and the second at executing basic blocks that were not seen before. However, none can be applied in practice to examine the complex loop in `nginx`. The search is so costly that we measured the runtime for only 5-6 byte long symbolic `uri` fields. The DFS strategy handled the 5-byte-long input in 139 seconds, the 6-byte-long in 824 seconds. A 7-byte input requires more than 1 hour to finish. Likewise, the code coverage strategy required 159, and 882 seconds, respectively. The code coverage heuristic does not speed up the search for buffer overflows either, since besides executing specific instructions from the loop, memory corruptions require a very particular execution context. Even if 100% code coverage is reached, they may stay undetected.

As we explained in Section 5, the strategy employed by *Dowser* does not aim at full coverage. Instead, it actively searches for paths which involve new pointer dereferences. The learning phase uses a 4-byte-long

symbolic input to observe access patterns in the loop. It follows a simple depth first search strategy. As the bug clearly cannot be triggered with this input size, the search continues in the second, hunting bugs, phase. The result of the learning phase disables 66% of the conditional branches significantly reducing the exponentially of the subsequent symbolic execution. Because of this heuristic, *Dowser* easily scales up to 50 symbolic bytes and finds the bug after just a few minutes. A 5-byte-long symbolic input is handled in 20 seconds, 10 bytes in 42 seconds, 20 bytes in 63 seconds, 30 in 146 seconds, 40 in 174 seconds and 50 in 253 seconds. These numbers maintain an exponential growth of 1.1 for each added character. Even though *Dowser* still exhibits the exponential behavior, the growth rate is fairly low. Even in the presence of 50 symbolic bytes, *Dowser* quickly finds the complex bug.

In practice, symbolic execution has problems dealing with real world applications and input sizes. The number of execution paths quickly overwhelms these systems. Since triggering buffer overflows not only requires a vulnerable basic block, but also a special context, traditional symbolic execution tools are ill suited. *Dowser*, instead, requires the application to be executed symbolically for only a very short input, and then it deals with real-world input sizes instead of being limited to a few input bytes. Combined with the ability to extract the relevant parts of the original input, this enables searching for bugs in applications like web servers where input sizes were considered until now to be well beyond the scalability of symbolic execution tools.

6.2 Overview

In this section, we consider several applications. First, we evaluate the dowsing mechanism, and we show that it successfully highlights vulnerable code fragments. Then, we summarize the memory corruptions detected by *Dowser*. They come from six real world applications of several tens of thousands LoC, including the `ffmpeg` videoplayer of 300K LoC. The bug in `ffmpeg`, and one of the bugs in `poppler` were not documented before.

6.2.1 Dowsing for candidate instructions

We now examine several aspects of the dowsing mechanism. First, we show that there is a correlation between *Dowser*’s scoring function and the existence of memory corruption vulnerabilities. Then, we discuss how our focus on complex loops limits the search space, i.e., the amount of analysis groups to be tested. We start with a description of our data set.

Data set To evaluate the effectiveness of *Dowser*, we chose six real world programs: `nginx`, `ffmpeg`,

Program	Vulnerability	Dowsing			Symbolic input	Symbolic execution		
		AG score	Loops	Loc		V-S2E	M-S2E	Dowser
nginx 0.6.32	CVE-2009-2629 heap underflow	4th out of 62/140 630 points	517	66k	URI field 50 bytes	> 8 h	> 8 h	253 sec
ffmpeg 0.5	UNKNOWN heap overread	3rd out of 727/1419 2186 points	1286	300k	Huffman table 224 bytes	> 8 h	> 8 h	48 sec
inspircd 1.1.22	CVE-2012-1836 heap overflow	1st out of 66/176 625 points	1750	45k	DNS response 301 bytes	200 sec	200 sec	32 sec
poppler 0.15.0	UNKNOWN heap overread	39th out of 388/904 1075 points	1737	120k	JPEG image 1024 bytes	> 8 h	> 8 h	14 sec
poppler 0.15.0	CVE-2010-3704 heap overflow	59th out of 388/904 910 points	1737	120k	Embedded font 1024 bytes	> 8 h	> 8 h	762 sec
libexif 0.6.20	CVE-2012-2841 heap overflow	8th out of 15/31 501 points	121	10k	EXIF tag/length 1024 + 4 bytes	> 8 h	652 sec	652 sec
libexif 0.6.20	CVE-2012-2840 off-by-one error	15th out of 15/31 40 points	121	10k	EXIF tag/length 1024 + 4 bytes	> 8 h	347 sec	347 sec
libexif 0.6.20	CVE-2012-2813 heap overflow	15th out of 15/31 40 points	121	10k	EXIF tag/length 1024 + 4 bytes	> 8 h	277 sec	277 sec
snort 2.4.0	CVE-2005-3252 stack overflow	24th out of 60/174 246 points	616	75k	UDP packet 1100 bytes	> 8 h	> 8 h	617 sec

Table 2: Applications tested with *Dowser*. The *Dowsing* section presents the results of *Dowser*’s ranking scheme. *AG score* is the complexity of the vulnerable analysis group - its position among other analysis groups; X/Y denotes all analysis groups that are “complex enough” to be potentially analyzed/all analysis groups which access arrays; and the number of points it scores. *Loops* counts outermost loops in the whole program, and *Loc* - the lines of code according to *sloccount*. *Symbolic input* specifies how many and which parts of the input were determined to be marked as symbolic by the first two components of *Dowser*. The last section shows symbolic execution times until revealing the bug. Almost all applications proved to be too complex for the vanilla version of S2E (V-S2E). *Magic S2E* (M-S2E) is the time S2E takes to find the bug when we feed it with an input with only a minimal symbolic part (as identified in Symbolic input). Finally, the last column is the execution time of fully-fledged *Dowser*.

`inspircd`, `libexif`, `poppler`, and `snort`. Additionally, we consider the vulnerabilities in `sendmail` tested by Zitser et al. [45]. For these applications, we analyzed all buffer overflows reported in CVE [26] since 2009. For `ffmpeg`, rather than include all possible codecs, we just picked the ones for which we had test cases. Out of 27 CVE reports, we took 17 for the evaluation. The remaining ten vulnerabilities are out of the scope of this paper – nine of them are related to an erroneous usage of a correct function, e.g., `strcpy`, and one was not in a loop. In this section, we consider the analysis groups from all the applications together, giving us over 3000 samples, 17 of which are known to be vulnerable⁴.

When evaluating *Dowser*’s scoring mechanism, we also compare it to a straightforward scoring function that treats all instructions uniformly. For each array access, it considers exactly the same AGs as *Dowser*. However, instead of the scoring algorithm (Table 1), each instruction gets 10 points. We will refer to this metric as `count`.

Correlation For both *Dowser*’s and the `count` scoring functions, we computed the correlation between the number of points assigned to an analysis group and the existence of a memory corruption vulnerability. We used

the Spearman rank correlation [2], since it is a reliable measure that is appropriate even when we do not know the probability distribution of the variables, or when the association between the variables is non-linear.

The positive correlation for *Dowser* is statistically significant at $p < 0.0001$, for `count` — at $p < 0.005$. The correlation for *Dowser* is stronger.

Dowsing The *Dowsing* columns of Table 2 shows that our focus on complex loops limits the search space from thousands of LoC to hundreds of loops, and finally to a small number of “interesting” analysis groups. Observe that `ffmpeg` has more analysis groups than loops. That is correct. If a loop accesses multiple arrays, it contains multiple analysis groups.

By limiting the analysis to complex cases, we focus on a smaller fraction of all AGs in the program, e.g., we consider 36.9% of all the analysis groups in `inspircd`, and 34.5% in `snort`. `ffmpeg`, on the other hand, contains lots of complex loops that decode videos, so we also observe many “complex” analysis groups.

In practice, symbolic execution, guided or not is expensive, and we can hardly afford a thorough analysis of more than just a small fraction of the target AGs of an application, say 20%-30%. For this reason, *Dowser* uses a scoring function, and tests the analysis groups in order of

⁴Since the scoring functions are application agnostic, it is sound to compare their results across applications.

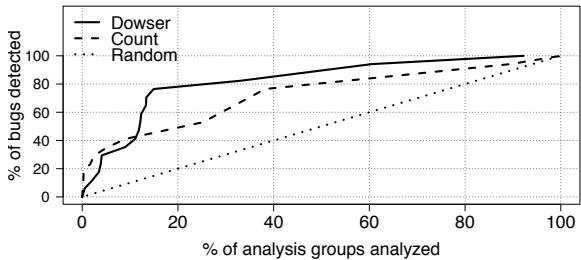


Fig. 6: A comparison of random testing and two scoring functions: *Dowser*'s and `count`. It illustrates how many bugs we detect if we test a particular fraction of the analysis groups.

decreasing score. Specifically, *Dowser* looks at complexity. However, alternative heuristics are also possible. For instance, one may count the instructions that influence array accesses in an AG. To evaluate whether *Dowser*'s heuristics are useful, we compare how many bugs we discover if we examine increasing fractions of all AGs, in descending order of the score. So, we determine how many of the bugs we find if we explore the top 10% of all AGs, how many bugs we find when we explore the top 20%, and so on. In our evaluation, we are comparing the following ranking functions: (1) *Dowser*'s complexity metric, (2) counting instructions as described above, and (3) random.

Figure 6 illustrates the results. The random ranking serves as a baseline—clearly both `count` and *Dowser* perform better. In order to detect all 17 bugs, *Dowser* has to analyze 92.2% of all the analysis groups. However, even with just 15% of the targets, we find almost 80% (13/17) of all the bugs. At that same fraction of targets, `count` finds a little over 40% of the bugs (7/17). Overall, *Dowser* outperforms `count` beyond the 10% in the ranking. It also reaches the 100% bug score earlier than the alternatives, although the difference is minimal.

The reason why *Dowser* still requires 92% of the AGs to find *all* bugs, is that some of the bugs were very simple. The “simplest” cases include a trivial buffer overflow in `poppler` (worth 16 points), and two vulnerabilities in `sendmail` from 1999 (worth 20 points each). Since *Dowser* is designed to prioritize complex array accesses, these buffer overflows end up in the low scoring group. (The “simple” analysis groups – with less than 26 points – start at 47.9%). Clearly, both heuristics provide much better results than random sampling. Except for the tail, they find the bugs significantly quicker, which proves their usefulness.

To summarize, we have shown that a testing strategy based on *Dowser*'s scoring function is effective. It lets us find vulnerabilities quicker than random testing or a

scoring function based on the length of an analysis group.

6.2.2 Symbolic execution

Table 2 presents attacks detected by *Dowser*. The last section shows how long it takes before symbolic execution detects the bug. Since the vanilla version of S2E cannot handle these applications with the whole input marked as symbolic, we also run the experiments with minimal symbolic inputs (“Magic S2E”). It represents the best-case scenario when an all-knowing oracle tells the execution engine exactly which bytes it should make symbolic. Finally, we present *Dowser*'s execution times.

We run S2E for as short a time as possible, e.g., a single request/response in `nginx` and transcoding a single frame in `ffmpeg`. Still, in most applications, vanilla S2E fails to find bugs in a reasonable amount of time. `inspircd` is an exception, but in this case we explicitly tested the vulnerable DNS resolver only. In the case of `libexif`, we can see no difference between “Magic S2E” and *Dowser*, so *Dowser*'s guidance did not influence the results. The reason is that our test suite here was simple, and the execution paths reached the vulnerability condition quickly. In contrast, more complex applications process the inputs intensively, moving symbolic execution away from the code of interest. In all these cases, *Dowser* finds bugs significantly faster. Even if we take the 15 minute tests of higher-ranking analysis groups into account, *Dowser* provides a considerable improvement over existing systems.

7 Related work

Dowser is a ‘guided’ fuzzer which draws on knowledge from multiple domains. In this section, we place our system in the context of existing approaches. We start with the scoring function and selection of code fragments. Next, we discuss traditional fuzzing. We then review previous work on dynamic taint analysis in fuzzing, and finally, discuss existing work on whitebox fuzzing and symbolic execution.

Software complexity metrics Many studies have shown that software complexity metrics are positively correlated with defect density or security vulnerabilities [29, 35, 16, 44, 35, 32]. However, Nagappan et al. [29] argued that no single set of metrics fits all projects, while Zimmermann et al. [44] emphasize a need for metrics that exploit the unique characteristics of vulnerabilities, e.g., buffer overflows or integer overruns. All these approaches consider the broad class of post-release defects or security vulnerabilities, and consider a very generic set of measurements, e.g., the number of basic blocks in a function’s control flow graph, the number of global or local variables read or written, the maximum nesting level

of `if` or `while` statements and so on. *Dowser* is very different in this respect, and to the best of our knowledge, the first of its kind. We focus on a narrow group of security vulnerabilities, i.e., buffer overflows, so our scoring function is tailored to reflect the complexity of pointer manipulation instructions.

Traditional fuzzing Software fuzzing started in earnest in the 90s when Miller et al. [25] described how they fed random inputs to (UNIX) utilities, and managed to crash 25–33% of the target programs. More advanced fuzzers along the same lines, like Spike [39], and SNOOZE [5], deliberately generate malformed inputs, while later fuzzers that aim for deeper bugs are often based on the input grammar (e.g., Kaksonen [20] and [40]). DeMott [13] offers a survey of fuzz testing tools. As observed by Godefroid et al. [18], traditional fuzzers are useful, but typically find only shallow bugs.

Application of DTA to fuzzing BuzzFuzz [15] uses DTA to locate regions of seed input files that influence values used at library calls. They specifically select library calls, as they are often developed by different people than the author of the calling program and often lack a perfect description of the API. Buzzfuzz does not use symbolic execution at all, but uses DTA only to ensure that they preserve the right input format. Unlike *Dowser*, it ignores implicit flows completely, so it could never find bugs such as the one in nginx (Figure 1). In addition, *Dowser* is more selective in the application of DTA. It’s difficult to assess which library calls are important and require a closer inspection, while *Dowser* explicitly selects complex code fragments.

TaintScope [42] is similar in that it also uses DTA to select fields of the input seed which influence security-sensitive points (e.g., system/library calls). In addition, TaintScope is capable of identifying and bypassing checksum checks. Like Buzzfuzz, it differs from *Dowser* in that it ignores implicit flows and assumes only that library calls are the interesting points. Unlike BuzzFuzz, TaintScope operates at the binary level, rather than the source.

Symbolic-execution-based fuzzing Recently, there has been much interest in whitebox fuzzing, symbolic execution, concolic execution, and constraint solving. Examples include EXE [8], KLEE [7], CUTE [33], DART [17], SAGE [18], and the work by Moser et al. [28]. Microsoft’s SAGE, for instance, starts with a well-formed input and symbolically executes the program under test in attempt to sweep through all feasible execution paths of the program. While doing so, it checks security properties using AppVerifier. All of these systems substitute (some of the) program inputs with symbolic values, gather input constraints on a program trace, and generate new input that exercises differ-

ent paths in the program. They are very powerful, and can analyze programs in detail, but it is difficult to make them scale (especially if you want to explore many loop-based array accesses). The problem is that the number of paths grows very quickly.

Zesti [24] takes a different approach and executes existing regression tests symbolically. Intuitively, it checks whether they can trigger a vulnerable condition by slightly modifying the test input. This technique scales better and is useful for finding bugs in paths in the neighborhood of existing test suites. It is not suitable for bugs that are far from these paths. As an example, a generic input which exercises the vulnerable loop in Figure 1 has the `uri` of the form “`//{arbitrary characters}`”, and the shortest input triggering the bug is “`//.../`”. When fed with “`//abc`”, [24] does not find the bug—because it was not designed for this scenario. Instead, it requires an input which is much closer to the vulnerability condition, e.g., “`//...{an arbitrary character}`”. For *Dowser*, the generic input is sufficient.

SmartFuzz [27] focuses on integer bugs. It uses symbolic execution to construct test cases that trigger arithmetic overflows, non-value-preserving width conversions, or dangerous signed/unsigned conversions. In contrast, *Dowser* targets the more common (and harder to find) case of buffer overflows. Finally, Babić et al. [4] guide symbolic execution to potentially vulnerable program points detected with static analysis. However, the interprocedural context- and flow-sensitive static analysis proposed does not scale well to real world programs and the experimental results contain only short traces.

8 Conclusion

Dowser is a guided fuzzer that combines static analysis, dynamic taint analysis, and symbolic execution to find buffer overflow vulnerabilities deep in a program’s logic. It starts by determining ‘interesting’ array accesses, i.e., accesses that are most likely to harbor buffer overflows. It ranks these accesses in order of complexity—allowing security experts to focus on complex bugs, if so desired. Next, it uses taint analysis to determine which inputs influence these array accesses and fuzzes only these bytes. Specifically, it makes (only) these bytes symbolic in the subsequent symbolic execution. Where possible *Dowser*’s symbolic execution engine selects paths that are most likely to lead to overflows. Each three of the steps contain novel contributions in and of themselves (e.g., the ranking of array accesses, the implicit flow handling in taint analysis, and the symbolic execution based on pointer value coverage), but the overall contribution is a new, practical and complete fuzzing approach that scales to real applications and complex bugs that would be hard or impossible to find with existing tech-

niques. Moreover, *Dowser* proposes a novel ‘spot-check’ approach to finding buffer overflows in real software.

Acknowledgment

This work is supported by the European Research Council through project ERC-2010-StG 259108-ROSETTA, the EU FP7 SysSec Network of Excellence and by the Microsoft Research PhD Scholarship Programme through the project MRL 2011-049. The authors would like to thank Bartek Knapik for his help in designing the statistical evaluation.

References

- [1] CVE-2009-2629: Buffer underflow vulnerability in nginx. <http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2009-2629>, 2009.
- [2] ACZEL, A. D., AND SOUNDERPANDIAN, J. *Complete Business Statistics*, sixth ed. McGraw-Hill, 2006.
- [3] AKRITIDIS, P., CADAR, C., RAICIU, C., COSTA, M., AND CASTRO, M. Preventing memory error exploits with WIT. In *Proceedings of the 2008 IEEE Symposium on Security and Privacy* (2008), S&P’08.
- [4] BABIĆ, D., MARTIGNONI, L., MCCAMANT, S., AND SONG, D. Statically-directed dynamic automated test generation. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis* (2011), ISSTA’11.
- [5] BANKS, G., COVA, M., FELMETSGER, V., ALMEROOTH, K., KEMMERER, R., AND VIGNA, G. SNOOZE: toward a stateful network protocol fuzzZEr. In *Proceedings of the 9th international conference on Information Security* (2006), ISC’06.
- [6] BAO, T., ZHENG, Y., LIN, Z., ZHANG, X., AND XU, D. Strict control dependence and its effect on dynamic information flow analyses. In *Proceedings of the 19th International Symposium on Software testing and analysis* (2010), ISSTA’10.
- [7] CADAR, C., DUNBAR, D., AND ENGLER, D. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation* (2008), OSDI’08.
- [8] CADAR, C., GANESH, V., PAWLOWSKI, P. M., DILL, D. L., AND ENGLER, D. R. EXE: Automatically generating inputs of death. In *CCS ’06: Proceedings of the 13th ACM conference on Computer and communications security* (2006).
- [9] CAVALLARO, L., SAXENA, P., AND SEKAR, R. On the Limits of Information Flow Techniques for Malware Analysis and Containment. In *Proceedings of the Fifth Conference on Detection of Intrusions and Malware & Vulnerability Assessment* (2008), DIMVA’08.
- [10] CHIPOUNOV, V., KUZNETSOV, V., AND CANDEA, G. S2E: A platform for in vivo multi-path analysis of software systems. In *Proceedings of the 16th Intl. Conference on Architectural Support for Programming Languages and Operating Systems* (2011), AS-PLOS’11.
- [11] COWAN, C., PU, C., MAIER, D., HINTONY, H., WALPOLE, J., BAKKE, P., BEATTIE, S., GRIER, A., WAGLE, P., AND ZHANG, Q. StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks. In *Proceedings of the 7th USENIX Security Symposium* (1998), SSYM’98.
- [12] CWE/SANS. CWE/SANS TOP 25 Most Dangerous Software Errors. www.sans.org/top25-software-errors, 2011.
- [13] DEMOTT, J. The evolving art of fuzzng. DEFCON 14, http://www.appliedsec.com/files/The_Evolving_Art_of_Fuzzing.pdf, 2005.
- [14] FERRANTE, J., OTTENSTEIN, K. J., AND WARREN, J. D. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.* 9 (1997), 319–349.
- [15] GANESH, V., LEEK, T., AND RINARD, M. Taint-based directed whitebox fuzzing. In *Proceedings of the 31st International Conference on Software Engineering* (2009), ICSE’09.
- [16] GEGICK, M., WILLIAMS, L., OSBORNE, J., AND VOUK, M. Prioritizing software security fortification through code-level metrics. In *Proc. of the 4th ACM workshop on Quality of protection* (Oct. 2008), QoP’08, ACM Press.
- [17] GODEFROID, P., KLARLUND, N., AND SEN, K. DART: directed automated random testing. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation* (2005), PLDI’05.
- [18] GODEFROID, P., LEVIN, M. Y., AND MOLNAR, D. A. Automated Whitebox Fuzz Testing. In *Proceedings of the 15th Annual Network and Distributed System Security Symposium* (2008), NDSS’08.
- [19] GODEFROID, P., AND LUCHAUP, D. Automatic partial loop summarization in dynamic test generation. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis* (2011), ISSTA’11.
- [20] KAKSONEN, R. A functional method for assessing protocol implementation security. Tech. Rep. 448, VTT, 2001.
- [21] KANG, M. G., MCCAMANT, S., POOSANKAM, P., AND SONG, D. DTA++: Dynamic taint analysis with targeted control-flow propagation. In *Proceedings of the 18th Annual Network and Distributed System Security Symposium* (2011), NDSS’11.
- [22] KHURSHID, S., PÄSÄREANU, C. S., AND VISSER, W. Generalized symbolic execution for model checking and testing. In *Proceedings of the 9th international conference on Tools and algorithms for the construction and analysis of systems* (2003), TACAS’03.
- [23] LATTNER, C., AND ADVE, V. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the international symposium on Code generation and optimization* (2004), CGO’04.
- [24] MARINESCU, P. D., AND CADAR, C. make test-zesti: a symbolic execution solution for improving regression testing. In *Proc. of the 2012 International Conference on Software Engineering* (June 2012), ICSE’12, pp. 716–726.
- [25] MILLER, B. P., FREDRIKSEN, L., AND SO, B. An empirical study of the reliability of UNIX utilities. *Commun. ACM* 33 (Dec 1990), 32–44.
- [26] MITRE. Common Vulnerabilities and Exposures (CVE). <http://cve.mitre.org/>, 2011.
- [27] MOLNAR, D., LI, X. C., AND WAGNER, D. A. Dynamic test generation to find integer bugs in x86 binary linux programs. In *Proceedings of the 18th conference on USENIX security symposium* (2009), SSYM’09.
- [28] MOSER, A., KRUEGEL, C., AND KIRDA, E. Exploring multiple execution paths for malware analysis. In *Proceedings of the 2007 IEEE Symposium on Security and Privacy* (2007), SP’07, IEEE Computer Society.
- [29] NAGAPPAN, N., BALL, T., AND ZELLER, A. Mining metrics to predict component failures. In *Proceedings of the 28th international conference on Software engineering* (2006), ICSE’06.

- [30] NETHERCOTE, N., AND SEWARD, J. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. In *Proceedings of the Third International ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environments* (2007), VEE'07.
- [31] NEWSOME, J., AND SONG, D. Dynamic taint analysis: Automatic detection, analysis, and signature generation of exploit attacks on commodity software. In *Proceedings of the Network and Distributed Systems Security Symposium* (2005), NDSS'05.
- [32] NGUYEN, V. H., AND TRAN, L. M. S. Predicting vulnerable software components with dependency graphs. In *Proc. of the 6th International Workshop on Security Measurements and Metrics* (Sept. 2010), MetriSec'10, ACM Press.
- [33] SEN, K., MARINOV, D., AND AGHA, G. CUTE: a concolic unit testing engine for C. In *Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering* (2005), ESEC/FSE-13.
- [34] SEREBRYANY, K., BRUENING, D., POTAPENKO, A., AND VYUKOV, D. AddressSanitizer: A fast address sanity checker. In *Proceedings of USENIX Annual Technical Conference* (2012).
- [35] SHIN, Y., AND WILLIAMS, L. An initial study on the use of execution complexity metrics as indicators of software vulnerabilities. In *Proceedings of the 7th International Workshop on Software Engineering for Secure Systems* (2011), SESS'11.
- [36] SLOWINSKA, A., AND BOS, H. Pointless tainting?: evaluating the practicality of pointer tainting. In *EuroSys '09: Proceedings of the 4th ACM European conference on Computer systems* (2009).
- [37] SLOWINSKA, A., STANCESCU, T., AND BOS, H. Body Armor for Binaries: preventing buffer overflows without recompilation. In *Proceedings of USENIX Annual Technical Conference* (2012).
- [38] SOTIROV, A. Modern exploitation and memory protection bypasses. USENIX Security invited talk, <http://www.usenix.org/events/sec09/tech/slides/sotirov.pdf>, 2009.
- [39] SPIKE. <http://www.immunitysec.com/resources-freesoftware.shtml>.
- [40] SUTTON, M., GREENE, A., AND AMINI, P. *Fuzzing: Brute Force Vulnerability Discovery*. Addison-Wesley Professional, 2007.
- [41] VAN DER VEEN, V., DUTT-SHARMA, N., CAVALLARO, L., AND BOS, H. Memory Errors: The Past, the Present, and the Future. In *Proceedings of The 15th International Symposium on Research in Attacks, Intrusions and Defenses* (2012), RAID'12.
- [42] WANG, T., WEI, T., GU, G., AND ZOU, W. TaintScope: A Checksum-Aware Directed Fuzzing Tool for Automatic Software Vulnerability Detection. In *Proceedings of the 31st IEEE Symposium on Security and Privacy* (2010), SP'10.
- [43] WILLIAMS, N., MARRE, B., AND MOUY, P. On-the-Fly Generation of K-Path Tests for C Functions. In *Proceedings of the 19th IEEE international conference on Automated software engineering* (2004), ASE'04.
- [44] ZIMMERMANN, T., NAGAPPAN, N., AND WILLIAMS, L. Searching for a Needle in a Haystack: Predicting Security Vulnerabilities for Windows Vista. In *Proc. of the 3rd International Conference on Software Testing, Verification and Validation* (Apr. 2010), ICST'10.
- [45] ZITSER, M., LIPPmann, R., AND LEEK, T. Testing static analysis tools using exploitable buffer overflows from open source code. In *Proc. of the 12th ACM SIGSOFT twelfth international symposium on Foundations of software engineering* (Nov. 2004), SIGSOFT '04/FSE-12.

MetaSymploit: Day-One Defense Against Script-based Attacks with Security-Enhanced Symbolic Analysis

Ruowen Wang, Peng Ning, Tao Xie, Quan Chen

Department of Computer Science

North Carolina State University

Raleigh, NC, USA

{*rwang9, pning, qchen10*}@ncsu.edu, *xie@csc.ncsu.edu*

Abstract

A script-based attack framework is a new type of cyber-attack tool written in scripting languages. It carries various attack scripts targeting vulnerabilities across different systems. It also supports fast development of new attack scripts that can even exploit zero-day vulnerabilities. Such mechanisms pose a big challenge to the defense side since traditional malware analysis cannot catch up with the emerging speed of new attack scripts. In this paper, we propose MetaSymploit, the first system of fast attack script analysis and automatic signature generation for a network Intrusion Detection System (IDS). As soon as a new attack script is developed and distributed, MetaSymploit uses security-enhanced symbolic execution to quickly analyze the script and automatically generate specific IDS signatures to defend against all possible attacks launched by this new script from Day One. We implement a prototype of MetaSymploit targeting Metasploit, the most popular penetration framework. In the experiments on 45 real attack scripts, MetaSymploit automatically generates Snort IDS rules as signatures that effectively detect the attacks launched by the 45 scripts. Furthermore, the results show that MetaSymploit substantially complements and improves existing Snort rules that are manually written by the official Snort team.

1 Introduction

Over the years, with rapid evolution of attacking techniques, script-based attack frameworks have emerged and become a new threat [2, 3, 6, 39]. A script-based attack framework is an attack-launching platform written in scripting languages, such as Ruby and Python. Such framework carries various attack scripts, each of which exploits one or more vulnerabilities of a specific application across multiple versions. With the high productivity of using scripting languages, attackers can easily develop new attack scripts to exploit new vulnerabilities.

To launch an attack, an attacker runs an attack script on the framework remotely. By probing a vulnerable target over the network, the attack script dynamically composes an attack payload, and sends the payload to the target to exploit the vulnerability. The attack framework also provides many built-in components with APIs of various attack functionalities to support rapid development of new attack scripts. Once a zero-day vulnerability is found, a new attack script can be quickly developed and distributed in hacking communities, where other attackers even script kiddies can directly download the new script to launch attacks exploiting the zero-day vulnerability.

A well-known example of the script-based attack frameworks is Metasploit [3], the most popular Ruby-based penetration framework. It has more than 700 attack scripts targeting various vulnerable applications on different operating systems (OSes). It also provides built-in components for creating new attack scripts. Metasploit was originally developed for penetration testing using proof-of-concept scripts. But with years of improvements, it has become a full-fledged attack framework. Unfortunately, as an open source project, Metasploit can be easily obtained and used by attackers for illegal purposes. For example, it was reported that the well-known worm “Conficker” used a payload generated by Metasploit to spread [5]. A Metasploit attack script was immediately distributed after a zero-day vulnerability was found in Java 7 [32]. A four-year empirical study shows real malicious network traffic related to Metasploit on a worldwide scale. Moreover, the study shows that many Metasploit attack scripts are used by attackers almost immediately after the scripts are distributed in hacking communities [33].

When a new attack script is distributed and captured by security vendors, the traditional approach to defend against it is to first set up a controlled environment with a vulnerable application installed. Then security analysts repeatedly run the script to exploit the environment over a monitored network, collecting a large number of at-

tack payload samples, and finally extract common patterns from the samples to generate IDS signatures.

However, with the attack framework, new attack scripts can be quickly developed and distributed to exploit the latest vulnerabilities. This poses a great challenge that the traditional approach can hardly catch up with the release speed of new attacks, due to the time-consuming process of setting up test environments and analyzing attack payload samples. In our evaluation (Section 5), we observe that even the latest Snort IDS rules written by security analysts cannot detect many Metasploit-based attacks.

In this paper, we propose *MetaSymploit*, the first system of fast attack script analysis and automatic IDS signature generation. As soon as a new attack script is distributed, MetaSymploit quickly analyzes the attack script and automatically generates IDS signatures of its attack payloads, thereby providing defense against new attacks launched by this script from Day One. Particularly, MetaSymploit gives the *first aid* to zero-day vulnerabilities whose security patches are not available while the attack scripts that exploit them are already distributed.

Specifically, MetaSymploit leverages symbolic execution while enhancing it with several security features designed for attack script analysis and signature generation. By treating environment-dependent values as symbolic values, MetaSymploit symbolically executes attack scripts without interacting with actual environments or vulnerable applications, thus substantially reducing the time and cost of the analysis. With path exploration of symbolic execution, MetaSymploit also explores different execution paths in an attack script, exposing different attack behaviors and payloads that the script produces under different attack conditions.

To generate signatures of attack payloads, instead of analyzing large volumes of payload samples, MetaSymploit keeps track of the payload composing process in the attack script during symbolic execution. MetaSymploit uses symbolic values to represent variant contents in a payload (e.g., random paddings), in order to distinguish constant contents (e.g., vulnerability-trigger bytes) from variant ones. When the script sends a composed payload to launch an attack, MetaSymploit captures the payload’s entire contents, extracts constant contents as patterns and generates a signature specific to this payload.

In a case study, we implement a security-enhanced symbolic execution engine for Ruby, develop MetaSymploit as a practical tool targeting Metasploit, and generate Snort rules as IDS signatures. Particularly, instead of heavily modifying the script interpreters, we design a lightweight symbolic execution engine running on unmodified interpreters. This lightweight design can keep pace with the continuous upgrades of the language syntax and interpreter (e.g., Ruby 1.8/1.9/2.0). Therefore,

our design supports analyzing attack scripts written in different versions of the scripting language.

We evaluate MetaSymploit using real-world attack scripts. We assess our automatically generated Snort rules by launching attacks using 45 real-world Metasploit attack scripts from `exploit-db.com`, including one that exploits a zero-day vulnerability in Java 7. Our rules successfully detect the attack payloads launched by the 45 scripts. Furthermore, we also compare our rules with the official Snort rule set written by security analysts, and have three findings: (1) the official rule set is incomplete and 23 of the 45 attack scripts are not covered by the official rule set; (2) for the scripts covered by the official rules, our rules share similar but more specific patterns with the official ones; (3) our studies also expose 3 deficient official rules that fail to detect Metasploit attacks. Therefore, MetaSymploit is a helpful complement to improve the completeness and accuracy of existing IDS signatures to defend against attack scripts.

In summary, we make three major contributions:

1. We point out the security issues of script-based attacks, and propose a scalable approach called MetaSymploit that uses security-enhanced symbolic execution to automatically analyze attack scripts and generate IDS signatures for defense.
2. We implement a security-enhanced symbolic execution engine for Ruby and develop a practical tool for the popular Metasploit attack framework. Our tool can generate Snort rules to defend against newly distributed Metasploit attack scripts from Day One.
3. We demonstrate the effectiveness of MetaSymploit using recent Metasploit attack scripts in real-world attack environments, and also show that MetaSymploit can complement and improve existing manually-written IDS signatures.

2 Background

We first give the background of how an attack script works. Generally, when an attack script runs on top of an attack framework, the script performs four major steps to launch an attack. (1) The script probes the version and runtime environment of the vulnerable target over the network. (2) Based on the probing result and the script’s own hard-coded knowledge base, the script identifies the specific vulnerability existing in this target. The knowledge base is usually a list containing the information (e.g., vulnerable return addresses) of all targets that this script can attack. (3) Then the script dynamically composes an attack payload customized for this target. (4) Finally, the script sends the payload to the target to exploit the vulnerability.

```

1 def exploit
2   connect()
3   preamble = "\x00\x4d\x00\x03\x00\x01"
4   version = probe_ver()
5   if version == 5
6     payload = prep_ark5()
7   else
8     payload = prep_ark4()
9   end
10  preamble << payload.length
11  sock.put(preamble) # Required by protocol
12  sock.get_once()
13  sock.put(payload) # Send attack payload
14  sock.get_once()
15  ... # vulnerability triggered
16 end
17 def prep_ark5()
18   payload = shellcode()
19   payload << rand_alpha(1167 -
20     payload.length)
21   payload << "\xe9" + [-1172].pack("V")
22   payload << "\xeb\xf9"
23   payload << get_target_ret(5) # Tar_Ver: 5
24   payload << rand_alpha(4096 -
25     payload.length)
26   return payload
27 end

```

Listing 1: The code snippet from a real Metasploit attack script *type77.rb* [4] (slightly modified for better presentation)

Depending on the attack strategy and vulnerability type, different scripts may have different attack behaviors when performing these steps. For example, a brute-force attack may keep composing and sending payloads with guessed values until the target is compromised, while a stealthy attack may carefully clean up the trace in the target’s log after sending the payload.

Among these steps, composing and sending an attack payload are the key steps of launching an attack. An attack payload is typically a string of bytes composed with four elements: (a) special and fixed bytes that can exploit a specific vulnerability; (b) an arbitrary shellcode that attackers choose to execute after the vulnerability is exploited. The shellcode content is usually variant, especially when obfuscated; (c) random or special paddings (e.g., NOP 0x90) that make the payload more robust; (d) other format bytes required by network protocols.

With the help of the rich libraries of scripting languages and the built-in components provided by the attack framework, an attack script can call APIs of related libraries or components to help it perform each step, especially composing an attack payload.

As an example, Listing 1 shows a Ruby code snippet extracted from a real Metasploit attack script exploiting a vulnerable application called Arkeia. In the example, the script defines two methods. `exploit` is the main method that performs the major steps to launch the attack. `prep_ark5` is one of the payload composing methods. When the script runs on Metasploit, it first

```

alert tcp any any -> any 617 (
msg:"Script: type77 (Win), Target Version: 5,
  Behavior: Version Probing, Stack Overflow,
  Pattern: JMP to Shellcode with
    Vul_Ret_Addr";
content:
"\xe9 38 6c fb ff ff eb f9 ad 32 aa 71|";
pcre:"/[.] {1167} \xe9{38}{6c}{fb}{ff}{eb}{f9}{ad}{32}{aa}{71}[a-zA-Z]{2917}/";
classtype:shellcode-detect; sid:5000656;

```

Listing 2: One Snort rule signature generated for the attack payload composed by `prep_ark5`.

connects to the target over the network (Line 2), and then probes the target’s version (Line 4). Here both `connect` and `probe_ver` are API methods of a built-in network protocol component. Based on the version, it calls the corresponding method to start composing the attack payload specific to the target (Lines 5-9).

When `prep_ark5` is called, the payload is first assigned by the shellcode component, which returns a configured shellcode (Line 18). Note that the shellcode can be freely chosen and obfuscated. The shellcode component offers several different shellcodes for different purposes. Then the payload is appended (<<) with several contents (Lines 19-23). `rand_alpha` generates random alphabet padding to not only extend the payload to the required size of the network protocol, but also introduce more randomness for evasion. The concrete bytes represent some assembly code that will jump to the shellcode (e.g., “\xeb\xf9” and “\xe9” are two JMP instructions). `pack ("V")` converts the integer to bytes as the offset of one JMP. `get_target_ret` is another attack framework API that queries the script’s knowledge base (omitted here due to space limit, please refer to [4]) to retrieve the exploitable return address based on the target version, which can hijack the control flow¹ (Line 22). After the payload is composed, the script first sends a preamble packet to the target, followed by the attack payload packet to exploit the vulnerability (Lines 11-13).

Popular attack frameworks provide plenty of built-in components covering various network protocols, OSes, and offering different shellcodes and NOP paddings, which enable attackers to quickly develop new attack scripts to exploit different targets. Furthermore, advanced attackers can create even sophisticated attack scripts, which have multiple execution paths performing different attack behaviors and payloads. Some of them may be triggered only under certain attack conditions.

Therefore, the traditional approach that requires both controlled environments and vulnerable applications is not scalable for analyzing attack scripts. Since differ-

¹In [4], the exploitable return address actually points to a POP/POP/RET instruction sequence, which is a typical SEH-based attack to hijack control flow in Windows.

ent attack scripts target different applications and OSes, it is costly and time-consuming to obtain every application (let alone the expensive commercial ones) and set up environments for every OS. It is even harder to create different attack conditions to expose different attack behaviors and payloads in sophisticated attack scripts.

3 MetaSymploit

In this section, we first state the problem and assumptions we focus on, and then give an overview of MetaSymploit, followed by the detailed techniques in its two core parts.

3.1 Problem Statement and Assumptions

Problem Statement. We focus on the problem caused by script-based attack frameworks and their attack scripts: how to provide an automated mechanism that can analyze and defend against newly distributed attack scripts. Particularly, the mechanism should be time-efficient in order to address the security issues caused by two major features of attack scripts: a large number of scripts with wide-ranging targets, and fast development and distribution of new scripts that can be directly used to exploit zero-day vulnerabilities.

Assumptions. We assume that both script-based attack frameworks and attack scripts are available from either public or underground hacking communities. As soon as a new attack script is distributed, it can be immediately captured and analyzed. We also assume that the scripting languages used by attack frameworks are general-purpose object-oriented scripting languages, such as Ruby and Python. In reality, sectools.org lists 11 most popular attack tools [6] in the public community. 8 of them are Ruby/Python-based attack frameworks. Most of them are actively maintained with frequent updates of new attack scripts.

3.2 MetaSymploit Overview

Given an attack script, the goal of MetaSymploit is to quickly analyze fine-grained attack behaviors that the script can perform, and automatically generate specific IDS signatures for every attack payload that the script can compose, providing a fast and effective defense against attacks launched by this script. To achieve this goal, MetaSymploit leverages symbolic execution and enhances it with a number of security features designed for attack scripts analysis and signature generation.

Symbolic execution² is a program analysis technique that executes programs with symbolic rather than concrete values. When executing branches related to sym-

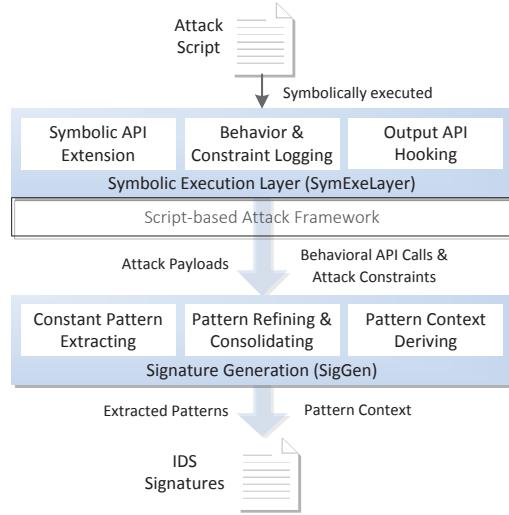


Figure 1: MetaSymploit consists of two major parts drawn in grey. (The arrows show the workflow of an attack script analysis.)

bolic values, it maintains a path constraint set and forks to explore different execution paths. By using symbolic execution, MetaSymploit has three advantages to achieve fast analysis and defense against attack scripts: (1) analyzing scripts without requiring actual environments or vulnerable targets, (2) exploring different execution paths to expose different attack behaviors, (3) using symbolic values to represent variant contents in attack payloads to ease the extraction of constant patterns.

Figure 1 shows the architecture of MetaSymploit, which consists of two major parts, the symbolic execution layer (SymExeLayer) and the signature generator (SigGen). Given an attack framework, SymExeLayer is built upon the framework. It reuses the framework's execution facility while extending the framework interface to support symbolic execution of attack scripts. When a script is symbolically executed, SymExeLayer captures all attack behaviors and payloads that the script can perform and compose. After the symbolic execution is done, SigGen takes the captured results as inputs. It extracts constant patterns by parsing the contents of the attack payloads. It also analyzes the attack behaviors to derive the semantic contexts that describe the extracted patterns. Finally, SigGen combines the patterns and the contexts to generate IDS signatures for this attack script.

More specifically, three key techniques are developed to realize the functionalities of SymExeLayer and SigGen, respectively. As shown in Figure 1, SymExeLayer consists of (1) *Symbolic API Extension*. It extends the APIs of both the attack framework and the scripting language to support symbolic values and operations. Notably, it extends the APIs related to environments/tar-

²For more background of symbolic execution, please refer to [25]

gets and variant payload contents to return symbolic values. (2) *Behavioral API & Attack Constraint Logging*. It records critical API calls that represent attack behaviors. It also logs path constraints of symbolic values related to environments and targets. Both logs will be used for deriving pattern context (described later). (3) *Output API Hooking*. It hooks various output APIs that are used to send attack payloads, in order to capture complete payload contents for extracting constant patterns.

SigGen consists of (1) *Constant Pattern Extracting*. By parsing the payload contents, it extracts constant patterns that can represent the payload. Constant patterns include fixed contents, fixed lengths of contents, and fixed offsets of the contents in the format. (2) *Pattern Refining and Consolidating*. It refines patterns by distinguishing critical patterns from common benign bytes and trivial patterns. It also avoids generating duplicated signatures by examining repeated patterns. (3) *Pattern Context Deriving*. In order to describe what the extracted pattern represents, it analyzes the logs of behaviors and constraints to derive the semantic context of the pattern.

To illustrate the workflow of MetaSymploit, we revisit the script in Listing 1. First, SymExeLayer takes the script as input and symbolically executes it. The script calls a number of symbolic-extended APIs, including `probe_ver`, `shellcode` and `rand_alpha`. Instead of returning a concrete number, `probe_ver` assigns `version` a symbolic integer representing the target version. `shellcode` and `rand_alpha` return symbolic strings to represent all possible shellcodes and random paddings, respectively. Meanwhile, `probe_ver` indicates the probing behavior. SymExeLayer logs it as one attack behavior. SymExeLayer also logs the path constraint `version==5` since it indicates that the Line 6 branch is taken only under the attack condition that the target version is 5. In contrast, when symbolic execution forks to explore Line 8, SymExeLayer logs the negated constraint `version!=5`.

When executing `prep_ark5`, SymExeLayer logs `shellcode`, `rand_alpha`, and `get_target_ret`, since these APIs indicate a typical attack behavior of composing a stack overflow payload. Note that because `get_target_ret` is a call with a concrete argument, SymExeLayer uses the underlying framework to execute it normally to get the concrete return address value. On the other hand, SymExeLayer symbolically extends the `<<` API to support appending symbolic strings. Finally, when the composed payload is sent, the hooked output API `sock.put` captures the complete payload contents.

SigGen then analyzes the payload contents and the behavior & constraint logs to generate signatures. Listing 2 shows one Snort rule generated by SigGen. The content is the byte pattern extracted from the constant bytes in the payload composed in Lines 20-22. The first 8

bytes are two JMP instructions and the last 4 bytes are the return address. The `pcre` is a regular expression matching the entire payload packet, including constant bytes and random paddings. `content` provides general fast matching, while `pcre` provides more precise matching. The `msg` shows the pattern context. The target version is derived from the `version==5` constraint. The behavior and the meaning of the patterns are derived from the logged behavioral API calls. The `msg` gives more insights that guide security analysts to use the signature to protect vulnerable application of specific version.

3.3 Symbolic Execution Layer

This section explains more details about the three techniques of SymExeLayer that extend the attack framework to perform symbolic execution and attack logging.

3.3.1 Symbolic API Extension

The key point of performing symbolic execution on attack scripts is to treat all variant values involved in the attack launching process as symbolic values, so that all possible attack variations can be covered. Since attack scripts use APIs to operate variant values, we extend the variant-related APIs of both the scripting language and the attack framework with symbolic support.

The variant-related APIs can be further divided into two categories: direct and indirect. Direct-variant-related APIs always return variant values. There are two major types in this category, (1) the APIs probing external environments/targets, (2) the APIs generating random payload contents. In both cases, we replace the original APIs with our symbolic-extended ones, which directly return symbolic values when called. As a result, the first type of APIs skips probing the actual environment/target, such as `probe_ver` in the example. Such skipping makes MetaSymploit scalable and efficient, since there is no need to prepare different environments or applications when analyzing different scripts. For the second type, as the payload content is a string of bytes, the APIs use symbolic values to represent any variant bytes, such as `shellcode` and `rand_alpha`. Hence, we can clearly distinguish concrete contents from symbolic contents in one payload. In addition, every symbolic value is assigned with a label showing what it represents based on its related API, such as `sym_ver`, `sym_shellcode`, and `sym_rand_alpha`. Note that SymExeLayer uses these labels to keep the semantics of the values, rather than relying on variable names, which can be freely decided by attackers.

Indirect-variant-related APIs return variant values only when their arguments are variant values. Such case typically happens in the operations of some primitive

classes such as String, Integer, and some payload composing operations. In SymExeLayer, we extend such APIs by adding the logic of handling symbolic arguments. If the arguments are concrete, the APIs execute the original logic and return concrete values as normal. If the arguments are symbolic, the APIs switch to the symbolic handling logic, which propagates the symbolic argument in accord with the API functionality, and returns a symbolic expression. In Listing 1, for a concrete string argument, the symbolic-extended `<<` appends it as normal. For a symbolic argument, it holds both the original string and the new appended symbolic one in order and returns them as one symbolic string expression.

3.3.2 Behavioral API & Attack Constraint Logging

Since symbolic execution is a general program analysis technique, in order to provide additional security analysis of attack scripts, for every execution path, we keep a log recording both critical API calls that reflect attack behaviors and path constraints that represent the attack condition when exploring each execution path.

Behavioral API Logging. As mentioned in Section 2, attack scripts use APIs provided by the language library and the attack framework to launch attacks. In the analysis, it is critical to capture the API calls that perform the detailed attack behaviors during the launching process. There are two major types of behavioral APIs, network protocol APIs and payload-related APIs. By logging the first type, we are able to capture all the interactions between the attack script and the target. By logging the second type, we know exactly how a payload is composed and keep track of its detailed format and contents.

In practice, given an attack framework, we build a knowledge base collecting the APIs from the libraries and components that provide network protocols and payload-related operations. During execution, SymExeLayer identifies behavioral APIs and logs them while keeping the API call sequence in the execution path. Note that we also log the arguments and return values of the APIs, especially for payload-related APIs, whose return values may be a part of the payload contents.

Attack Constraint Logging. In symbolic execution, path constraints are the set of branch conditions involving symbolic values in one execution path. When encountering a new symbolic branch condition, symbolic execution consults a constraint solver to decide which branch(es) is feasible to take, and adds the new branch constraint into the path constraint set. If both branches are feasible to take, the execution path `forks` into two paths to explore both branches [25].

In attack scripts, we focus on the constraints related to environments and targets. We regard these constraints as *attack constraints* because different symbolic conditions

that they represent typically indicate different attack conditions reflecting the probing results of environments or targets, therefore leading to different execution paths that compose different payloads in consequence. In the example, `version==5 ? prep_ark5 : prep_ark4`.

Recall that the APIs that probe external environments and targets are symbolic-extended. The symbolic return values of these APIs carry the labels showing what external source they represent. When executing a symbolic branch condition, we check if any symbolic value with external-source label is involved. If so, we log the corresponding constraint. In the example, when `version==5` is executed, we find that `sym_ver` is an external source, and thus log the constraint.

In summary, this behavior & constraint logging provides a fine-grained analysis report that saves the time-consuming work for security analysts. More importantly, the behaviors and constraints logged in each execution path can be further parsed to derive the semantic context for the extracted patterns (discussed in Section 3.4.3).

3.3.3 Output API Hooking

After an attack script finishes composing an attack payload, the script sends the payload as a network packet to the target to exploit the vulnerability. This payload sending step is the exact point of launching an attack. In order to capture the complete content of the attack payload for pattern extraction, we hook the output APIs that are used by attack scripts for sending payload.

Starting from the network layer to the application layer in the OSI model, we keep a list of the output APIs and their corresponding network protocols from both the scripting language’s own network library and the built-in components of the attack framework.

We symbolically extend the output APIs by overriding their functionality from sending real network packets to dumping the entire packets locally. By doing so, the entire network flow sent from the attack script can be dumped throughout the execution. To keep the semantic context of each dumped packet, we associate them with the behavior & constraint log of that execution path, so that later the payload packets can be identified and the extracted patterns can be correlated with the context derived from the log. In the example script, the hooked `sock.put` dumps two packets. With the associated log, we identify the payload packet for pattern extraction.

Note that as a part of the network protocol APIs, the output APIs are also behavioral APIs that need to be logged. In addition, we also include the corresponding network protocols in the log. Later during signature generation, the log gives a clear view of which network protocol is used, and therefore SigGen can apply the correct packet format when parsing the packet contents.

```

18: payload=>[<sym_shellcode, len=Sym_Int>]
19: payload=>[<sym_shellcode, len=Sym_Int>,
   <sym_rand_alpha, len=(1167-Sym_Int)>]
20-22: # Appending concrete substrings
payload => [<sym_shellcode, len=Sym_Int>,
   <sym_rand_alpha, len=(1167-Sym_Int)>,
   <"\xe9\x38\x6c\xfb\xff\xff\xeb\xf9
\xad\x32\xaa\x71", 12>]
23: payload => [<sym_shellcode, len=Sym_Int>,
   <sym_rand_alpha, len=(1167-Sym_Int)>,
   <"\xe9\x38\x6c\xfb\xff\xff\xeb\xf9
\xad\x32\xaa\x71", 12>,
   <sym_rand_alpha, 2917>]

```

Listing 3: The symbolic string form showing the content of `payload` when `prep_ark5` is executed. `Sym_Int` is a symbolic integer representing the size of the shellcode.

3.4 Signature Generator

Given the dumped payload packets and the logs as inputs, SigGen includes three techniques to generate signatures.

3.4.1 Constant Pattern Extracting

In order to generate a signature that can detect a payload packet, it is necessary to extract a set of constant patterns that always stay the same across different variations of the payload. Specifically, there are three constant patterns that can be extracted: fixed-content pattern, fixed-length pattern and fixed-offset pattern. For ease of explanation, we first present the formal form of a dumped symbolic attack payload.

Recall that an attack payload is a string of bytes containing both concrete contents (e.g., fixed vulnerable return address) and variant contents (e.g., arbitrary shellcode, random padding). When a payload is being composed during the symbolic execution of the attack script, we use symbolic strings to represent variant contents and use extended APIs to perform symbolic string operations, while keeping concrete values and operations as normal. Thus the dumped payload packet is a big symbolic string composed of a sequence of substrings, where each substring is either a concrete byte string or a symbolic string by itself. Formally, $S_{sym} = (s_1 s_2 \dots s_i \dots s_n)$, where $s_i \in \{S_{con}\} \cup \{S_{sym}\}$. In addition, we also embed $\langle sym_label, length \rangle$ in S_{sym} to keep the semantics and the possible length of the string, where the length is either a concrete or symbolic integer. As an example, Listing 3 shows the contents of the payload when being composed in Lines 18-23 of the example script. The final dumped payload is the same as the one in Line 23.

Fixed-content pattern. This pattern has two types, either a simple byte string or a regular expression (regex). When parsing the payload, for each concrete substring, we extract it as a byte string pattern, such as the 12-byte string in the payload of Line 23. For each symbolic sub-

string, if it can be matched by a regex, we extract the regex as a fixed pattern. If no regex is found, we move on to the next substring. In practice, we keep a mapping between regex-matchable symbolic labels and the regexes. Currently, we mainly focus on using regexes on payload paddings to achieve precise matching. For instance, we map the symbolic label `sym_rand_alpha` to a regex pattern `[a-zA-Z]`.

Fixed-length pattern. In some cases, although the contents may vary, their lengths stay the same. Such case typically happens when using padding to meet the size requirement. To achieve precise matching, SymExeLayer keeps track of the payload length during the composition. When parsing the payload, we identify the symbolic substrings with fixed lengths and extract them as patterns. When executing the example script in SymExeLayer, we keep updating the payload length. Later when parsing `<sym_rand_alpha, 2917>` in the dumped payload, we produce a length-quantified regex `[a-zA-Z]{2917}` as shown in Listing 2.

Fixed-offset pattern. Due to the format of some network protocols, some payloads can be located only after certain offsets of the packets. For instance, some FTP-based attack packets have regular FTP commands, followed with overlong paths as payloads to launch overflow attacks. In such cases, since the network protocol of the output API is logged, by applying the packet format of the protocol, we extract the offset of the payload, which is a pattern for precise matching of the payload location.

3.4.2 Pattern Refining and Consolidating

As MetaSymploit automatically generates signatures in a large scale, there are two requirements for the quality of the signatures. First, we should avoid generating signatures only having patterns of common benign bytes or patterns of trivial bytes/regexes, which may otherwise cause false positive. Second, we should avoid generating duplicated signatures with the same pattern set, which may cause useless redundancy and confuse the IDS.

First requirement. When a payload is finally sent through the output API, common benign bytes are introduced by network protocols as concrete substrings in the payload packet, including default protocol bytes (e.g., “Content-Type:text/html”) and delimiter bytes (e.g., “\r\n”). To identify them, for each protocol, we keep a list of benign bytes. Based on the packet format, we examine the concrete substrings to search for the occurrences of benign bytes. If found, we strip the benign part and focus on the rest bytes for pattern extraction.

In addition, it is also important to avoid generating signatures only using trivial patterns such as too short byte string or too general regex patterns. Thus, we set a

threshold of minimum byte string length (e.g., ≥ 10) and a list of critical regexes (e.g., NOP regex $[\x90]^*$). Given a set of extracted patterns, we generate signatures only if we can find at least one pattern whose length is above the threshold or whose regex is critical. Note that both the threshold and the critical regex list are adjustable. Security analysts can also define different thresholds and lists for different network protocols.

Second Requirement. Recall that SymExeLayer explores different execution paths in an attack script and dumps payloads in each path. Sometimes, two paths may differ only in a branch that is irrelevant to the payload content, thus finally composing the same payloads. Furthermore, two attack scripts may also share the same patterns. To consolidate the same patterns from different payloads into one signature, we keep a key-value hash map where each key is a pattern set and each value is a set of different payloads with the same pattern set. When a new payload is parsed, if its pattern set already exists in the hash map, we add this new payload, particularly its behavior & constraint log into the corresponding value set. The payloads and the logs in one set are analyzed together to generate only one signature.

3.4.3 Pattern Context Deriving

Apart from pattern extraction, it is equally important to provide the context of the patterns. The pattern context shows the insight into the attack script, such as what attack behavior and attack payload the patterns represent. It also gives security analysts the guidance on how to use the patterns, such as which target version and what OS environment the patterns can be used to protect.

Therefore, we analyze the behavior & constraint log to derive the pattern context. Since attack behaviors are captured as behavioral APIs in the log, we derive the context by translating the behavioral APIs into human-readable phrases. Some APIs have straightforward names, which can be simply translated into the description phrase (or even directly used), such as `probe_ver => Version Probing`. Others may not be intuitive. Particularly, certain behavior cannot be shown from a single API but a series of API calls. In such case, we group these API calls together as one behavioral pattern. When such pattern is found in the log, we translate it into the matched behavior name, such as `shellcode + get_target_ret => Stack Overflow`.

Sometimes, sophisticated attack scripts may have unprecedented behaviors whose APIs do not match any patterns. In such cases, we keep the derivable context while highlighting underived behavioral APIs in the log to help security analysts discover new attack behaviors. In fact, we use this technique in our prototype to collect patterns.

In regard to attack constraints, since the involved sym-

bolic values represent attack conditions of each execution path, we retrieve the external source names in the symbolic labels and bind them with the conditions derived from the constraints (e.g., Target Version: 5).

Finally, when both the extracted pattern set and the derived context are ready, SigGen combines two together and generates a signature, which can be used to detect the payloads associated with this specific pattern set.

4 Implementation

We implement a prototype of MetaSymploit as a practical analysis tool targeting the Ruby-based attack framework Metasploit. Given a Metasploit attack script, our tool quickly analyzes it and automatically generates Snort rules as signatures that can defend against this specific script. Particularly, we developed a lightweight Ruby symbolic execution engine designed for attack script analysis. Powered by the engine, we build SymExeLayer on top of the launching platform of Metasploit. In this section, we first describe how the engine is designed and then explain how to adapt the engine for Metasploit.

4.1 A Lightweight Symbolic Execution Engine for Ruby

Traditionally, developing a symbolic execution engine requires heavy modification of the interpreter, which causes great engineering effort since Ruby has multiple active versions and interpreters (e.g., 1.8/1.9/2.0). However, we discover a new way to design a lightweight engine without modifying the interpreter. The engine is developed purely in Ruby (9.3K SLOC) as a loadable package compatible with multiple versions of Ruby. Thus it supports analyzing attack scripts written in different versions. Specifically, our engine has two modules: (1) a symbolic library that introduces rich symbolic support into Ruby; (2) a symbolic execution tracer that performs symbolic execution based on the actual script execution.

4.1.1 Library of Symbolic Support

The symbolic library realizes the functionality of *Symbolic API Extension*. The library introduces symbolic classes to hold symbolic values (e.g., `SymbolicString`, `SymbolicInteger`). To be transparent to attack scripts, we develop the same APIs in the symbolic classes as their concrete counterparts. On the other hand, we also extend indirect-variant-related APIs in the concrete classes to support handling symbolic arguments, so that concrete and symbolic objects can operate with each other.

Notably, `SymbolicString` class plays the key role in representing attack payloads. To hold the con-

tents, `SymbolicString` has an internal ordered array, where each item is either a concrete substring, or a symbolic substring with the `<sym_label, length>` embedded. When a `SymbolicString` API is called, it first checks whether the original concrete operation is still applicable to the concrete substrings. If so, the API uses the original logic in `String` to operate the concrete substrings. Otherwise, the API treats the contents as symbolic substrings, and processes the internal string array as symbolic expressions. When a symbolic-extended `String` API is called with symbolic arguments, it handles concrete and symbolic substrings in the same way as above and returns a `SymbolicString` object.

Later when `SymExeLayer` is integrated with Metasploit, we further include the symbolic-extended APIs of Metasploit into the symbolic library.

4.1.2 Symbolic Execution Tracer

The symbolic execution tracer transforms normal script execution into symbolic execution. It also realizes the functionality of *Behavior & Constraint logging*. To this end, we develop three techniques based on three advanced language features in Ruby (& Python³).

(1) Fine-grained execution tracing. This technique traces the symbolic execution line-by-line in an attack script. It keeps track of every method call. It also explores different paths when executing branches. We develop it by enhancing a language feature called *Debug tracing function* with Control Flow Graph (CFG).

Debug tracing function is a step-by-step execution tracing facility used for debugging such as Ruby’s `set_trace_func` (Python’s `sys.settrace`). It captures three major events, *line*, *call*, *return*. The *line* event shows the number of the current executing line. The *call/return* event shows the name of the method being called/returned. Every time an event happens, *Debug tracing function* suspends the execution and calls a registered callback function for further event analysis.

We develop our callback function using the CFG of the attack script. Since the CFG holds both the source code and the control flows, it offers rich semantics for analyzing the execution details when parsing every event. When a *line* event happens, we locate the current line’s source code in the CFG. Then we retrieve all call sites in the current line, which will be matched with the following *call/return* events happening in this line. Particularly, this tracing mechanism can log behavioral API calls when they are found in the call sites.

Our callback function also handles branches to explore different paths. When the *line* event reaches a symbolic branch, we evaluate the branch source code and consult

³The techniques can also build an engine to analyze Python-based attack scripts, since Ruby and Python share many language features.

a constraint solver for both true and false branch constraints. If a solution exists, we concretize the symbolic branch condition to guide the interpreter to the desired branch (explained next). If both branches can be satisfied, we `fork` the script execution process into two processes to trace both branches. Otherwise, if no solution is returned, we terminate the execution process. Particularly, if attack constraints are found, the callback function would perform constraint logging.

(2) Runtime symbolic variable manipulation. This technique leverages the *Runtime context binding* language feature to manipulate the runtime values of symbolic variables. In particular, it inspects the values of attack payloads during composing. It also concretizes symbolic branch conditions to guide branch execution.

Runtime context binding can inspect and modify the runtime states of the script, such as Ruby’s `Binding` and Python’s `inspect`. It provides a `context` object that binds the runtime scope of the current traced code. The callback function can use this object to access all variables and methods in the scope of the traced code.

The first use of `context` is to inspect the runtime value of an attack payload when it is being composed. When a variable is detected to be assigned by payload composing APIs, the callback uses `context` to keep track of its value. The callback then logs the inspected values together with the payload composing APIs in the behavior log.

The second use of `context` is to guide symbolic branch execution. Since the interpreter cannot move forward with a symbolic condition, when the constraint solver returns a solution, for each symbolic variable in the condition, we use `context` to temporarily replace the symbolic value with the solved concrete value to guide the interpreter to the desired branch. Later when the *line* event shows that the branch is taken, we recover them back to their symbolic form. Recall the `version==5` in Listing 1. Since `version` is symbolic value, we temporarily replace its value with 5 to explore one branch, and uses a non-5 value for the other branch.

(3) Dynamic symbolic method wrapping. In some cases, the symbolic return values of method calls are not associated with any variables, thus cannot be manipulated using the second technique. To handle this, we leverage the *Dynamic method overriding* language feature to dynamically wrap the traced method, associate its return value with a temporary variable for manipulation.

Dynamic method overriding is a common feature in Ruby and Python that methods can be runtimely overridden and take effect immediately. Using this language feature, we dynamically create a wrapper method and override the original method right before the *call* event. Meanwhile, we also preserve the original method, and recover it right after the *return* event.

A more important use of the wrapping technique is to

concretize symbolic methods in branch conditions. If no variable holds the symbolic return value of a method call in a branch condition, to guide symbolic branch execution, we override the symbolic method with the wrapper to return a solved concrete value. In practice some constraint solvers require the symbolic method calls to be associated with variables to enable the solving.

4.2 Adaptation for Metasploit

To analyze Metasploit attack scripts, we adapt the engine and the six techniques in both SymExeLayer and SigGen to work with the APIs provided by Metasploit and its built-in components.

The current prototype is based on Metasploit version 4.4 (released in Aug 2012). We select the top 10 most popular built-in components in Metasploit: Tcp, Udp, Ftp, Http, Imap, Exe, Seh, Omelet, Egghunter, Brute. The first 5 are popular network protocol components. The next 4 are used to attack Windows systems. Exe can generate exe file payloads. Seh can create SEH-based attacks. Both Omelet and Egghunter can compose staged payloads. The last Brute can create bruteforce attacks. These components cover 548 real attack scripts carried in Metasploit. By examining the APIs provided by the launching platform and these components of Metasploit, we perform three steps to adapt the engine for SymExeLayer and SigGen.

First, in the symbolic library, we apply symbolic API extension to the environment-related APIs such as `tcp.get`, `ftp.login`, `http.read_response`, and variant-payload-content-related APIs such as `rand_text`, `make_nops`, `gen_shellcode`⁴. The library also replaces the output APIs such as `ftp.send_cmd`, `http.send_request` with our local-dumping APIs. When the script calls these APIs during symbolic execution, SymExeLayer redirects the calls to the symbolic-extended APIs.

Second, to equip the symbolic execution tracer with behavior & constraint logging ability, we build a knowledge base collecting behavioral APIs such as `http.fingerprint`, `gen_egghunter` and keep a mapping between APIs and their behavior meaning for pattern context deriving. We also keep a list of symbolic labels for identifying attack constraints.

Third, based on the standards of the protocols and the implementation of the built-in components, we add the packet formats and common benign bytes of the five network protocols into the knowledge base. For instance, we develop specific parsers to parse payloads embedded in HTTP headers and FTP commands.

⁴The listed API names are abbreviated due to space limits. Note that Metasploit uses `payload` to represent shellcode. We use `shellcode` as a more general term to avoid confusion with attack payloads.

Note that both the API extension and the knowledge base are one-time system configuration. Since Metasploit components and their APIs are relatively stable for compatibility with various attack scripts, once they are collected and supported by MetaSymploit, newly distributed attack scripts that rely on these components can be directly supported and automatically analyzed.

5 Evaluation

We conduct our evaluation on an Intel Core i7 Quad 2.4GHz, 8GB memory, Ubuntu 12.10 machine. We run MetaSymploit based on Metasploit 4.4, using the official Ruby 1.9.3 interpreter. We evaluate our approach from three perspectives: (1) the percentage of real-world attack scripts that can be analyzed by MetaSymploit's symbolic execution; (2) the effectiveness of our automatically generated signatures to defend against real-world attacks; (3) the difference between our automatically generated rules and official Snort rules.

5.1 Coverage Testing with Symbolic Execution Engine

We first evaluate whether MetaSymploit can symbolically execute various attack scripts. We use MetaSymploit to analyze all 548 real attack scripts created with the top 10 popular Metasploit components. As the result shown in Table 1, 509 scripts (92.88%) are automatically executed by MetaSymploit in the symbolic mode without any manual modification of the scripts. Different attack conditions in the scripts are explored. The attack payloads are captured and Snort rules are generated.

In terms of analysis cost, since MetaSymploit reuses the launching platform of Metasploit on the official Ruby interpreter, the symbolic execution has almost the same speed as that Metasploit executes attack scripts normally (less than one minute on average). In fact, since the environment-related APIs are symbolic-extended, the time for real network communication is saved. Furthermore, signatures are generated in less than 10 seconds.

Among the remaining 39 scripts that MetaSymploit cannot automatically deal with, we encounter five main situations that deserve more discussion.

Loop with Symbolic Condition. We find that 9 scripts have conditional loops whose symbolic conditions cannot be solved by constraint solvers, which may cause infinite looping. As a common problem in classical symbolic execution, some previous approaches proposed using random concrete values to replace symbolic conditions to execute loops [20]. However, in our case, doing so may affect the precision of the payload contents. Other approaches such as LESE [35] specifically handle loops, which we plan to explore in future work.

Category	Num	Percentage	Require Manual Modification
Automatically Executed	509	92.88%	No
Symbolic Loop	9	1.64%	Avg 10 LOC/per script
Non-Symbolic-Extended API Call	12	2.19%	Avg 3 LOC/per script
Obfuscation & Encryption	13	2.37%	Not Supported
Multi-threading	3	0.55%	Not Supported
Bug in Scripts	2	0.37%	2 LOC in each script
Total Coverage		Auto 92.88%	All 96.90%

Table 1: The distribution of different situations in the symbolic execution of the 548 Metasploit attack scripts.

Currently, after manual analysis, we find that there are two cases of using the loops: byte-by-byte modifying a symbolic string whose length is a symbolic integer, and performing repeated attack steps in a bruteforce attack. In the first case, since the string length is not concrete, the looping rounds cannot be decided. However, we find no matter how many rounds are, the looping result is still a symbolic string. Therefore, we replace the loop code that operates the symbolic string with a new symbolic string to represent the looping result (10 LOC per script on average), while propagating the symbolic label and logging the loop information for further investigation.

In the second case, the Brute component provides an API that checks whether the target is compromised or not. It is typically used as a while loop condition. The loop keeps attacking the target until the API returns that the target is compromised. Since in our case the API returns a symbolic value as the target status, to avoid infinite looping, we set a counter with an upper bound in the extended version of this API, to control the looping rounds. If there are payloads and logs captured inside the loop, the differences between each round are analyzed to identify the constant patterns.

Non-Symbolic-Extended API Call. Due to the time limitations, other than the top 10 components, we have not symbolically extended other APIs in Metasploit. We detect 12 scripts that call the non-extended APIs related to assembly translating and encoding the payloads. Since very few APIs are involved, we decide to modify each of them individually at this time, and extend the entire components in future work. To handle these API calls, since `SymbolicString` supports payload content processing, when applicable to the concrete substrings, we allow the APIs to operate on the concrete parts, while preventing them from using the symbolic substrings, which may otherwise cause runtime errors. When the API operates on a pure symbolic string with no concrete substrings, we replace the API calls by creating new symbolic strings to represent the results of the API calls (3 LOC per script on average).

Obfuscation & Encryption. There are 13 cases with complicated obfuscation and encryption on the payload, where payload content processing is not feasible. Since the output of these operations is completely random,

there is no constant pattern that can be extracted from the obfuscated or encrypted payload. Defending against obfuscation and encryption is an open question, which is beyond the scope of signature-based defense.

Multi-threading. Handling multi-threading is an advanced topic in symbolic execution. Existing research [37] explored the possibility by extending symbolic execution to handle multi-threaded programs. Currently, due to only 3 cases related to this situation, we plan to address this issue in future work.

Bug in Scripts. Interestingly, during the testing, we also discover 2 scripts with bugs that hang the execution when the script is generating a specific assembly code that jumps to the shellcode. From this result, we see that our approach is also useful for the purpose of finding bugs in attack scripts.

In summary, the percentage of scripts that are automatically handled is 92.88%. If the manually modified scripts are included, the percentage reaches 96.90%.

5.2 Effectiveness Validation using Real-world Attacks

To evaluate whether the automatically generated Snort rules can effectively detect real attacks, we use Metasploit attack scripts to attack 45 real-world vulnerable applications. These applications are acquired from `exploit-db.com`, a popular hacking website collecting attack scripts and free vulnerable applications. In all, there are 45 free vulnerable applications available in the website, with 45 corresponding Metasploit scripts. They include Java 7, Adobe Flash Player 10, Apache servers 2.0, Firefox 3.6, RealPlayer 11, multiple FTP servers such as Dream FTP, ProFTPD, VLC player 1.1, IRC servers and some less popular web-based programs.

We first use MetaSymplloit to analyze the 45 attack scripts and automatically generate Snort rules. Then we set up two virtual machines, with one running Metasploit to simulate the attacker and the other running the vulnerable application as the vulnerable target. For each script, we choose two different shellcodes to launch two real attacks. To expose the entire attack flow, we allow the attack to compromise the target, and use Snort IDS 2.9.2 with our generated rules to detect attack payloads. Note

that due to the limited available versions of the applications, we focus on the rules of the attack payloads that target the application versions that we are able to obtain.

The initial results show that except the HTTP-based ones, all attack payload packets with both two types of shellcodes are correctly detected. Recall that our rules are based on the constant patterns of the payload, variant parts such as shellcodes do not affect the detection. But for Apache server attacks and Firefox attacks, our rules fail to catch the attack packets because the order of each HTTP header field is different from the one in our rules. Since the order of the HTTP header fields is not enforced by RFC definition, the extracted patterns from the HTTP header cannot be simply put into the signature in sequence. Therefore, we further improve our HTTP parser to handle each header field separately, to enable order-insensitive pattern matching. In the second round of testing, the HTTP-based attacks are also correctly detected.

Another interesting case is the Java 7 attack. In late Aug 2012, two days after a zero-day vulnerability in Java 7 was disclosed (CVE 2012-4681), a Metasploit attack script was distributed targeting this vulnerability [32]. At that time, we immediately used MetaSymploit to analyze this attack script and automatically generate a Snort rule based on the malicious jar payload composed by this script, and tested it in our environment. Our rule successfully detected the jar payload. Admittedly, there might be other ways different from the distributed Metasploit script to exploit the vulnerability. Nevertheless, our rule provides the first aid to the vulnerability without available security patch, to defend against attackers who directly use this widely-distributed script to launch attacks.

Apart from the effectiveness evaluation, we also use our rules generated from the 45 attack scripts to monitor normal network traffic, to investigate whether our rules would raise false positives on benign packets. We run the Snort with our rules in promiscuous mode to monitor the traffic of two Windows machines (Vista & 7) and a Ubuntu 12.04 machine. These machines are everyday-use machines in the CS department (no personal data is recorded). The monitoring is online for two months. No false positive is raised on benign packets. Such result is expected since our rules contain multiple specific patterns that matches only the Metasploit attack payloads. Appendix A shows a rule example for one of the 45 scripts.

5.3 Comparison with Official Snort Rules

To further assess the quality of the generated rules, we compare the MetaSymploit rules (MRs) of the 45 attack scripts with the recent Official Snort rules (ORs),

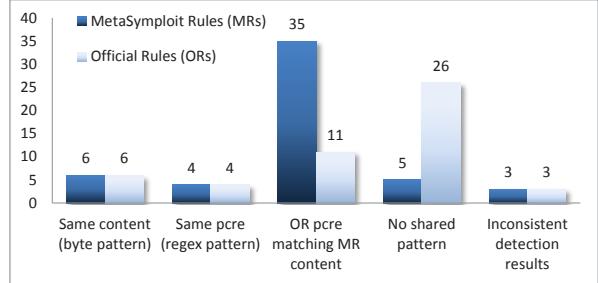


Figure 2: Pattern comparison between 53 MetaSymploit generated rules and 50 official Snort rules for 22 Metasploit attack scripts.

released in Nov 2012⁵. We use CVE number carried in both attack scripts and ORs to match each other. The result is surprising that only 22 attack scripts have corresponding ORs. The rest 23 are not even covered by ORs. This reveals a serious issue that existing defense is still quite insufficient compared to the fast spreading of public attack resources.

For the 22 officially covered scripts, there are 53 MRs and 50 ORs. In MetaSymploit, one script may have multiple rules detecting different payloads for different target versions. Whereas in the official rule set, one vulnerability may also have multiple rules detecting different ways that exploit it. By comparing the patterns in both rule sets, we summarize the result in Figure 2. We find that 44 MRs share patterns with 21 ORs. Specifically, 6 MRs and 6 ORs share the same content byte patterns. 4 MRs and 4 ORs share the same pcre regex patterns. Notably, 35 MRs have specific content that are matched with 11 ORs' general pcre. This is because the pcre regexes are generalized by security analysts based on large volumes of samples, while the content bytes (usually including vulnerable return addresses) are generated based on every attack payload of the scripts. An example is shown in Appendix A. Although in this case, the MR set is a subset of the OR one, we argue that as our goal is to defend against specific attack scripts, MRs give more insight of the attack payloads with more precise matching. Meanwhile, there are 5 MRs and 26 ORs with no pattern shared. This is because some vulnerabilities can be exploited in different ways, and the ORs have more patterns defined by analysts, while Metasploit scripts usually choose one way to exploit one vulnerability. Nevertheless, we still find that 2 scripts have 5 MRs whose patterns are not seen in ORs, which complement the OR set.

Besides, we also load the 50 ORs into Snort to test whether they can detect attacks launched by the 22 attack scripts. Interestingly, the result shows that only 17 scripts' attack payloads are detected, while no alert

⁵snortrules-snapshot-2922.tar.gz on www.snort.org/snort-rules/

is raised for the other 5 scripts. 2 scripts⁶ are missed due to the lack of OR patterns as we mentioned above. The other 3 scripts, which have 3 MRs, are supposed to be detected by 3 corresponding ORs. After comparing these rules, we find the 3 ORs have some deficiencies that cause this inconsistent detection results. We list the detailed information of the 3 scripts and the deficiencies of the 3 ORs in Table 2. Note that some deficiencies are actually caused by inaccurate use of Snort rule flags such as the `http_uri`, `flow`. We find them by comparing these flags with the pattern context (e.g., Behaviors) in our rules. We have reported these discoveries to the official Snort team.

In sum, these results show that even the official Snort rules written by security analysts are incomplete and tend to be error-prone. MetaSymploit serves as a useful tool to complement and augment the existing IDS signatures by improving the completeness and the accuracy.

6 Discussion

Scenarios of using MetaSymploit signatures. As shown in the comparison (35 MRs vs 11 ORs), due to different pattern extracting mechanisms, ORs have less rules with more general patterns, while MRs have more rules with more specific patterns. It is possible that as the number of attack scripts is increasing, more and more signatures will be generated. If all signatures are loaded into the IDS, this may slow down the matching speed.

However, we argue that unlike ORs are used for general detection, MetaSymploit signatures should be used in two typical scenarios, which do not require loading all MRs in an IDS. First, as the goal of MetaSymploit is to provide quick defense against newly distributed attack scripts, the typical way of using our signatures is to give first aid to the vulnerable application without available patches to prevent attackers especially script kiddies using the new scripts to launch attacks (e.g., the Java 7 case). When the vulnerability is patched or the application is upgraded, our signatures can be removed from the IDS. Second, as the pattern contexts are embedded with the signatures, security analysts only need to deploy the signatures whose contexts are related to the protected environment or the protected target version, to avoid loading irrelevant signatures which may slow down the matching speed of the IDS.

Limitations. MetaSymploit inherits the limitations of classical symbolic execution. As we mentioned in Section 5.1, our current prototype requires manual analysis on handling complex symbolic loops. Recent approaches propose to use bounded iteration [21], search-guiding heuristics [40] and loop summary [22, 35] to address the

loop issue. In MetaSymploit, different loop cases of attack scripts may require different techniques. For example, bounded iteration can be applied to handle loops of brute-force attacks. Loop summaries can summarize the post-loop effect on symbolic payload contents. Search-guiding heuristics can help target payload-related loops to avoid getting stuck in irrelevant loops.

Apart from loops, path explosion is a more general issue related to performance and scalability. Too many paths in an attack script may prolong the analysis and delay the defense. In addition, it is possible that different paths in a script finally lead to the same attack payload output. Exploring these paths incurs extra efforts of pruning redundant payloads. Several techniques such as equivalent state tracking [9], state merging [26] and path partitioning [31] have been proposed to mitigate the path explosion issue. We plan to incorporate these techniques into MetaSymploit to avoid exploring paths that would compose redundant payload contents.

The limitations of constraint solvers may also affect the effectiveness of path exploration. Currently, we use Gecode/R [1] for solving integer/boolean constraints and HAMPI [23] for solving string constraints. In case when encountering complicated constraints (e.g., a non-linear constraint), the solvers cannot decide which branch to take. For the sake of completeness, we conservatively explore both branches, while marking the path constraints as uncertain in the log, which require more investigation by security analysts. Due to this fact, we regard our prototype as an assistant tool to reduce the workload of analysts, so that they only need to focus on complicated ones when facing large numbers of new attack scripts.

We envision possible attacks directly against MetaSymploit’s defense mechanism. As MetaSymploit rules stick to the patterns in the distributed attack scripts, it is possible that experienced attackers may modify the distributed one to create new script variants without releasing them, which may evade the detection of MetaSymploit rules. Besides, experienced attackers may also try to exploit the limitation of symbolic execution when developing new scripts, such as introducing complex loops, non-linear constraints or even obfuscating the script code. However, both cases are non-trivial. They require advanced attack developing techniques, which are usually time-consuming and slow down the speed of developing and launching new attacks. In other words, with MetaSymploit, we raise the bar of the skill level and the time cost for developing and launching new attacks.

7 Related Work

Signature Generation. There has been a lot of work on automatic signature generation for malware defense. From the perspective of attacks, Autograph [24], Poly-

⁶`adobe_flash_sps.rb`, `mozilla_mchannel.rb`

Metasploit Script Name	CVE	Failure Reason of Official Snort Rules Missing Metasploit Payloads	Official Rule SID
badblue_ext_overflow.rb	2005-0595	The <code>http_uri</code> flag restricts the pattern searching in one header field, thus missing the Metasploit payload located in the following fields.	3816
sascam_get.rb	2008-6898	The <code>flow</code> pattern is set to check packets sent to the client while our pattern context shows the Metasploit payload is sent to the server.	16715
mozilla_reduceright.rb	2011-2371	The <code>content</code> byte pattern is wrong since it includes two variant bytes, which are randomly generated in the Metasploit payload.	19713

Table 2: The list of three Metasploit attack scripts which evade the detection from 3 Official Snort Rules

graph [29] and Hamsa [27] automatically generate worm signatures by extracting invariant contents from the network traffic of worms. Particularly, these approaches are based on the observation that even polymorphic worms have invariant contents that can be used as signature patterns. In MetaSymploit, we have the same observation when analyzing constant and variant payload contents composed by attack scripts. On the other hand, these approaches require collecting large amounts of malicious network traffic to identify invariant contents. However, this process is usually time-consuming and cannot provide quick defense against new attacks. In contrast, MetaSymploit does not need to collect any network traffic but only attack scripts, thus largely reducing the time of performing analysis and providing defense.

From the perspective of vulnerabilities, Vigilante [18], ShieldGen [19] and Bouncer [17] analyze vulnerable applications and their execution traces to generate signatures to block exploit inputs that can trigger the vulnerability. Brumley et al. [10, 11] also provide the formal definition of vulnerability-based signatures and propose constraint-solving-based techniques to generate such signatures. Elcano [13] and MACE [16] further use protocol-level concolic exploration to generate vulnerability-based signatures. Notably, program analysis techniques such as symbolic execution play an important role in these approaches as well as in MetaSymploit. But unlike these approaches, MetaSymploit only analyzes attack scripts without requiring the presence of vulnerable applications, thus avoiding the cost of obtaining various vulnerable applications or preparing various testing environments.

Symbolic Execution. Symbolic execution has been actively applied for security purposes [36]. BitBlaze [38] is a binary analysis platform based on symbolic execution. SAGE [21] uses dynamic symbolic execution to detect vulnerabilities in x86 binaries. EXE [14] and AEG [8] generate malicious inputs and exploits by symbolically executing vulnerable applications. Moser et al. [28] explore multiple execution paths for malware analysis. Since our analysis target, attack script is quite different from host-based binary level malware, the techniques proposed in these approaches such as memory inspection, system call analysis are not adaptable in our case.

Symbolic execution for scripting languages is still

at early stage, due to the diversity of different kinds of scripting languages and various purposes of applications. Most work focuses on the web-based scripting languages, such as JavaScript [34], PHP [7, 41], and Ruby on Rails [15] web frameworks. Since these approaches are specifically designed for testing web applications (e.g., finding XSS and SQL Injection vulnerability), they are not applicable for analyzing general attack scripts and attack frameworks that target various vulnerable applications on different OS environments.

In particular, little work has been done for the symbolic execution of general-purpose scripting languages, such as Ruby and Python. PyStick [30] is an automated testing tool with input generation and invariant detection for Python. It is different from our purpose of using symbolic execution for security analysis. Bruni et al. [12] propose a library-based approach to develop symbolic execution. However, it uses only the dynamic dispatching feature, which limits symbolic execution only in primitive types. This limited functionality is insufficient for practical use.

8 Conclusion

Script-based attack frameworks have become an increasing threat to computer security. In this paper, we have presented MetaSymploit, the first system of automatic attack script analysis and IDS signature generation. MetaSymploit leverages security-enhanced symbolic execution to analyze attack scripts. We have implemented a prototype targeting the popular attack framework Metasploit. The results have shown the effectiveness of MetaSymploit in real-world attacks, and also the practical use in improving current IDS signatures.

9 Acknowledgements

We would like to thank the conference reviewers and shepherds for their feedback in finalizing this paper. This work is supported by the U.S. Army Research Office (ARO) under a MURI grant W911NF-09-1-0525, and also supported in part by an NSA Science of Security Tablet grant at North Carolina State University.

sity, NSF grants CCF-0845272, CCF-0915400, CNS-0958235, CNS-1160603.

References

- [1] Constraint programming in ruby. <http://gecoder.rubyforge.org/>.
- [2] The exploit database. <http://www.exploit-db.com>.
- [3] Metasploit. <http://www.metasploit.com>.
- [4] Arkeia backup client type 77 overflow (win32). <http://www.metasploit.com/modules/exploit/windows/arkiea/type77>, visited in Jan 2013.
- [5] Conficker worm using metasploit payload to spread. <http://blogs.mcafee.com/mcafee-labs/conficker-worm-using-metasploit-payload-to-spread>, visited in Jan 2013.
- [6] Top vulnerability exploit tools. <http://sectools.org/tag/splloits>, visited in Jan 2013.
- [7] ARTZI, S., KIE, A., DOLBY, J., ERNST, M. D., KIEZUN, A., TIP, F., DIG, D., AND PARADKAR, A. Finding Bugs In Dynamic Web Applications. In *Proceedings of the 2008 International Symposium on Software Testing and Analysis (ISSTA'08)*, pp. 261–272.
- [8] AVGERINOS, T., CHA, S. K., LIM, B., HAO, T., AND BRUMLEY, D. AEG: Automatic Exploit Generation. In *Proceedings of the Network and Distributed System Security Symposium (NDSS'11)*.
- [9] BOONSTOPPEL, P., CADAR, C., AND ENGLER, D. RWset: Attacking Path Explosion in Constraint-Based Test Generation. In *Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'08)*, pp. 351–366.
- [10] BRUMLEY, D., NEWSOME, J., AND SONG, D. Towards Automatic Generation of Vulnerability-Based Signatures. In *Proceedings of the 2006 IEEE Symposium on Security and Privacy (S&P'06)*, pp. 2–16.
- [11] BRUMLEY, D., WANG, H., JHA, S., AND SONG, D. Creating Vulnerability Signatures Using Weakest Preconditions. In *Proceedings of the 20th IEEE Computer Security Foundations Symposium (CSF'07)*, pp. 311–325.
- [12] BRUNI, ALESSANDRO DISNEY, T. A Peer Architecture for Lightweight Symbolic Execution. Tech. rep., UC Santa Cruz, 2011.
- [13] CABALLERO, J., LIANG, Z., POOSANKAM, P., AND SONG, D. Towards Generating High Coverage Vulnerability-Based Signatures with Protocol-Level Constraint-Guided Exploration. In *Proceedings of the 12th International Symposium on Recent Advances in Intrusion Detection (RAID'09)*, pp. 161–181.
- [14] CADAR, C., GANESH, V., AND PAWLOWSKI, P. EXE: Automatically Generating Inputs of Death. In *Proceedings of the 13th ACM Conference on Computer and Communications Security (CCS'06)*, pp. 322 – 335.
- [15] CHAUDHURI, A., AND FOSTER, J. S. Symbolic Security Analysis of Ruby-on-Rails Web Applications. In *Proceedings of the 17th ACM Conference on Computer and Communications Security (CCS'10)*, pp. 585–594.
- [16] CHO, C., BABIC, D., AND POOSANKAM, P. MACE: Model-inference-Assisted Concolic Exploration for Protocol and Vulnerability Discovery. In *Proceedings of the 20th USENIX Security Symposium (2011)*.
- [17] COSTA, M., CASTRO, M., AND ZHOU, L. Bouncer: Securing Software by Blocking Bad Input. In *Proceedings of twenty-first ACM SIGOPS Symposium on Operating Systems Principles (SOSP'07)*, pp. 117–130.
- [18] COSTA, M., CROWCROFT, J., AND CASTRO, M. Vigilante: End-to-End Containment of Internet Worms. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP'05)*, pp. 133–147.
- [19] CUI, W., PEINADO, M., WANG, H. J., AND LOCASTO, M. E. ShieldGen: Automatic Data Patch Generation for Unknown Vulnerabilities with Informed Probing. *Proceedings of the 2007 IEEE Symposium on Security and Privacy (S&P'07)*, 252–266.
- [20] GODEFROID, P., AND KLARLUND, N. DART: Directed Automated Random Testing. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'05)*, pp. 213–223.
- [21] GODEFROID, P., LEVIN, M. Y., AND BERKELEY, U. C. Automated Whitebox Fuzz Testing. In *Proceedings of Network and Distributed Systems Security (NDSS'08)*.
- [22] GODEFROID, P., AND LUCHAUP, D. Automatic Partial Loop Summarization in Dynamic Test Generation. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis (ISSTA'11)*, pp. 23–33.
- [23] KIEZUN, A., GANESH, V., GUO, P. J., HOOIMEIJER, P., AND ERNST, M. D. HAMPI: A Solver for String Constraints. In *Proceedings of the 18th International Symposium on Software Testing and Analysis (ISSTA'09)*, pp. 105–116.
- [24] KIM, H., AND KARP, B. Autograph: Toward Automated, Distributed Worm Signature Detection. In *Proceedings of the 13th USENIX Security Symposium (2004)*, pp. 271–286.
- [25] KING, J. C. Symbolic execution and program testing. *Commun. ACM* 19, 7 (1976), 385–394.
- [26] KUZNETSOV, V., KINDER, J., BUCUR, S., AND CANDEA, G. Efficient State Merging in Symbolic Execution. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'12)*, pp. 193–204.
- [27] LI, Z., SANGHI, M., CHAVEZ, B., CHEN, Y., AND KAO, M. Hamsa: Fast Signature Generation for Zero-day Polymorphic Worms with Provable Attack Resilience. In *Proceedings of the 2006 IEEE Symposium on Security and Privacy (S&P'06)*, pp. 32–47.
- [28] MOSER, A., KRUEGEL, C., AND KIRDA, E. Exploring Multiple Execution Paths for Malware Analysis. In *Proceedings of the 2007 IEEE Symposium on Security and Privacy (S&P'07)*, vol. 0, pp. 231–245.
- [29] NEWSOME, J., KARP, B., AND SONG, D. Polygraph: Automatically Generating Signatures for Polymorphic Worms. In *Proceedings of the 2005 IEEE Symposium on Security and Privacy (S&P'05)*, pp. 226–241.
- [30] NOTO-MONIZ, A. Software Agitation of a Dynamically Typed Language. Tech. rep., Worcester Polytechnic Institute, 2012.
- [31] QI, D., NGUYEN, H. D., AND ROYCHOUDHURY, A. Path Exploration based on Symbolic Output. In *Proceedings of the 19th ACM SIGSOFT Symposium on Foundations of Software Engineering (FSE'11)*, pp. 278–288.
- [32] RAGAN, S. Java zero-day added to blackhole exploit kit and metasploit. <http://www.securityweek.com/java-zero-day-added-blackhole-exploit-kit-and-metasploit>, visited in Aug 2012.
- [33] RAMIREZ-SILVA, E., AND DACIER, M. Empirical Study of the Impact of Metasploit-Related Attacks in 4 Years of Attack Traces. In *Proceedings of the 12th Asian Computing Science Conference on Advances in Computer Science: Computer and Network Security (ASIAN'07)*, pp. 198–211.
- [34] SAXENA, P., AKHawe, D., HANNA, S., MAO, F., MCCAMANT, S., AND SONG, D. A Symbolic Execution Framework for JavaScript. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy (S&P'10)*, pp. 513–528.

- [35] SAXENA, P., POOSANKAM, P., MCCAMANT, S., AND SONG, D. Loop-Extended Symbolic Execution on Binary Programs. In *Proceedings of the 18th International Symposium on Software Testing and Analysis (ISSTA'09)*, pp. 225–236.
- [36] SCHWARTZ, E. J., AVGERINOS, T., AND BRUMLEY, D. All You Ever Wanted to Know about Dynamic Taint Analysis and Forward Symbolic Execution (but Might Have Been Afraid to Ask). In *Proceedings of the 2010 IEEE Symposium on Security and Privacy (S&P'10)*, pp. 317–331.
- [37] SEN, K., AND AGHA, G. CUTE and jCUTE: Concolic Unit Testing and Explicit Path Model-Checking Tools. In *Proceedings of the 18th International Conference on Computer Aided Verification (CAV'06)*, pp. 419–423.
- [38] SONG, D., BRUMLEY, D., YIN, H., AND CABALLERO. Bit-Blaze: A New Approach to Computer Security via Binary Analysis. In *Proceedings of the 4th International Conference on Information Systems Security (ICISS'08)*, pp. 1–25.
- [39] SOPHOSLABS. Exploring the blackhole exploit kit. <http://nakedsecurity.sophos.com/exploring-the-blackhole-exploit-kit>, visited in Jan 2013.
- [40] XIE, T., TILLMANN, N., DE HALLEUX, J., AND SCHULTE, W. Fitness-Guided Path Exploration in Dynamic Symbolic Execution. In *IEEE/IFIP International Conference on Dependable Systems & Networks (DSN'09)*, pp. 359–368.
- [41] XIE, Y., AND AIKEN, A. Static Detection of Security Vulnerabilities in Scripting Languages. In *Proceedings of the 15th USENIX Security Symposium (2006)*, pp. 179–192.

Appendix A Example of Rule Comparison

```

1 def exploit
2 ...
3 trigger = '/ldap://localhost/%3fA%3fA%3
4 # Sending payload
5 send_request_raw({
6   'uri' => '/' + rewrite_path() + trigger +
7   shellcode(),
8   'version' => '1.0',
9   }, 2)
10 ...
11 end

```

Listing 4: The code snippet from a Metasploit attack script apache_mod_rewrite_ldap.rb

```

alert tcp any any -> any 80 (
  msg:"Metasploit apache_mod_rewrite_ldap,
  Target:[Apache 1.3/2.0/2.2],
  Behavior:[HTTP request with Vul-specific
    bytes];
  content:"GET";
  content:"/ldap|3A|//localhost/%3fA%3fA%3
    fCCCCCCCCC%3fC%3f%90";
  content:"|20|HTTP/1.0|0D 0A|Host|3A 20|";
  reference:cve,2006-3747;
  sid:5000539; rev:0;
)

```

Listing 5: One MetaSymploit Rule (MR) for an attack payload of apache_mod_rewrite_ldap.rb.

```

alert tcp $EXTERNAL_NET any -> $HOME_NET 80 (
  msg:"WEB-MISC Apache mod_rewrite buffer
    overflow attempt";
  content:"GET";
  content:"ldap|3A|";
  pcre:"/ldap\x3A\x2F\x2F[^x0A]*(%3f|\x3F)[^\x0A]*
    (%3f|\x3F)[^\x0A]*(%3f|\x3F)[^\x0A]*
    (%3f|\x3F)/smi";
  reference:cve,2006-3747;
  sid:11679; rev:5;
)

```

Listing 6: One Official Snort Rule (OR) related to the Metasploit attack script in Listing 4.

In Appendix A, we give a simple example to illustrate the comparison between an official Snort rule containing general patterns with a MetaSymploit rule containing specific patterns.

Listing 4 shows the code snippet of the exploit method in the Metasploit attack script apache_mod_rewrite_ldap.rb. The script launches the attack by sending an HTTP GET request packet that contains a special URI byte string to trigger the vulnerability. Here send_request_raw is a Metasploit HTTP output API method that is symbolically extended by MetaSymploit to dump the entire payload packet.

Listing 5 is a MetaSymploit Rule (MR) based on the attack payload composed by the script. It contains the constant byte string patterns, especially the vulnerability triggering string that can identify the specific payload packet. Listing 6 is the corresponding Official Rule (OR) based on CVE matching. It contains a regular expression (regex) pattern generalized by security analysts based on large amounts of samples.

According to the Snort rule manual, a rule can have multiple content byte string patterns. By default, given a packet, Snort searches these content patterns in order. A rule can also have one pcre regex pattern. Snort searches the entire packet for the pcre pattern.

In the example rules, the first content in both rules share the same pattern “GET”. The second content of the MR captures the triggering string, which includes the second content of the OR “ldap|3A|” as a substring. Furthermore, the second content of the MR is also matched by the general pcre regex of the OR. In addition, there is another content in the MR that captures the HTTP protocol version of the packet.

Although both rules can detect the attack payload of this script, the MR has multiple specific patterns that can precisely pinpoint the attacks launched by this script, thus having very low false-positive rate compared to the general OR. In practice, the MRs can help identify what attack scripts are used by attackers, providing a way for the defense side to profile and obtain more knowledge of the attacker side.

Towards Automatic Software Lineage Inference

Jiyong Jang, Maverick Woo, and David Brumley

{jiyongj, pooh, dbrumley}@cmu.edu

Carnegie Mellon University

Abstract

Software lineage refers to the evolutionary relationship among a collection of software. The goal of software lineage inference is to recover the lineage given a set of program binaries. Software lineage can provide extremely useful information in many security scenarios such as malware triage and software vulnerability tracking.

In this paper, we systematically study software lineage inference by exploring four fundamental questions not addressed by prior work. First, how do we automatically infer software lineage from program binaries? Second, how do we measure the quality of lineage inference algorithms? Third, how useful are existing approaches to binary similarity analysis for inferring lineage in reality, and how about in an idealized setting? Fourth, what are the limitations that any software lineage inference algorithm must cope with?

Towards these goals we build iLINE, a system for automatic software lineage inference of program binaries, and also iEVAL, a system for scientific assessment of lineage quality. We evaluated iLINE on two types of lineage—straight line and directed acyclic graph—with large-scale real-world programs: 1,777 goodware spanning over a combined 110 years of development history and 114 malware with known lineage collected by the DARPA Cyber Genome program. We used iEVAL to study seven metrics to assess the diverse properties of lineage. Our results reveal that partial order mismatches and graph arc edit distance often yield the most meaningful comparisons in our experiments. Even without assuming any prior information about the data sets, iLINE proved to be effective in lineage inference—it achieves a mean accuracy of over 84% for goodware and over 72% for malware in our data sets.

1 Introduction

Software *evolves* to adapt to changing needs, bug fixes, and feature additions [28]. As such, software lineage—the evolutionary relationship among a set of software—can be a rich source of information for a number of security questions. Indeed, the literature is replete with analyses of known or manually recovered software lineages. For example, software engineering researchers have analyzed

the histories of open source projects and the Linux kernel to understand software evolution [14, 45] as well as its effect on vulnerabilities in Firefox [33]. The security community has also studied malware evolution based upon the observation that the majority of newly detected malware are tweaked variants of well-known malware [2, 18, 20]. With over 1.1 million malware appearing daily [43], researchers have exploited such evolutionary relationships to identify new malware families [23, 31], create models of provenance and lineage [9], and generate phylogeny models based upon the notion of code similarity [22].

The wealth of existing research demonstrating the utility of software lineage immediately raises the question—“Can we infer software lineage *automatically*?”. We foresee a large number of security-related applications once this becomes feasible. In forensics, lineage can help determine software provenance. For example, if we know that a closed-source program p_A is written by author X and another program p_B is derived from p_A , then we may deduce that the author of p_B is likely to be related to X . In malware triage [2, 18, 20], lineage can help malware analysts understand trends over time and make informed decisions about which malware to analyze first. This is particularly important since the order in which the variants of a malware are captured does not necessarily mirror its evolution. In software security, lineage can help track vulnerabilities in software of which we do not have source code. For example, if we know a vulnerability exists in an earlier version of an application, then it may also exist in applications that are derived from it. Such logic has been fruitfully applied at the source level in our previous work [19]. Indeed, these and related applications are important enough that the US Defense Advanced Research Projects Agency (DARPA) is funding a \$43-million Cyber Genome program [6] to study them.

Having established that automatically and accurately infer software lineage is an important open problem, let us look at how to formalize it. Software lineage inference is the task of inferring a temporal ordering and ancestor/descendant relationships among programs. We model software lineage by a *lineage graph*:

Definition 1.1. A lineage graph $G = (N, A)$ is a directed acyclic graph (DAG) comprising a set of nodes N and a set of arcs A . A node $n \in N$ represents a program, and

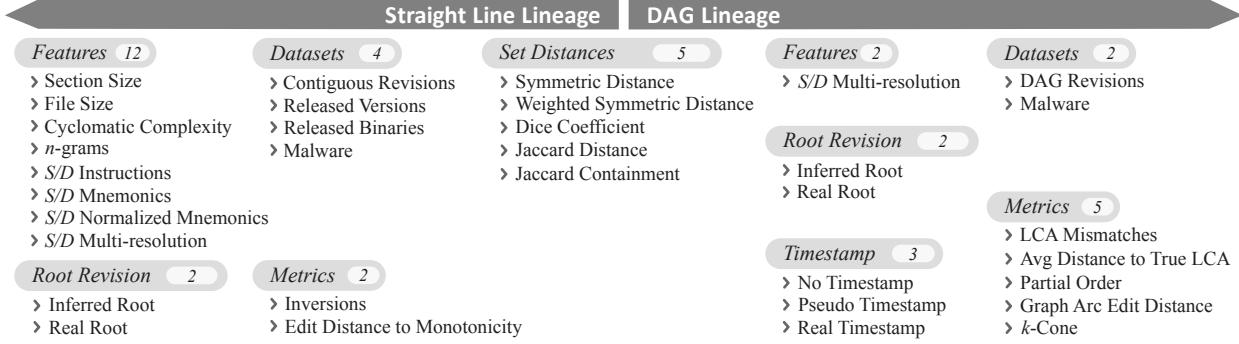


Figure 1: Design space in software lineage inference (S/D represents static/dynamic analysis-based features.)

an arc $(x, y) \in A$ denotes that program y is a derivative of program x . We say that x is a parent of y and y is a child of x .

A *root* is a node that has no incoming arc and a *leaf* is a node that has no outgoing arc. The set of *ancestors* of a node n is the set of nodes that can reach n . Note that n is an ancestor of itself. The set of *common ancestors* of x and y is the intersection of the two sets of ancestors. The set of *lowest common ancestors* (LCAs) of x and y is the set of common ancestors of x and y that are not ancestors of other common ancestors of x and y [4]. Notice that in a tree each pair of nodes must have a unique LCA, but in a DAG some pair of nodes can have multiple LCAs.

In this paper, we ask four basic research questions:

1. Can we automatically infer software lineage? Existing research focused on studying known software history and lineage [14, 33, 45], not creating lineage. Creating lineage is different from building a dendrogram based upon similarity [22, 23, 31]. A dendrogram can be used to identify families; however it does not provide any information about a temporal ordering, e.g., root identification.

In order to infer a temporal ordering and evolutionary relationships among programs, we develop new algorithms to automatically infer lineage of programs for two types of lineage: straight line lineage (§4.1) and directed acyclic graph (DAG) lineage (§4.2). In addition, we extend our approach for straight line lineage to k -straight line lineage (§4.1.4). We build iLINE to systematically evaluate the effectiveness of our lineage inference algorithms using twelve software feature sets (§2), five distance measures between feature sets (§3), two policies on the root identification (§4.1.1), and three policies on the use of timestamps (§4.2.2).

Without any prior information about data sets, for straight line lineage, the mean accuracies of iLINE are 95.8% for goodware and 97.8% for malware. For DAG lineage, the mean accuracies are 84.0% for goodware and 72.0% for malware.

2. What are good metrics? Existing research focused on building a phylogenetic tree of malware [22, 23], but did not provide quantitative metrics to scientifically measure

the quality of their output. Good metrics are necessary to quantify how good our approach is with respect to the ground truth. Good metrics also allow us to compare different approaches. To this end, we build iEVAL to assess our lineage inference algorithms using multiple metrics, each of which represents a different perspective of lineage.

iEVAL uses two metrics for straight line lineage (§5.1). Given an inferred lineage graph G and the ground truth G^* , the *number of inversions* measures how often we make a mistake when answering the question “which one of programs p_i and p_j comes first”. The *edit distance to monotonicity* asks “how many nodes do we need to remove in G so that the remaining nodes are in the sorted order (and thus respect G^*)”.

iEVAL also utilizes five metrics to measure the accuracy of DAG lineage (§5.2). An *LCA mismatch* is a generalized version of an inversion because the LCA of two nodes in a straight line is the earlier node. We also measure the *average pairwise distance between true LCA(s) and derived LCA(s) in G^** . The *partial order mismatches* in a DAG asks the same question as inversions in a straight line. The *graph arc edit distance* for (labeled) graphs measures “what is the minimum number of arcs we need to delete from G and G^* to make both graphs equivalent”. A *k -Cone mismatch* asks “how many nodes have the correct set of descendants counting up to depth k ”.

Among the above seven metrics, we recommend two metrics—partial order mismatches and graph arc edit distance. In §5.3, we discuss how the metrics are related, which metric is useful to measure which aspect of a lineage graph, which metric is efficient to compute, and which metric is deducible from other metrics.

3. How well are we doing now? We would like to understand the limits of our techniques even in *ideal* cases, meaning we have (i) control over variables affecting the compilation of programs, (ii) reliable feature extraction techniques to abstract program binaries accurately and precisely, and (iii) the ground truth with which we can compare our results to measure accuracy and to spot error cases. We discuss the effectiveness of different feature

sets and distance measures on lineage inference in §8.

We argue that it is necessary to also systematically validate a lineage inference technique with “goodware”, e.g., open source projects. Since malware is often surreptitiously developed by adversaries, it is typically hard or even impossible to obtain the ground truth. More fundamentally, we simply cannot hope to understand the evolution of adversarial programs unless we first understand the limits of our approach in our idealized setting.

We systematically evaluated iLINE with both goodware and malware that we have the ground truth on: 1,777 goodware spanning over a combined 110 years of development history and 114 malware collected by the DARPA Cyber Genome program.

4. What are the limitations? We investigate error cases in G constructed by iLINE and highlight some of the difficult cases where iLINE failed to recover the correct evolutionary relationships. Since some of our experiments are conducted on goodware with access to source code, we are able to pinpoint challenging issues that must be addressed before we can improve the accuracy in software lineage inference. We discuss such challenging issues including reverting/refactoring, root identification, clustering, and feature extraction in §9. This is important because we may not be able to understand malware evolution without understanding limits of our approach with goodware.

2 Software Features for Lineage

In this study, we use three program analysis methods: syntax-based analysis, static analysis, and dynamic analysis. Given a set of program binaries \mathcal{P} , various features f_i are extracted from each $p_i \in \mathcal{P}$ to evaluate different abstractions of program binaries. Source code or metadata such as comments, commit messages or debugging information is not used as we are interested in results in security scenarios where source code is typically not available, e.g., forensics, proprietary software, and malware.

2.1 Using Previous Observations

Previous work analyzed software release histories to understand a software evolution process. It has been often observed that program size and complexity tend to increase as new revisions are released [14, 28, 45]. This observation also carries over to security scenarios, e.g., the complexity of malware is likely to grow as new variants appear [8]. We measured code section size, file size, and code complexity to assess how useful these features are in inferring lineage of program binaries.

- **Section size:** iLINE first identifies executable sections in binary code, e.g., `.text` section, which contain executable program code, and calculates the size.
- **File size:** Besides the section size, iLINE also calculates the file size, including code and data.

```

8b5dd485db750783c42c5b5e5dc383c42c5b5e5de9adf8ffff
(a) Byte sequence of program code
8b5dd485 5dd485db d485db75 85db7507 db750783
750783c4 0783c42c 83c42c5b c42c5b5e 2c5b5e5d
5b5e5dc3 5e5dc383 5dc383c4 c383c42c 5b5e5de9
5e5de91d 5de9adf8 e9adf8ff adf8ffff
(b) 4-grams
mov -0x2c(%ebp),%ebx;test %ebx,%ebx;jne 805e198
add $0x2c,%esp;pop %ebx;pop %esi;pop %ebp;ret
add $0x2c,%esp;pop %ebx;pop %esi;pop %ebp;jmp 805da50
(c) Disassembled instructions
mov mem,reg;test reg,reg;jne imm
add imm,reg;pop reg;pop reg;pop reg;ret
add imm,reg;pop reg;pop reg;pop reg;jmp imm
(d) Instructions mnemonics with operands type
mov mem,reg;test reg,reg;jcc imm
add imm,reg;pop reg;pop reg;pop reg;ret
add imm,reg;pop reg;pop reg;pop reg;jmp imm
(e) Normalized mnemonics with operands type

```

Figure 2: Example of feature extraction

- **Cyclomatic complexity:** Cyclomatic complexity [34] is a common metric that indicates code complexity by measuring the number of linearly independent paths. From the control flow graph (CFG) of a program, the complexity M is defined as $M = E - N + 2P$ where E is the number of edges, N is the number of nodes, and P is the number of connected components of the CFG.

2.2 Using Syntax-Based Feature

While syntax-based analysis may lack semantic understanding of a program, previous work has shown its effectiveness on classifying unpacked programs. Indeed, n -gram analysis is widely adopted in software similarity detection, e.g., [20, 22, 26, 40]. The benefit of syntax-based analysis is that it is fast because it does not require disassembling.

- **n -grams:** An n -gram is a consecutive subsequence of length n in a sequence. From the identified executable sections, iLINE extracts program code into a hexadecimal sequence. Then, n -grams are obtained by sliding a window of n bytes over it. For example, Figure 2b shows 4-grams extracted from Figure 2a.

2.3 Using Static Features

Existing work utilized semantically richer features by first disassembling a binary. After reconstructing a control flow graph for each function, each basic block can be considered as a feature [12]. In order to maximize the probability of identifying similar programs, previous work also normalized disassembly outputs by considering instruction mnemonics without operands [23, 46] or instruction mnemonics with only the types of each operand (such as memory, a register or an immediate value) [39].

In our experiments, we introduce an additional normalization step of normalizing the instruction mnemonics themselves. This was motivated by our observations when

we analyzed the error cases in the lineages constructed using the above techniques. Our results indicate that this normalization notably improves lineage inference quality.

We also evaluate binary abstraction methods in an idealized setting in which we can deploy reliable feature extraction techniques. The limitation with static analysis comes from the difficulty of getting precise disassembly outputs from program binaries [27, 30]. In order to exclude the errors introduced at the feature extraction step and focus on evaluating the performance of software lineage inference algorithms, we also leverage assembly generated using `gcc -S` (not source code itself) to obtain basic blocks more accurately. Note that we use this to simulate what the results would be with ideal disassembling, which is in line with our goal of understanding the limits of the selected approaches.

- **Basic blocks comprising disassembly instructions:** iLINE disassembles a binary and identifies its basic blocks. Each feature is a sequence of instructions in a basic block. For example, in Figure 2c, each line is a series of instructions in a basic block; and each line is considered as an individual feature. This feature set is semantically richer than n -grams.

- **Basic blocks comprising instruction mnemonics:** For each disassembled instruction, iLINE retains only its mnemonic and the types of its operands (immediate, register, and memory). For example, `add $0x2c, %esp` is transformed into `add imm, reg` in Figure 2d. By normalizing the operands, this feature set helps us mitigate errors from syntactical differences, e.g., changes in offsets and jump target addresses, and register renaming.

- **Basic blocks comprising normalized mnemonics:** iLINE also normalizes mnemonics. First, mnemonics for all conditional jumps, e.g., `je`, `jne` and `jg`, are normalized into `jcc` because the same branching condition can be represented by flipped conditional jumps. For example, program p_1 uses `cmp eax, 1; jz addr_1` while program p_2 has `cmp eax, 1; jnz addr_2`. Second, iLINE removes the `nop` instruction.

2.4 Using Dynamic Features

Modern malware is often found in a packed binary format [15, 21, 32, 38] and it is often not easy to analyze such packed/obfuscated programs with static analysis tools. In order to mitigate such difficulties, dynamic analysis has been proposed to monitor program executions and changes made to a system at run time [1, 2, 13, 35]. The idea of dynamic analysis is to run a program to make it disclose its “behaviors”. Dynamic analysis on malware is typically performed in controlled environments such as virtual machines and isolated networks to prevent infections spreading to other machines [37].

- **Instructions executed at run time:** For malware specifically, iLINE traces an execution using a binary in-

strumentation tool and collects a set of instruction traces. Similar to static features, iLINE also generates additional sets of features by normalizing operands and mnemonics.

2.5 Using Multi-Resolution Features

Besides considering each feature set individually, iLINE utilizes multiple feature sets to benefit from normalized and specific features. Specifically, iLINE first uses the most normalized feature set to detect similar programs and gradually employs less-normalized feature sets to distinguish highly similar programs. This ensures that less similar programs (e.g., major version changes) will be connected only after more similar programs (e.g., only changes of constant values) have been connected.

3 Distance Measures Between Feature Sets

To measure the distance between two programs p_1 and p_2 , iLINE uses the symmetric difference between their feature sets, which captures both additions and deletions made between p_1 and p_2 . Let f_1 and f_2 denote the two feature sets extracted from p_1 and p_2 , respectively. The *symmetric distance* between f_1 and f_2 is defined to be

$$SD(f_1, f_2) = |f_1 \setminus f_2| + |f_2 \setminus f_1|, \quad (1)$$

which denotes the cardinality of the set of features that are in f_1 or f_2 but not both. The symmetric distance basically measures the number of unique features in p_1 and p_2 .

Distance metrics other than symmetric distance may be used for lineage inference as well. For example, the *Dice coefficient distance* $DC(f_1, f_2) = 1 - \frac{2|f_1 \cap f_2|}{|f_1| + |f_2|}$, the *Jaccard distance* $JD(f_1, f_2) = 1 - \frac{|f_1 \cap f_2|}{|f_1 \cup f_2|}$, and the *Jaccard containment distance* $JC(f_1, f_2) = 1 - \frac{|f_1 \cap f_2|}{\min(|f_1|, |f_2|)}$ can all be used to calculate the dissimilarity between two sets.

Besides the above four distance measures, which are all symmetric, i.e., $distance(f_1, f_2) = distance(f_2, f_1)$, we have also evaluated an *asymmetric* distance measure to determine the direction of derivation between p_1 and p_2 . We call it the *weighted symmetric distance*, denoted $WSD(f_1, f_2) = |f_1 \setminus f_2| \times C_{del} + |f_2 \setminus f_1| \times C_{add}$ where C_{del} and C_{add} denote the cost for each deletion and each addition, respectively. Note that $WSD(f_1, f_2) = SD(f_1, f_2)$ when $C_{del} = C_{add} = 1$.

Our hypothesis is that additions and deletions should have different costs in a software evolution process, and we should be able to infer the derivative direction between two programs more accurately using the weighted symmetric distance. For example, in many open source projects and malware, code size usually grows over time [8, 45]. In other words, addition of new code is preferred to deletion of existing code. Differentiating C_{del} and C_{add} can help us to decide a direction of derivation. In this paper, we set $C_{del} = 2$ and $C_{add} = 1$. (We leave it as

future work to investigate the effect of these values.) Suppose program p_i has feature set $f_i = \{m_1, m_2, m_3\}$, and program p_j contains feature set $f_j = \{m_1, m_2, m_4, m_5\}$. By introducing asymmetry, evolving from p_i to p_j has a distance of 4 (deletion of m_3 and addition of m_4 and m_5), while the opposite direction has a distance of 5 (deletion of m_4 and m_5 and addition of m_3). Since $p_i \rightarrow p_j$ has a smaller distance, we conclude that it is the more plausible scenario.

For the rest of our paper, we use SD as a representative distance metric when we explain our lineage inference algorithms. We evaluated the effectiveness of all five distance measures on inferring lineage using SD as a baseline (see §8). Regarding metric-based features, e.g., section size, we measure the distance between two samples as the difference of their metric values.

4 Software Lineage Inference

Our goal is to automatically infer software lineage of program binaries. We build iLINE to systematically explore the design space illustrated in Figure 1 to understand advantages and disadvantages of our algorithms for inferring software lineage. We applied our algorithms to two types of lineage: straight line lineage (§4.1) and directed acyclic graph (DAG) lineage (§4.2). In particular, this is motivated by the observation that there are two common development models: serial/mainline development and parallel development. In serial development, every developer makes a series of check-ins on a single branch; and this forms straight line lineage. In parallel development, several branches are created for different tasks and are merged when needed, which results in DAG lineage.

4.1 Straight Line Lineage

The first scenario that we have investigated is 1-straight line lineage, i.e., a program source tree that has no branching/merging history. This is a common development history for smaller programs. We have also extended our technique to handle multiple straight line lineages (§4.1.4).

Software lineage inference in this setting is a problem of determining a temporal ordering. Given N unlabeled revisions of program p , the goal is to output label “1” for the 1st revision, “2” for the 2nd revision, and so on. For example, if we are given 100 revisions of program p and we have no timestamp of the revisions (or 100 revisions are randomly permuted), we want to rearrange them in the correct order starting from the 1st revision p^1 to the 100th revision p^{100} .

4.1.1 Identifying the Root Revision

In order to identify the root/first revision that has no parent in lineage, we explore two different choices: (i) inferring

the root/earliest revision, and (ii) using the real root revision from the ground truth.

iLINE picks the root revision based upon Lehman’s observation [28]. The revision that has the minimum code complexity (the 2nd software evolution law) and the minimum size (the 6th software evolution law) is selected as the root revision. The hypothesis is that developers are likely to add more code to previous revisions rather than delete other developers’ code, which can increase code complexity and/or code size. This is also reflected in security scenarios, e.g., malware authors are also likely to add more modules to make it look different to bypass anti-virus detection, which leads to high code complexity [8]. In addition, provenance information such as first seen date [10] and tool-chain components [36] can be leveraged to infer the root.

We also evaluate iLINE with the real root revision given from the ground truth in case the inferred root revision was not correct. By comparing the accuracy of the lineage with the real root revision to the accuracy of the lineage with the inferred root revision, we can assess the importance of identifying the correct root revision.

4.1.2 Inferring Order

From the selected root revision, iLINE greedily picks the closest revision in terms of the symmetric distance as the next revision. Suppose we have three contiguous revisions: p^1 , p^2 , and p^3 . One hypothesis is $SD(p^1, p^2) < SD(p^1, p^3)$, i.e., the symmetric distance between two adjacent revisions would be smaller. This hypothesis follows logically from Lehman’s software evolution laws.

There may be cases where the symmetric distance between two different pairs are the same, i.e., a tie. Suppose $SD(p^1, p^2) = SD(p^1, p^3)$. Then both p^2 and p^3 become candidates for the next revision of p^1 . Using normalized features can cause more ties than using specific features because of the information loss.

iLINE utilizes more specific features in order to break ties more correctly (see §2.5). For example, if the symmetric distances using normalized mnemonics are the same, then the symmetric distances using instruction mnemonics are used to break ties. iLINE gradually reduces normalization strength to break ties.

4.1.3 Handling Outliers

As an optional step, iLINE handles outliers in our recovered ordering, if any. Since iLINE constructs lineage in a greedy way, if one revision is not selected mistakenly, the revision may not be selected until the very last round. To see this, suppose we have 5 revisions p^1 , p^2 , p^3 , p^4 , and p^5 . If iLINE falsely selects p^3 as the next revision of p^1 ($p^1 \rightarrow p^3$) and $SD(p^3, p^4) < SD(p^3, p^2)$, then p^4 will be chosen as the next revision ($p^1 \rightarrow p^3 \rightarrow p^4$). It is likely that $SD(p^4, p^5) < SD(p^4, p^2)$ holds because

p^4 and p^5 are neighboring revisions, and then p^5 will be selected ($p^1 \rightarrow p^3 \rightarrow p^4 \rightarrow p^5$). The probability of selecting p^2 is getting lower and lower if we have more revisions. At last p^2 is added as the last revision ($p^1 \rightarrow p^3 \rightarrow p^4 \rightarrow p^5 \rightarrow p^2$) and becomes an outlier.

In order to handle such outliers, iLINE monitors the symmetric distance between every adjacent pair in the constructed lineage G . Since the symmetric distance at an outlier is the accumulation of changes from multiple revisions, it would be much larger than the difference between two contiguous revisions. (See Figure 10 for a real life example.) iLINE detects outliers by detecting peaks among the symmetric distances between consecutive pairs by means of a user-configurable threshold.

Once an outlier r has been identified, iLINE eliminates it in two steps. First, iLINE locates the revision y that has the minimum distance with r . Then, iLINE places r immediately next to y , favoring the side with a gap that has a larger symmetric distance. In our example, suppose p^3 is the closest revision to p^2 . iLINE will compare $\text{SD}(p^1, p^3)$ (*before*) with $\text{SD}(p^3, p^4)$ (*after*) and then insert p^2 into the bigger of the two gaps. Therefore, in the case when $\text{SD}(p^1, p^3)$ is larger than $\text{SD}(p^3, p^4)$, we will recover the correct lineage, i.e., $p^1 \rightarrow p^2 \rightarrow p^3 \rightarrow p^4 \rightarrow p^5$.

4.1.4 k -Straight Line Lineage

We consider k -straight line lineage where we have a mixed data set of k different programs instead of a single program, and each program has straight line lineage.

For k -straight line lineage, iLINE first performs clustering on a given data set \mathcal{P} to group the same (similar) programs into the same cluster $P_k \subseteq \mathcal{P}$. Programs are similar if $D(p_i, p_j) \leq t$ where $D(\cdot)$ means a distance measurement between two programs and t is a distance threshold to be considered as a group. After we isolate distinct program groups between each other, iLINE identifies the earliest revision p_k^1 and infers straight line lineage for each program group P_k using the straight line lineage method. We denote the r -th revision of the program k as p_k^r . One caveat with the use of clustering as a preprocessing step is that more precise clustering may require reliable “components” extraction from program binaries, which is out of our scope.

Given a collection of programs and revisions, previous work shows that clustering can effectively separate them [5, 18, 20, 46]. iLINE uses hierarchical clustering because the number of variants k is not determined in advance. Other clustering methods like k -means clustering require that k is set at the beginning. iLINE groups two programs if $\text{JD}(f_1, f_2) \leq t$ where t is a distance threshold ($0 \leq t \leq 1$). In order to decide an appropriate distance threshold t , we explore entire range of t and find the value where the resulting number of clusters becomes stable (see Figure 7 for an example).

4.2 Directed Acyclic Graph Lineage

The second scenario we studied is directed acyclic graph (DAG) lineage. This generalizes straight line lineage to include branching and merging histories. Branching and merging are common in large scale software development because branches allow developers to modify and test code without affecting others.

In a lineage graph G , branching is represented by a node with more than one *outgoing* arcs, i.e., a revision with multiple children. Merging is denoted by a node with more than one *incoming* arcs, i.e., a revision with multiple parents.

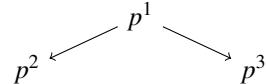
4.2.1 Identifying the Root Revision

In order to identify the root revision in lineage, we explore two different choices: (i) inferring the root/earliest revision and (ii) using the real root revision from the ground truth as discussed in §4.1.1.

4.2.2 Building Spanning Tree Lineage

iLINE builds (directed) spanning tree lineage by greedy selection. This step is similar to, but different from the ordering recovery step of the straight line lineage method. In order to recover an ordering, iLINE only allows the *last revision* in the recovered lineage G to have an outgoing arc so that the lineage graph becomes a straight line. For DAG lineage, however, iLINE allows *all revisions* in the recovered lineage G to have an outgoing arc so that a revision can have multiple children.

For example, given three revisions p^1 , p^2 , and p^3 , if p^1 is selected as a root and $\text{SD}(p^1, p^2) < \text{SD}(p^1, p^3)$, then iLINE connects p^1 and p^2 ($p^1 \rightarrow p^2$). If $\text{SD}(p^1, p^3) < \text{SD}(p^2, p^3)$ holds, p^1 will have another child p^3 and a lineage graph looks like the following:



We evaluate three different policies on the use of a timestamp in DAG lineage: *no* timestamp, the *pseudo* timestamp from the recovered straight line lineage, and the *real* timestamp from the ground truth. Without a timestamp, the revision p^j to be added to G is determined by the minimum symmetric distance $\min\{\text{SD}(p^i, p^j) : p^i \in \hat{N}, p^j \in \hat{N}^c\}$ where $\hat{N} \subseteq N$ represents a set of nodes already inserted into G and \hat{N}^c denotes a complement of \hat{N} ; and an arc (p^i, p^j) is added. However, with the use of a timestamp, the revision $p^j \in \hat{N}^c$ to be inserted is determined by the earliest timestamp and an arc is drawn based upon the minimum symmetric distance. In other words, we insert nodes in the order of timestamps.

4.2.3 Adding Non-Tree Arcs

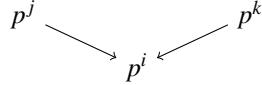
While building (directed) spanning tree lineage, iLINE identifies *branching* points by allowing the revisions $p^i \in$

\hat{N} to have more than one outgoing arcs—revisions with multiple children. In order to pinpoint *merging* points, iLINE adds non-tree arcs also known as cross arcs to spanning tree lineage.

For every non-root node p^i , iLINE identifies a unique feature set u^i that does not come from its parent p^j , i.e., $u^i = \{x : x \in f^i \text{ and } x \notin f^j\}$. Then iLINE identifies possible parents $p^k \in N$ as follows:

- i) if real/pseudo timestamps are given, p^k with earlier timestamps than the timestamp of p^i
- ii) for symmetric distance measures such as SD, DC, JD, and JC, non-ancestors p^k added to G before p^i
- iii) for the asymmetric distance measure WSD, non-ancestors p^k satisfying $\text{WSD}(p^k, p^i) < \text{WSD}(p^i, p^k)$

become possible parents. Among the identified possible parents p^k , if u^i and f^k extracted from p^k have common features, then iLINE adds a non-tree arc from p^k to p^i . Consequently, p^i becomes a merging point of p^j and p^k and a lineage graph looks like the following:



After adding non-tree arcs, iLINE outputs DAG lineage showing both branching and merging.

5 Software Lineage Metrics

We build iEVAL to scientifically measure the quality of our constructed lineage with respect to the ground truth.

5.1 Straight Line Lineage

We use dates of commit histories and version numbers as the ground truth of ordering $G^* = (N, A^*)$, and compare the recovered ordering by iLINE $G = (N, A)$ with the ground truth to measure how close G is to G^* .

iEVAL measures the accuracy of the constructed lineage graph G using two metrics: *number of inversions* and *edit distance to monotonicity* (EDTM). An inversion happens if iLINE gives a wrong ordering for a chosen pair of revisions. The total number of inversions is the number of wrong ordering for all $\binom{|N|}{2}$ pairs. The EDTM is the minimum number of revisions that need to be removed to make the remaining nodes in the lineage graph G in the correct order. The longest increasing subsequence (LIS) can be computed in G , which is the longest (not necessarily contiguous) subsequence in the sorted order. Then the EDTM is calculated by $|N| - |LIS|$, which depicts how many nodes are out-of-place in G .

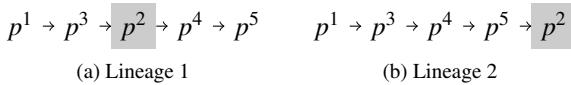


Figure 3: Inversions and edit distance to monotonicity

For example, we have 5 revisions of a program and iLINE outputs lineage 1 in Figure 3a and lineage 2 in Figure 3b. Lineage 1 has 1 inversion (a pair of $p^3 - p^2$) and 1 EDTM (delete p^2). Lineage 2 has 3 inversions ($p^3 - p^2$, $p^4 - p^2$, and $p^5 - p^2$) and 1 EDTM (delete p^2). As shown in both cases, the number of inversions can be different even when the EDTM is the same.

5.2 Directed Acyclic Graph Lineage

We evaluate the practical use of five metrics for measuring the accuracy of the constructed DAG lineage: *number of LCA mismatches*, *average pairwise distance to true LCA*, *partial order mismatches*, *graph arc edit distance*, and *k-Cone mismatches*.

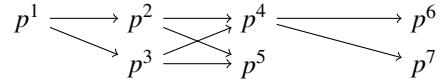


Figure 4: Lowest common ancestors

We define $\text{SLCA}(x, y)$ to be the set of LCAs of x and y because there can be multiple LCAs. For example, in Figure 4, $\text{SLCA}(p^4, p^5) = \{p^2, p^3\}$, while $\text{SLCA}(p^6, p^7) = \{p^4\}$. Given $\text{SLCA}(x, y)$ in G and the true $\text{SLCA}^*(x, y)$ in G^* , we can evaluate the correct LCA score of (x, y) $L(\text{SLCA}(x, y), \text{SLCA}^*(x, y))$ in the following four ways.

- i) 1 point (correct) if $\text{SLCA}(x, y) = \text{SLCA}^*(x, y)$
- ii) 1 point (correct) if $\text{SLCA}(x, y) \subseteq \text{SLCA}^*(x, y)$
- iii) 1 point (correct) if $\text{SLCA}(x, y) \supseteq \text{SLCA}^*(x, y)$
- iv) $1 - \text{JD}(\text{SLCA}(x, y), \text{SLCA}^*(x, y))$ point

Then the number of LCA mismatches is

$$|N \times N| - \sum_{(x,y) \in N \times N} L(\text{SLCA}(x, y), \text{SLCA}^*(x, y)).$$

The 1st policy is sound and complete, i.e., we only consider an exact match of SLCA. However, even small errors can lead to a large number of LCA mismatches. The 2nd policy is sound, i.e., every node in SLCA is indeed a true LCA (no false positive). Nonetheless, including any extra node will result in a mismatch. The 3rd policy is complete, i.e., SLCA must contain all true LCAs (no false negative). However, missing any true LCA will result in a mismatch. The 4th policy uses the Jaccard distance to measure dissimilarity between SLCA and SLCA*. In our evaluation, iLINE followed the 4th policy since it allows us to attain a more fine-grained measure.

We also measure the distance between the true LCA(s) and reported LCA(s). For example, if iLINE falsely reports p^5 as an LCA of p^6 and p^7 in Figure 4, then the pairwise distance to the true LCA is 2 (=distance between p^4 and p^5). Formally, let $D(u, v)$ represent the distance between nodes u and v in the ground truth G^* . Given $\text{SLCA}(x, y)$ and $\text{SLCA}^*(x, y)$, we define the pairwise distance to true LCA $T(\text{SLCA}(x, y), \text{SLCA}^*(x, y))$ to be

SPECIAL CASE ← → GENERAL CASE		Property measured
Straight Line	DAG	
Inversions	PO	SLCA
EDTM		GAED
		<i>k</i> -Cone

Table 1: Relationships among metrics

$$\sum_{(l,l^*) \in \text{SLCA}(x,y) \times \text{SLCA}^*(x,y)} \frac{D(l,l^*)}{|\text{SLCA}(x,y) \times \text{SLCA}^*(x,y)|}$$

and the average pairwise distance to true LCA to be

$$\sum_{(x,y) \in N \times N} \frac{T(\text{SLCA}(x,y), \text{SLCA}^*(x,y))}{|N \times N|}.$$

A partial order (PO) of x and y is to identify which one of x and y comes first: either x or y , or incomparable if they are not each other’s ancestors. For example, in Figure 4, the PO of p^3 and p^7 is p^3 , while the PO of p^6 and p^7 is incomparable. The total number of PO mismatches is the number of wrong ordering for all $\binom{|N|}{2}$ pairs.

A graph arc edit distance (GAED) measures how many arcs need to be deleted from G and G^* to make both G and G^* identical. For every node x , we calculate $E(x) = \text{SD}(\text{Adj}(x), \text{Adj}^*(x))$ where $\text{Adj}(x)$ and $\text{Adj}^*(x)$ denotes the adjacency list of x in G and G^* respectively. Then GAED becomes $\sum_{x \in N} E(x)$.

We define k -CONE(x) to be the set of descendants within depth k from node x . For example, in Figure 4, 2-CONE of p^1 is $\{p^2, p^3, p^4, p^5\}$. Then the given k -CONE(x) in G and the true k -CONE $^*(x)$ in G^* , we can evaluate the correct k -CONE score of x $R(k\text{-CONE}(x))$ using four different ways of set comparisons: an exact match, a subset match, a superset match, or the Jaccard index. In our evaluation, iLINE used the Jaccard index for a more fine-grained measure. Then the number of k -CONE mismatches is $|N| - \sum_{x \in N} R(k\text{-CONE}(x))$. With smaller k , we can measure the accuracy of nearest descendants.

5.3 Relationships among Metrics

Table 1 shows the relationships among different metrics and a property measured by each metric. A PO mismatch is a special case of an LCA mismatch because when x and y are in different branches, an LCA mismatch measures the accuracy of SLCA while a PO mismatch just says two nodes are incomparable. An inversion is also a special case of an LCA mismatch because querying the LCA of x and y in a straight line is the same as asking which one of x and y comes first. Essentially, a PO mismatch in a DAG is equal to an inversion in a straight line.

EDTM is a special case of GAED and an upper bound of GAED in a straight line is $\text{GAED} \leq \text{EDTM} \times 6$. One out-of-place node can cause up to six arcs errors. For example, $p^1 \rightarrow p^2 \rightarrow p^4 \rightarrow p^3 \rightarrow p^5$ has 1 EDTM (delete

p^3 or p^4) and 6 GAED (delete $p^2 \rightarrow p^4$, $p^4 \rightarrow p^3$, and $p^3 \rightarrow p^5$ in G and $p^2 \rightarrow p^3$, $p^3 \rightarrow p^4$, and $p^4 \rightarrow p^5$ in G^*).

A k -Cone mismatch is a *local* metric to assess the correctness of nearest descendants of nodes while the other six metrics are *global* metrics to evaluate the correctness of the order of nodes and to count out-of-place nodes/arcs.

What are good metrics? Among the seven metrics, we recommend two metrics—partial order mismatches and graph arc edit distance. PO mismatches and GAED are both desirable because they evaluate different properties of lineage and are not deducible from each other.

To see this, observe that PO mismatches and SLCA mismatches measure the same property of lineage and have similar accuracy results in our evaluation. However, PO mismatches are more efficient to compute than SLCA mismatches; moreover, PO gives an answer for a more intuitive question, “which one of these two programs comes first”. Thus, PO mismatches are preferred. Average distance to true LCA is supplementary to SLCA mismatches and so this metric is not necessary if we exclude SLCA mismatches. The number of inversions and edit distance to monotonicity can be respectively seen as special cases of PO mismatches and GAED in the case of straight line lineages. k -Cone mismatches can be extremely useful to an analyst during manual analysis, but it can be difficult to pick the right value of k automatically.

6 Implementation

iLINE is implemented using C (2.5 KLoC) and IDA Python plugin (100 LoC). We use the IDA Pro disassembler¹ to disassemble program binaries and to identify basic blocks. As discussed in §2.3, `gcc -S` output is used to compensate the errors introduced at the disassembling step. We utilize Cuckoo Sandbox² to monitor native functions, API calls and network activities of malware. On top of Cuckoo Sandbox, we use malwasm³ with pintool⁴, which allows us to obtain more fine-grained instruction level of traces. Since some kinds of malicious activities require “live” connections, we also employ INetSim⁵ to simulate various network services, e.g., web,

¹<http://www.hex-rays.com/products/ida/index.shtml>

²<http://cuckoosandbox.org/>

³<http://code.google.com/p/malwasm/>

⁴<http://software.intel.com/en-us/articles/pintool>

⁵<http://www.inetsim.org/>

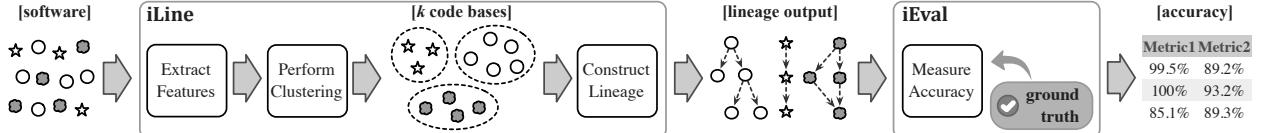


Figure 5: Software lineage inference overview

email, DNS, FTP, IRC, and so on. For example, Blaster-Worm in our data set sent exploit packets and propagated itself via TFTP only when there were (simulated) live vulnerable hosts.

For the scalability reason, we use the feature hashing technique [20, 44] to encode extracted features into bit-vectors. For example, let bv_1 and bv_2 denote two bit-vectors generated from f_1 and f_2 using feature hashing. Then the symmetric distance in Equation 1 can be calculated by:

$$SD_{bv}(bv_1, bv_2) = S(bv_1 \otimes bv_2) \quad (2)$$

where \otimes denotes bitwise-XOR and $S(\cdot)$ means the number of bits set to one.

7 Evaluation

As depicted in Figure 5, we systematically evaluated our lineage inference algorithms using (i) iLINE to explore all the design spaces described in Figure 1 with a variety of data sets and (ii) iEVAL to measure the accuracy of our outputs with respect to the ground truth.

7.1 Straight Line Lineage

7.1.1 Data sets

For straight line lineage experiments, we have collected three different kinds of goodware data sets, e.g., contiguous revisions, released versions, and actual release binaries, and malware data sets.

i) Contiguous Revisions: Using a commit history from a version control system, e.g., subversion and git, we downloaded contiguous revisions of a program. The time gap between two adjacent commits varies a lot, from <10 minutes to more than a month. We excluded some revisions that changed only comments because they did not affect the resulting program binaries.

Programs	# revisions	First rev	Last rev	Period
memcached	124	2008-10-14	2012-02-02	3.3 yr
redis	158	2011-09-29	2012-03-28	0.5 yr
redislite	89	2011-06-02	2012-01-18	0.6 yr

Table 2: Data sets of contiguous revisions

In order to set up idealized experiment environments, we compiled every revision with the same compiler and the same compiling options. We excluded variations that can come from the use of different compilers.

ii) Released Versions: We downloaded only released versions of a program meant to be distributed to end users. For example, subversion maintains them under the `tags` folder. The difference with contiguous revisions is that contiguous revisions may have program bugs (committed before testing) or experimental functionalities that would be excluded in released versions. In other words, released versions are more controlled data sets. We compiled source code with the same compiler and the same compiling options for ideal settings.

Programs	# releases	First release		Last release		Period
		Ver	Date	Ver	Date	
grep	19	2.0	1993-05-22	2.11	2012-03-02	18.8 yr
nano	114	0.7.4	2000-01-09	2.3.1	2011-05-10	11.3 yr
redis	48	1.0	2009-09-03	2.4.10	2012-03-30	2.6 yr
sendmail	38	8.10.0	2000-03-03	8.14.5	2011-05-15	11.2 yr
openssh	52	2.0.0	2000-05-02	5.9p1	2011-09-06	11.4 yr

Table 3: Data sets of released versions

iii) Actual Release Binaries: We collected binaries (not source code) of released versions from `rpm` or `deb` package files.

Programs	# files	First release		Last release		Period
		Ver	Date	Ver	Date	
grep	37	2.0-3	2009-08-02	2.11-3	2012-04-17	2.7 yr
nano	69	0.7.9-1	2000-01-24	2.2.6-1	2010-11-22	10.8 yr
redis	39	0.094-1	2009-05-06	2.4.9-1	2012-03-26	2.9 yr
sendmail	41	8.13.3-6	2005-03-12	8.14.4-2	2011-04-21	6.1 yr
openssh	75	3.9p1-2	2005-03-12	5.9p1-5	2012-04-02	7.1 yr
FileZilla	62	3.0.0	2007-09-13	3.5.3	2012-01-08	4.3 yr
p7zip	32	0.91	2004-08-21	9.20.1	2011-03-16	6.6 yr

Table 4: Data sets of actual release binaries

The difference is that we did not have any control over the compiling process of the program, i.e., different programs may be compiled with different versions of compilers and/or optimization options. This data set is a representative of real-world scenarios where we do not have any information about development environments.

iv) Malware: We used 84 samples with known lineage collected by the Cyber Genome program. The data set includes bots, worms, and Trojan horses and contains 7 clusters.

Cluster	# samples	Family	Cluster	# samples	Family
MC1	10	KBot	MC5	10	CleanRoom.B
MC2	17	BlasterWorm	MC6	15	MiniPanzer.B
MC3	15	MiniPanzer.A	MC7	10	CleanRoom.C
MC4	7	CleanRoom.A			

Table 5: Data sets of malware

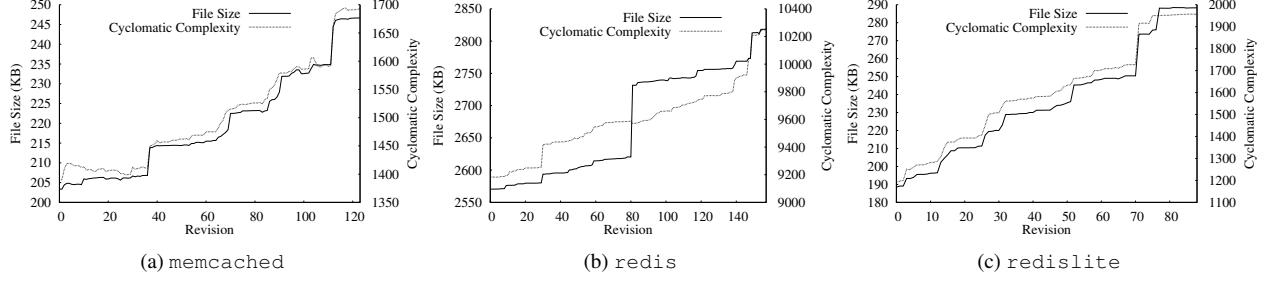


Figure 6: File size and complexity for contiguous revisions

7.1.2 Results

What selection of features provides the best lineage graph with respect to the ground truth? We evaluated different feature sets on diverse data sets.

i) Contiguous Revisions: In order to identify the first revision of each program, code complexity and code size of every revision were measured. As shown in Figure 6, both file size and cyclomatic complexity generally increased as new revisions were released. For these three data sets, the first revisions were correctly identified by selecting the revision that had the minimum file size and cyclomatic complexity.

A lineage for each program was constructed as described in §4.1. Although section/file size achieved high accuracies, e.g., 95.5%–99.5%, they are not reliable features because many ties can decrease/increase the accuracies depending on random guesses. n -grams over byte sequences generally achieved better accuracies; however, 2-grams (small size of n) were relatively unreliable features, e.g., 6.3% inversion error in *redis*. In our experiments, $n=4$ bytes worked reasonably well for these three data sets. The use of disassembly instructions had up to 5% inversion error in *redislite*. Most errors came from syntactical differences, e.g., changes in offsets and jump target addresses. After normalizing operands, instruction mnemonics with operands types decreased the errors substantially, e.g., from 5% to 0.4%. With additional normalization, normalized instruction mnemonics with operands types achieved the same or better accuracies. Note that more normalized features can result in better or worse accuracies because there may be more ties where random guesses are involved.

In order to break ties, more specific features were used in multi-resolution features. For example, all 10 tie cases in *memcached* were correctly resolved by using more specific features. This demonstrated the effectiveness of using multi-resolution features for breaking ties.

ii) Released Versions: The first/root revisions were also correctly identified by selecting the revision that had the minimum code size. In some cases, simple feature sets, e.g., section/file size, could achieve higher accuracies than semantically rich feature sets (requiring more expensive

process), e.g., instruction sequences. For example, iLINE with section size yielded 88.3% accuracy, while iLINE with instructions achieved 77.8% accuracy in *grep*. This, however, was improved to 100% with normalization. Like the experiments on contiguous revisions, 2-grams performed worse in the experiments on released versions, e.g., 18.9% accuracy in *sendmail*. Among various feature sets, multi-resolution features outperformed the other feature sets, e.g., 99.3%–100%.

iii) Actual Release Binaries: The first/root revisions for *nano* and *openssh* were correctly identified by selecting the revision that had the minimum code size. For the other five data sets, we performed the experiments both with the wrong inferred root and with the correct root given from the ground truth.

Overall accuracy of the constructed lineage was fairly high across all the data sets even though we did not control the variables of the compiling process, e.g., 83.3%–99.8% accuracy with the correct root. One possible explanation is that closer revisions (developed around the same time) might be compiled with the same version of compiler (available around the same time), which can make neighboring revisions look related to each other at the binary code level.

It was confirmed that lineage inference can be improved with the knowledge of the correct root. For example, iLINE picked a wrong revision as the first revision in *FileZilla*, which resulted in 51.6% accuracy; in contrast, the accuracy increased to 99.8% with the correct root revision.

iv) Malware: The first/root samples for all seven clusters were correctly identified by selecting the sample that had the minimum code size. Section size achieved high accuracies, e.g., 93.3%–100%, which showed new variants were likely to add more code to previous malware. File size was not a good feature to infer a lineage of MC2 because all samples in MC2 had the same file size. The multi-resolution feature yielded 94.9%–100% accuracy. Dynamic instrumentations at the instruction level enabled us to catch minor updates between two adjacent variants. For example, subsequent BlasterWorm samples add more checks for virtual environments to hide its malicious activities if it is being monitored, e.g., examin-

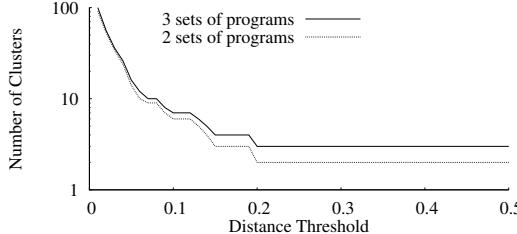


Figure 7: Clustering mixed data set of 2 and 3 programs

ing user names (sandbox, vmware, honey), running processes (VBoxService.exe, joeboxserver.exe), and current file names (C:\sample.exe). Dynamic feature sets yielded worse accuracy in MC1, MC2, MC3, MC5, and MC6 while achieving the same accuracy in MC4 and better accuracy in MC7. One main reason of the differences in accuracy is that dynamic analysis followed a specific execution path depending on the context. For example, in MC2, some variants exited immediately when they detected a VirtualBox service process, and produced limited execution traces.

v) **k -Straight Line Lineage:** We evaluated iLINE on mixed data sets including k different programs. For 2-straight line lineage, we mixed memcached and redislite in that both programs have the same functionality and similar code section sizes. Figure 7 shows the resulting number of clusters with various distance threshold values. From 0.2 to 0.5 distance threshold, the resulting number of clusters was 2. This means iLINE can first perform clustering to divide the data set into two groups, then build a straight line lineage for each group. The resulting number of clusters of the mixed data set of 3 programs including memcached, redislite, and redis became stabilized to 3 from 0.2 to 0.5 distance threshold, which means they were successfully clustered for the subsequent straight line lineage building process. We have also evaluated iLINE on three mixed malware data sets, each of which is a combination of different clusters in Table 5: {MC2+MC5}, {MC4+MC6}, and {MC2+MC3+MC7}. For each mixed data set, iLINE also clustered malware samples correctly for the subsequent straight line lineage inference. We discuss inferring lineage on incorrect clusters in §9.

7.2 Directed Acyclic Graph Lineage

7.2.1 Data sets

For DAG lineage experiments, we also evaluated iLINE on both goodware and malware.

i) **Goodware:** We have collected 10 data sets for directed acyclic graph lineage experiments from github⁶. We used github because we know *when* a project is forked from a *network graph* showing the development history as a graph including branching and merging.

We downloaded DAG revisions that had multiple times of branching and merging histories, and compiled with the same compilers and optimization options.

Programs	# revisions	First rev	Last rev	Period
http-parser	55	2010-11-05	2012-07-27	1.7 yr
libgit2	61	2012-06-25	2012-07-17	0.1 yr
redis	98	2010-04-29	2010-06-04	0.1 yr
redislite	97	2011-04-19	2011-06-12	0.1 yr
shell-fm	107	2008-10-01	2012-06-26	3.7 yr
stud	73	2011-06-09	2012-06-01	1.0 yr
tig	58	2006-06-06	2007-06-19	1.0 yr
uzbl	73	2011-08-07	2012-07-01	0.9 yr
webdis	96	2011-01-01	2012-07-20	1.6 yr
yajl	62	2010-07-21	2011-12-19	1.4 yr

Table 6: Goodware data sets for DAG lineage

ii) **Malware:** We used two malware families with known DAG lineage collected by the Cyber Genome program. They contain 30 samples in total.

Cluster	# samples	Family
MC8	21	WormBot
MC9	9	MinBot

Table 7: Malware data sets for DAG lineage

7.2.2 Results

We set two policies for DAG lineage experiments: the use of timestamp (none/pseudo/real) and the use of the real root (none/real). The real timestamp implies the real root so that we explored $3 \times 2 - 1 = 5$ different setups. We used multi-resolution feature sets for DAG lineage experiments because multi-resolution feature sets attained the best accuracy in constructing straight line lineage.

i) **Goodware:** Without having any prior knowledge, iLINE achieved 71.5%–94.1% PO accuracies. By using the real root revision, the accuracies increased to 71.5%–96.1%. For example, in case of tig, iLINE gained about 20% increase in the accuracy.

With pseudo timestamps, accuracies were worse even with the real root revisions for most of data sets, e.g., 64.0%–90.9% (see §8). By using the real timestamps, iLINE achieved higher accuracies of 84.1%–96.7%. This means that the recovered DAG lineages were very close to the true DAG lineages.

ii) **Malware:** iLINE achieved 68.6%–75.0% accuracies without any prior knowledge. Using the correct timestamps, the accuracies increased notably to 86.2%–91.7%. While we obtained the real timestamps from the ground truth in our experiments, we can also leverage first seen date of malware, e.g., Symantec’s Worldwide Intelligence Network Environment [10].

With dynamic features, iLINE achieved 59.0%–75.0% accuracies without any prior knowledge, and 68.6%–80.6% accuracies with real timestamps, which is a bit lower than the accuracies based upon static features.

⁶<https://github.com/>

7.3 Performance

Given N binaries with their features already extracted, the complexity of constructing lineage is $O(N^2)$ due to the computation of the $\binom{|N|}{2}$ pairwise distances. To give concrete values, we measured the time to construct lineage with multi-resolution features, SD, and 32 KB of bit-vectors on a Linux 3.2.0 machine with a 3.40 GHz i7 CPU utilizing a single core. Depending on the size of the data sets, it took 0.002–1.431s for straight line lineage and 0.005–0.385s for DAG lineage with the help of feature hashing. On average, this translates to 146 samples/s and 180 samples/s for straight line lineage and DAG lineage, respectively. As a comparison, our BitShred malware clustering system [20], which represents the state of the art at the time of its publication in 2011, can process 257 samples per second using a single core on the same machine. Since the running times of malware clustering and lineage inference are both dominated by distance comparisons, and since iLINE needs to resolve ties using multi-resolution features whereas BitShred needs not, we conclude that our current implementation of iLINE is competitive in terms of performance.

8 Discussion & Findings

Features. File/section size features yielded 94.6–95.5% mean accuracy in straight line lineage on goodware. Such high accuracy supports Lehman’s laws of software evolution, e.g., continuing growth. However, size is not a reliable feature to infer malware lineage where malware authors can obfuscate a feature, e.g., samples with the same file size in MC2. As simple syntactic features, 4/8/16-grams achieved 95.3–96.3% mean accuracy in straight line lineage on goodware, whereas 2-grams achieved only 82.4% mean accuracy. This is because 2-grams are not distinctive enough to differentiate between samples and cause too many ties. Basic blocks as semantic features achieved 94.0–95.6% mean accuracy in straight line lineage on goodware. This slightly lower accuracy when compared to n -grams was due to ties. Multi-resolution features performed best, e.g., it achieved 95.8–98.4% mean accuracy in straight line lineage on goodware. This is due to its use of both syntactic and semantic features.

Distance Metrics. Our evaluation indicates that our lineage inference algorithms perform similarly regardless of the distance metrics except for the Jaccard containment (JC) distance. JC turns out to be inappropriate for lineage inference because it cannot capture evolutionary changes effectively. Suppose there are three contiguous revisions p^1 , p^2 , and p^3 ; and p^2 adds 10 lines of code to p^1 and p^3 adds 10 lines of code to p^2 . Then, $JC(p^1, p^2) = JC(p^1, p^3) = JC(p^2, p^3) = 0$ because one revision is a subset of another revision. Such ties result

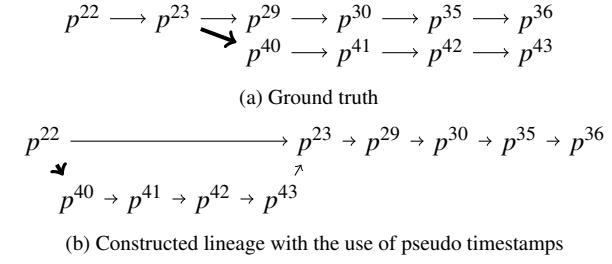


Figure 8: Error caused by pseudo timestamps in uzb1

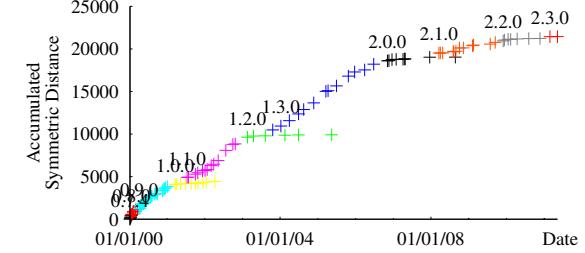


Figure 9: Development history of nano

in low accuracy. For example, JC yielded 74.5% mean accuracy, whereas SD yielded 84.0% mean accuracy in DAG lineage on goodware.

Pseudo Timestamp. iLINE computes pseudo timestamps by first building a straight line lineage and then use the recovered ordering as timestamps. Since iLINE achieved fairly high accuracy in straight line lineage, at first we expected this approach to do well in DAG lineage. To our initial surprise, iLINE with pseudo timestamps actually performed worse. In retrospect, we observed that since each branch had been developed separately, it is challenging to determine the precise ordering between samples from different branches. For example, Figure 8 shows the partial ground truth and the constructed lineage by iLINE for uzb1 with pseudo timestamps. Although iLINE *without* pseudo timestamps successfully recovered the ground truth lineage, the use of pseudo timestamps resulted in poor performance. The recovered ordering, i.e., pseudo timestamps were $p^{22}, p^{40}, p^{41}, p^{42}, p^{43}, p^{23}, p^{29}, p^{30}, p^{35}, p^{36}$. Due to the imprecise timestamps, the derivative relationships in the constructed lineage were not accurate.

Revision History vs. Release Date. Correct software lineage inference on a *revision history* may not correspond with software *release date* lineage. For example, Figure 9 shows the accumulated symmetric distance between two neighboring releases where a development branch of nano-1.3 and a stable branch of nano-1.2 are developed in parallel. iLINE infers software lineage consistent with a revision history.

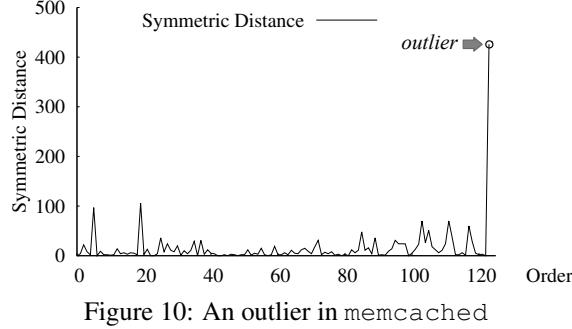


Figure 10: An outlier in memcached

Threats to Validity. Our malware experiments were performed on a relatively small data set because of difficulties in obtaining the ground truth. Although it is hard to indicate a representative of modern malware due to its surreptitious nature, we evaluated our methods on common malware categories such as bots, worms, and Trojan horses. To the best of our knowledge, we are the first to take a systematic approach towards software lineage inference to provide scientific evidence instead of speculative remarks.

9 Limitations

Reverting/Refactoring. Regression of code is a challenging problem in software lineage inference. A revision adding new functionalities is sometimes followed by stabilizing phases including bug fixes. Bug fixes might be done by reverting to the previous revision, i.e., undoing the modifications of the code.

Some revisions can become outliers because of iLINE’s greedy construction and reverting/refactoring issues. In §4.1.3, we propose a technique to detect and process outliers by looking for peaks of the distance between two contiguous revisions. For example, iLINE had 70 inversions and 1 EDTM for the contiguous revisions of memcached. The error came from the 53rd revision that was incorrectly located at the end of the lineage. Figure 10 shows the symmetric distance between two adjacent revisions in the recovered lineage before we process outliers. The outlier caused an exceptional peak of the symmetric distance at the rightmost of the Figure 10. iLINE identified such possible outliers by looking for peaks, then generated the perfect lineage of memcached after handling the outlier.

There can also be false positives among detected outliers, i.e., a peak is identified even revisions are in the correct order. For example, a peak can be identified between two contiguous revisions when there is a huge update like major version changes. However, such false positives do not affect overall accuracy of iLINE because the original (correct) position will be chosen again when minimizing the overall distance.

Although our technique improves lineage inference, it may not be able to resolve every case. Unless we design a

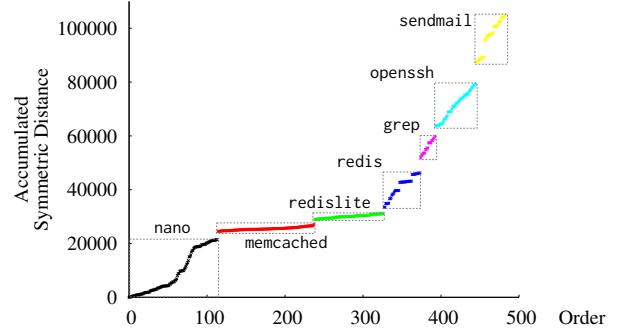


Figure 11: Recovered ordering of mixed data set

precise model describing the developers’ reverting/refactoring activity, *no* reasonable algorithm may be able to recover the same lineage as the ground truth. Rather, the constructed lineage can be considered as a more practical/pragmatic representation of the truth.

Root Identification. It is a challenging problem to identify the correct roots of data sets where we do not have any knowledge about the compilation process. iLINE successfully identified the correct roots based upon code size and complexity in all data sets except for some data sets of actual release binaries. This shows that the Lehman’s laws of software evolution are generally applicable to root identification, but with a caveat. For example, with actual release binaries data sets, iLINE achieved 77.8% mean accuracy with the inferred roots. The accuracy increased to 91.8% with the knowledge of the correct first revision.

In order to improve lineage inference, we can leverage “first seen” date of malware, e.g., Symantec’s Worldwide Intelligence Network Environment [10] or tool-chain provenance such as compilers and compilation options [36].

Clustering. Clustering may not be able to group program accurately due to noise or algorithmic limitations. In order to simulate cases where clustering failed, we mixed binaries from seven programs including memcached, redis, redislite, grep, nano, sendmail, and openssh into one set and ran our lineage inference algorithm on it. As shown in Figure 11, revisions from each program group located next to each other in the recovered order (each program is marked in a different color). This shows iLINE can identify close relationships within the same program group even with high noise in a data set. There are multiple *intra*-program gaps and *inter*-program gaps. Relatively big intra-program gaps corresponded to major version changes of a program where the Jaccard distances were 0.28–0.66. The Jaccard distances at the inter-program gaps were much higher, e.g., 0.9–0.95. This means we can separate the mixed data set into different program groups based on the inter-program gaps.

Feature Extraction. Although iLINE achieved an overall 95.8% mean accuracy in straight line lineage of goodware, iLINE achieved only 77.8% mean accuracy with actual released binaries. In order to improve lineage inference, future work may choose to leverage better features. For example, we may use recovered high-level abstraction of program binaries [41], or we may detect similar code that was compiled with different compilers and optimization options [24].

10 Related Work

While previous research focuses on studying known software lineage or development history, our focus is on designing algorithms to create lineage and evaluating metrics to assess the quality of constructed lineage.

Belady and Lehman studied software evolution of IBM OS/360 [3], and Lehman and Ramil formulated eight laws describing software evolution process [28]. Xie et al. analyzed histories of open source projects in order to verify Lehman’s laws of software evolution [45], and Godfrey and Tu investigated the Linux kernel to understand a software evolution process in open source development systems [14]. Shihab et al. evaluated the effects of branching in software development on software quality with Windows Vista and Windows 7 [42]. Kim et al. studied the history of code clones to evaluate the effectiveness of refactoring on software improvement with respect to clones [25].

Massacci et al. studied the effect of software evolution, e.g., patching and releasing new versions, on vulnerabilities in Firefox [33], and Jang et al. proposed a method to track known vulnerabilities in modern OS distributions [19]. Edwards and Chen statistically verified that an increase of security issues identified by a source code analyzer in a new release may indicate an increase of exploitable bugs in a release [11]. Davies et al. proposed a signature-based matching of a binary against a known library repository to identify library version information, which can be potentially used for security vulnerabilities scans [7].

Gupta et al. studied malware metadata collected by an anti-virus vendor to describe evolutionary relationships among malware [16]. Dumitras and Neamtiu studied malware evolution to find new variants of well-known malware [9]. Karim et al. generated phylogeny models based upon code similarity to understand how new malware related to previously seen malware [22]. Khoo and Lio investigated FakeAV-DO and Skyhoo malware families using phylogenetic methods to understand statistical relationships and to identify families [23]. Ma et al. studied diversity of exploits used by notorious worms and constructed dendograms to identify families and found non-trivial code sharing among different families [31]. Lindorfer et al. investigated the malware evolution process

by comparing subsequent versions of malware samples that were collected by exploiting embedded auto-update functionality [29]. Hayes et al. pointed out the necessity of systematic evaluation in malware phylogeny systems and proposed two models to artificially generate reference sets of samples: mutation-based model and feature accretion-based model [17].

11 Conclusion

In this paper, we proposed new algorithms to infer software lineage of program binaries for two types of lineage: straight line lineage and directed acyclic graph (DAG) lineage. We built iLINE to systematically explore the entire design space depicted in Figure 1 for software lineage inference and performed over 2,000 different experiments on large scale real-world programs—1,777 goodware spanning over a combined 110 years of development history and 114 malware with known lineage. We also built iEVAL to scientifically measure lineage quality with respect to the ground truth. Using iEVAL, we evaluated seven different metrics to assess diverse properties of lineage, and recommended two metrics—partial order mismatches and graph arc edit distance. We showed iLINE effectively extracted evolutionary relationships among program binaries with over 84% mean accuracy for goodware and over 72% for malware.

12 Acknowledgment

We would like to thank our shepherd Fabian Monroe for his support in finalizing this paper. We also would like to thank the anonymous reviewers for their insightful comments. This material is based upon work supported by Lockheed Martin and DARPA under the Cyber Genome Project grant FA975010C0170. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of Lockheed Martin or DARPA.

References

- [1] M. Bailey, J. Oberheide, J. Andersen, F. J. Z. Morley Mao, and J. Nazario. Automated classification and analysis of internet malware. In *International Symposium on Recent Advances in Intrusion Detection*, September 2007.
- [2] U. Bayer, P. M. Comparetti, C. Hlauschek, C. Kruegel, and E. Kirda. Scalable, behavior-based malware clustering. In *Network and Distributed System Security Symposium*, 2009.
- [3] L. A. Belady and M. M. Lehman. A model of large program development. *IBM Systems Journal*, 15(3):225–252, 1976.
- [4] M. Bender, M. Farach-Colton, G. Pemmasani, S. Skiena, and P. Sumazin. Lowest common ancestors in trees and directed acyclic graphs. *Journal of Algorithms*, 57(2):75–94, 2005.
- [5] zynamics BinDiff. <http://www.zynamics.com/bindiff.html>. Page checked 5/23/2013.
- [6] DARPA-BAA-10-36, Cyber Genome Program. <https://www.fbo.gov/spg/ODA/DARPA/CMO/DARPA-BAA-10-36/listing.html>. Page checked 5/23/2013.

- [7] J. Davies, D. M. German, M. W. Godfrey, and A. Hindle. Software berillonage: finding the provenance of an entity. In *Working Conference on Mining Software Repositories*, New York, New York, USA, 2011.
- [8] F. de la Cuadra. The genealogy of malware. *Network Security*, 2007(4):17–20, 2007.
- [9] T. Dumitras and I. Neamtiu. Experimental challenges in cyber security: a story of provenance and lineage for malware. In *Cyber Security Experimentation and Test*, 2011.
- [10] T. Dumitras and D. Shou. Toward a standard benchmark for computer security research: the worldwide intelligence network environment (WINE). In *Building Analysis Datasets and Gathering Experience Returns for Security*, 2011.
- [11] N. Edwards and L. Chen. An historical examination of open source releases and their vulnerabilities. In *ACM Conference on Computer and Communications Security*, 2012.
- [12] H. Flake. Structural comparison of executable objects. In *IEEE Conference on Detection of Intrusions, Malware, and Vulnerability Assessment*, 2004.
- [13] M. Fredrikson, S. Jha, M. Christodorescu, R. Sailer, and X. Yan. Synthesizing Near-Optimal Malware Specifications from Suspicious Behaviors. In *IEEE Symposium on Security and Privacy*, 2010.
- [14] M. W. Godfrey and Q. Tu. Evolution in open source software: A case study. In *International Conference on Software Maintenance*, 2000.
- [15] F. Guo, P. Ferrie, and T.-C. Chiueh. A study of the packer problem and its solutions. In *International Symposium on Recent Advances in Intrusion Detection*, 2008.
- [16] A. Gupta, P. Kuppili, A. Akella, and P. Barford. An empirical study of malware evolution. In *International Communication Systems and Networks and Workshops*, 2009.
- [17] M. Hayes, A. Walenstein, and A. Lakhotia. Evaluation of malware phylogeny modelling systems using automated variant generation. *Journal in Computer Virology*, 5(4):335–343, July 2008.
- [18] X. Hu, T. Chiueh, and K. G. Shin. Large-scale malware indexing using function call graphs. In *ACM Conference on Computer and Communications Security*, 2009.
- [19] J. Jang, A. Agrawal, and D. Brumley. ReDeBug: finding unpatched code clones in entire os distributions. In *IEEE Symposium on Security and Privacy*, 2012.
- [20] J. Jang, D. Brumley, and S. Venkataraman. BitShred: feature hashing malware for scalable triage and semantic analysis. In *ACM Conference on Computer and Communications Security*, 2011.
- [21] M. G. Kang, P. Poosankam, and H. Yin. Renovo: A hidden code extractor for packed executables. In *ACM Workshop on Rapid Malcode*, 2007.
- [22] M. E. Karim, A. Walenstein, A. Lakhotia, and L. Parida. Malware phylogeny generation using permutations of code. *Journal in Computer Virology*, 1:13–23, 2005.
- [23] W. M. Khoo and P. Lio. Unity in diversity: Phylogenetic-inspired techniques for reverse engineering and detection of malware families. In *SysSec Workshop*, 2011.
- [24] W. M. Khoo, A. Mycroft, and R. Anderson. Rendezvous: A search engine for binary code. In *Working Conference on Mining Software Repositories*, 2013.
- [25] M. Kim, V. Sazawal, D. Notkin, and G. C. Murphy. An empirical study of code clone genealogies. In *European software engineering conference - Foundations of software engineering*, 2005.
- [26] J. Z. Kolter and M. A. Maloof. Learning to detect and classify malicious executables in the wild. *Journal of Machine Learning Research*, 7:2721–2744, 2006.
- [27] C. Kruegel, W. Robertson, F. Valeur, and G. Vigna. Static disassembly of obfuscated binaries. In *USENIX Security Symposium*, 2004.
- [28] M. M. Lehman and J. F. Ramil. Rules and tools for software evolution planning and management. *Annals of Software Engineering*, 11(1):15–44, 2001.
- [29] M. Lindorfer, A. Di Federico, F. Maggi, P. M. Comparetti, and S. Zanero. Lines of malicious code: insights into the malicious software industry. In *Annual Computer Security Applications Conference*, 2012.
- [30] C. Linn and S. Debray. Obfuscation of executable code to improve resistance to static disassembly. In *ACM Conference on Computer and Communications Security*, 2003.
- [31] J. Ma, J. Dunagan, H. J. Wang, S. Savage, and G. M. Voelker. Finding diversity in remote code injection exploits. In *ACM SIGCOMM on Internet Measurement*, 2006.
- [32] L. Martignoni, M. Christodorescu, and S. Jha. OmniUnpack: fast, generic, and safe unpacking of malware. In *Annual Computer Security Applications Conference*, 2007.
- [33] F. Massacci, S. Neuhaus, and V. H. Nguyen. After-life vulnerabilities: a study on firefox evolution, its vulnerabilities, and fixes. In *International Conference on Engineering Secure Software and Systems*, 2011.
- [34] T. J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, SE-2(4):308–320, 1976.
- [35] K. Rieck, P. Trinius, C. Willem, and T. Holz. Automatic analysis of malware behavior using machine learning. *Journal of Computer Security*, 19(4):639–668, 2011.
- [36] N. Rosenblum, B. P. Miller, and X. Zhu. Recovering the toolchain provenance of binary code. In *International Symposium on Software Testing and Analysis*, 2011.
- [37] C. Rossow, C. J. Dietrich, C. Grier, C. Kreibich, V. Paxson, N. Pohlmann, H. Bos, and M. V. Steen. Prudent Practices for Designing Malware Experiments: Status Quo and Outlook. In *IEEE Symposium on Security and Privacy*, 2012.
- [38] 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*, December 2006.
- [39] A. Sæbjørnsen, J. Willcock, T. Panas, D. Quinlan, and Z. Su. Detecting code clones in binary executables. In *International Symposium on Software Testing and Analysis*, 2009.
- [40] S. Schleimer, D. Wilkerson, and A. Aiken. Winnowing: Local algorithms for document fingerprinting. In *ACM SIGMOD/PODS Conference*, 2003.
- [41] E. J. Schwartz, J. Lee, M. Woo, and D. Brumley. Native x86 decompilation using semantics-preserving structural analysis and iterative control-flow structuring. In *USENIX Security Symposium*, 2013.
- [42] E. Shihab, C. Bird, and T. Zimmermann. The effect of branching strategies on software quality. In *ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, 2012.
- [43] Symantec. Symantec internet security threat report, volume 17. <http://www.symantec.com/threatreport/>, 2012. Page checked 5/23/2013.
- [44] K. Weinberger, A. Dasgupta, J. Langford, A. Smola, and J. Attenberg. Feature hashing for large scale multitask learning. In *International Conference on Machine Learning*, 2009.
- [45] G. Xie, J. Chen, and I. Neamtiu. Towards a better understanding of software evolution: An empirical study on open source software. In *IEEE International Conference on Software Maintenance*, 2009.
- [46] Y. Ye, T. Li, Y. Chen, and Q. Jiang. Automatic malware categorization using cluster ensemble. In *ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2010.

A Appendix

A.1 Straight Line Lineage

Distance Metric	Features	Mean accuracy with the <i>inferred</i> root		Mean accuracy with the <i>real</i> root	
		Inversion Accuracy	ED	Inversion Accuracy	ED
SD	Multi	95.8%	8.6	98.4%	6.0
WSD		95.4%	9.0	98.1%	6.7
DC		93.7%	9.7	97.1%	8.4
JD		93.7%	9.7	97.1%	8.4
JC		93.0%	12.2	97.1%	9.1

Table 8: Mean accuracy for straight line lineage on goodware

Distance Metric	Features	Mean accuracy with the <i>inferred</i> root (=real root)	
		Inversion Accuracy	ED
SD	Static	97.8%	0.9
WSD		94.2%	1.3
DC		98.2%	0.9
JD		98.2%	0.9
JC		84.3%	3.1
SD	Dynamic	86.7%	2.6
WSD		80.0%	2.9
DC		85.5%	2.9
JD		85.5%	2.9
JC		70.9%	4.1

Table 9: Mean accuracy for straight line lineage on malware

A.2 DAG Lineage

Distance Metric	Features	Mean accuracy with <i>no</i> prior information		Mean accuracy with <i>real</i> timestamp	
		PO Accuracy	GAED	PO Accuracy	GAED
SD	Multi	84.0%	52.4	91.1%	20.3
WSD		82.6%	57.3	90.0%	23.0
DC		83.8%	56.1	91.1%	20.0
JD		83.8%	56.1	91.1%	20.0
JC		74.5%	90.0	90.6%	35.0

Table 10: Mean accuracy for DAG lineage on goodware

Distance Metric	Features	Mean accuracy with <i>no</i> prior information		Mean accuracy with <i>real</i> timestamp	
		PO Accuracy	GAED	PO Accuracy	GAED
SD	Static	69.5%	8.5	87.0%	6.0
WSD		72.0%	8.5	90.2%	5.5
DC		69.5%	8.5	87.0%	6.0
JD		69.5%	8.5	87.0%	6.0
JC		50.8%	19.5	86.6%	9.5
SD	Dynamic	61.4%	17.0	70.3%	13.0
WSD		62.2%	17.0	76.4%	12.5
DC		59.8%	19.0	72.8%	12.5
JD		59.8%	19.0	72.8%	12.5
JC		55.3%	17.5	72.8%	12.5

Table 11: Mean accuracy for DAG lineage on malware

Securing Embedded User Interfaces: Android and Beyond

Franziska Roesner and Tadayoshi Kohno
University of Washington

Abstract

Web and smartphone applications commonly embed third-party user interfaces like advertisements and social media widgets. However, this capability comes with security implications, both for the embedded interfaces and the host page or application. While browsers have evolved over time to address many of these issues, mobile systems like Android—which do not yet support true cross-application interface embedding—present an opportunity to redesign support for secure embedded user interfaces from scratch. In this paper, we explore the requirements for a system to support secure embedded user interfaces by systematically analyzing existing systems like browsers, smartphones, and research systems. We describe our experience modifying Android to support secure interface embedding and evaluate our implementation using case studies that rely on embedded interfaces, such as advertisement libraries, Facebook social plugins (e.g., the “Like” button), and access control gadgets. We provide concrete techniques and reflect on lessons learned for secure embedded user interfaces.

1 Introduction

Modern Web and smartphone applications commonly embed third-party content within their own interfaces. Websites embed iframes containing advertisements, social media widgets (e.g., Facebook’s “Like” or Twitter’s “tweet” button), Google search results, or maps. Smartphone applications include third-party libraries that display advertisements or provide billing functionality.

Including third-party content comes with potential security implications, both for the embedded content and the host application. For example, a malicious host may attempt to eavesdrop on input intended for embedded content or forge a user’s intent to interact with it, either by tricking the user (e.g., by clickjacking) or by programmatically issuing input events. On the other hand, a malicious embedded principal may, for example, attempt to take over a larger display area than expected.

The Web security model has evolved over time to address these and other threats. For example, the same-origin policy prevents embedded content from directly accessing or manipulating the parent page, and vice versa. As recently as 2010, browsers have added the `sandbox` attribute for iframes [1], allowing websites to

prevent embedded content from running scripts or redirecting the top-level page. However, other attacks—like clickjacking—remain a serious concern. Malicious websites frequently mount “likejacking” attacks [24] on the Facebook “Like” button, in which they trick users into sharing the host page on their Facebook profiles. If Facebook suspects a button of being part of such an attack, it asks the user to confirm any action in an additional popup dialog [7]—in other words, Facebook falls back on a non-embedded interface to compensate for the insecurity of embedded interfaces.

While numerous research efforts have attempted to close the remaining security gaps with respect to interface embedding on the Web [11, 25, 29], they struggle with maintaining backwards compatibility and are burdened with the complexity of the existing Web model. We argue that Android, which to date offers no cross-application embedding, offers a compelling opportunity to redesign secure embedded interfaces from scratch.

Today, applications on Android and other mobile operating systems cannot embed interfaces from another principal; rather, they include third-party libraries that run in the host application’s context and provide custom user interface elements (such as advertisements). On the one hand, these libraries can thus abuse the permissions of or otherwise take advantage of their host applications. On the other hand, interface elements provided by these libraries are vulnerable to manipulation by the host application. For example, Android applications can programmatically click on embedded ads in an attempt to increase their advertising revenue [18]. This lack of security also precludes desirable functionality from the Android ecosystem. For example, the social plugins that Facebook provides on the Web (e.g., the “Like” button or comments widget) are not available on Android.

Previous research efforts for Android [17, 23] have focused only on one interface embedding scenario: advertising. As a result, these systems, while valuable, do not provide complete or generalizable solutions for interface embedding. For example, to our knowledge, no existing Android-based solution prevents a host application from eavesdropping on input to an embedded interface.

In this paper, we explore what it takes to support secure embedded UIs on Android. We systematically analyze existing systems, including browsers, with respect

to whether and how they provide a set of security properties. We view this analysis and the framework we use for it as a contribution in its own right. Informed by this analysis, we describe our experiences modifying the Android framework to support cross-principal interface embedding in a way that meets our security goals. We evaluate our implementation using case studies that rely on embedded interfaces, including: (1) advertisement libraries that run in a separate process from the embedding application, (2) Facebook social plugins, to date available only on the Web, and (3) access control gadgets [19] that allow applications to access sensitive resources (like geolocation) only in response to real user input.

Through our implementation experience, we consolidate and evaluate approaches from prior work. We find that some techniques can be simplified in practice—such as an approach for maintaining invariants in the UI layout tree [18]—but that we face additional practical challenges, like propagating layout changes across processes. We discover that an embedded element’s size is an important factor in preventing clickjacking, as well as that we can apply prior work on access control gadgets [19] in novel ways to improve interaction flexibility beyond the browser model. We discuss these and other challenges and lessons in more detail in Section 8, which benefits from the context of the preceding sections.

Today’s system developers wishing to support secure embedded user interfaces have no systematic set of techniques or criteria upon which they can draw. Short of simply adopting the Web model by directly extending an existing browser—which may be undesirable for many reasons, including the need to maintain backwards compatibility with the existing Web ecosystem and programming model—system developers must (1) reverse-engineer existing techniques used by browsers, and (2) evaluate and integrate research solutions that address remaining issues. In addition to presenting the first secure interface embedding solution for Android, this paper provides a concrete, comprehensive, and system-independent set of criteria and techniques for supporting secure embedded user interfaces.

2 Motivation and Background

To motivate the need for secure embedded user interfaces, we describe (1) the functionality enabled by embedded applications and interfaces, and (2) the security concerns associated with this embedding. We argue that interface embedding often increases the usability of a particular interaction—embedded content is shown in context, and users can interact with multiple principals in one view—but that security concerns associated with cross-principal UI embedding lead to designs that are more disruptive to the user experience (e.g., prompts).

2.1 Functionality

Third-Party Applications. Web and smartphone applications often embed or redirect to user interfaces from other sources. Common use cases include third-party advertisements and social sharing widgets (e.g., Facebook’s “Like” button or comment feed, Google’s “+1” button, or a Twitter feed). Other examples of embeddable content include search boxes and maps.

On the Web, content embedding is done using HTML tags like `iframe` or `object`. On smartphone operating systems like iOS and Android, however, applications cannot directly embed UI from other applications but rather do one of two things: (1) launch another application’s full-screen view (via an Android Intent or an iOS `RemoteViewController` [2]) or (2) include a library that provides embeddable UI elements in the application’s own process. The former is generally used for sharing actions (e.g., sending an email) and the latter is generally used for embedded advertisements and billing.

System UI. Security-sensitive actions often elicit system interfaces, usually in the form of prompts. For example, Windows users are shown a User Account Control dialog [14] when an application requires elevation to administrative privileges, and iOS and browser users must respond to a permission dialog when an application attempts to access a sensitive resource (e.g., geolocation).

Because prompts result in a disruptive user experience, the research community has explored using embedded system interfaces to improve the usability of security-sensitive interactions like resource access. In particular, a recent paper [19] describes access control gadgets (ACGs), embeddable UI elements that—with user interaction—grant applications access to various system resources, including the camera, the clipboard, the GPS, etc. For example, an application might embed a location ACG, which is provided by the system and displays a recognizable location icon; when the user clicks the ACG, the embedding application receives the current GPS coordinates. As we describe below, ACGs cannot be introduced into most of today’s systems without significant changes to those systems.

2.2 Threat Model and Security Concerns

We consider user interfaces composed of elements from different, potentially mutually distrusting principals (e.g., a host application and an embedded advertisement or an embedded ACG). Host principals may attempt to manipulate interface elements embedded from another principal, and embedded principals may attempt to manipulate those of their host. We assume that the system itself is trustworthy and uncompromised.

We observe that while Web and smartphone applications rely heavily on third-party content and services, the associated third-party user interface is not always actu-

ally embedded inside of the client application. For example, websites redirect users to PayPal’s full-screen page, OAuth authorization dialogs appear in pop-up or redirect windows, and Web users who click on a Facebook “Like” button that is suspected of being part of a clickjacking attack will see an additional pop-up confirmation dialog. We observe two main security-related reasons for the choice not to embed or not to be embedded.

One reason is concern about phishing. If users become accustomed to seeing embedded third-party login or payment forms, they may become desensitized to their existence. Further, because users cannot easily determine the origin (or presence) of embedded content, malicious applications may try to trick users into entering sensitive information into spoofed embedded forms (a form of phishing). Thus, legitimate security-sensitive forms are often shown in their own tab or window.

Our goal in this paper is *not* to address such phishing attacks, but rather to evaluate and implement methods for securely embedding one legitimate (i.e., not spoofed) application within another. (While extensions of existing approaches, such as SiteKeys, may help mitigate embedded phishing attacks, these approaches do have limitations [21] and are orthogonal to the goals of this paper.¹⁾)

More importantly—and the subject of this paper—even legitimate embedded interfaces may be subject to a wide range of attacks, or may present a threat to the application or page that embeds them. In particular, drawing in part on [18], embedded interfaces or their parents may be subject to:

Display forgery attacks, in which the parent application modifies the child element (e.g., to display a false payment value), or vice versa.

Size manipulation attacks, in which the parent application violates the child element’s size requirements or expectations (e.g., to secretly take photos by hiding the camera preview [26]), or the child element sets its own size inappropriately (e.g., to display a full-screen ad).

Input forgery attacks, in which the parent application delivers forged user input to a child element (e.g., to programmatically click on an advertisement to increase ad revenue), or vice versa.

Clickjacking attacks, in which the parent application forces or tricks the user into clicking on an embedded element [11] using visual tricks (partially obscuring the child element or making it transparent) or via timing-based attacks (popping up the child element just as the user is about to click in a predictable place).

Focus stealing attacks, in which the parent application steals the input focus from an embedded element, capturing input intended for it, or vice versa.

¹⁾We also note that phished or spoofed interfaces are little threat if they do not accept private user input—for example, clicking on a fake ACG will not grant any permissions to the embedding application.

Ancestor redirection attacks, in which a child element redirects an ancestor (e.g., the top-level) application or page to a target of its choice, without user consent.

Denial-of-service attacks, in which the parent application prevents user input from reaching a child element (e.g., to prevent a user from clicking “Cancel” on an authorization dialog), or vice versa.

Data privacy attacks, in which the parent or child extract content displayed in the other.

Eavesdropping attacks, in which the parent application eavesdrops on user input intended for a child element (e.g., a sensitive search query), or vice versa.

2.3 Security Goals

Motivated by the above challenges and building on recent work [18], we now describe the security goals that we apply in our analysis and implementation. Where noted, we describe additional goals not discussed by prior work.

1. *Display Integrity*. One principal cannot alter the content or appearance of another’s interface element, either by direct pixel manipulation or by element size manipulation. This property prevents display forgery and size manipulation attacks.
2. *Input Integrity*. One principal cannot programmatically interact with another’s interface element. This property prevents input forgery attacks.
3. *Intent Integrity*. First, an interface element can implement (or request that the system enforce) protection against clickjacking attacks. Second, one principal cannot prevent intended user interactions with another’s interface element (denial-of-service). Finally, based on our implementation experience (Section 5), we add two additional requirements not discussed in previous work: an embedded interface element cannot redirect an ancestor’s view without user consent, and no interface element can steal focus from another interface element belonging to a different principal.
4. *Data Isolation*. One principal cannot extract content displayed in, nor eavesdrop on user input intended for, another’s interface element. This property prevents data privacy and eavesdropping attacks.
5. *UI-to-API Links*. APIs can verify that they were called by a particular principal or interface element.

These properties assume that principals can be reliably distinguished and isolated, either by process separation, run-time validation (e.g., of the same-origin policy), or compile-time validation (e.g., using static analysis).

3 The Case for Secure UIs in Android

While Section 2 considered UI embedding in general, we now specifically make the case for secure embedded UIs in Android. The fact that an Android application cannot embed another application’s interface results in a

fundamental trust assumption built into the Android UI toolkit. In particular, every UI element trusts its parent and its children, who each have unrestricted access to the element’s APIs. Vulnerabilities arise when this trust assumption is violated, e.g., because an embedded element is provided by a third-party library.

We now introduce several case studies illustrating that embedded user interface scenarios in stock Android are often either insecure or impossible. We will return to these case studies in Section 6 and reevaluate them in the context of our implementation.

Advertising. In stock Android, applications wishing to embed third-party advertisements must include an ad library, such as AdMob or Mobclix, which runs in the embedding application’s process. These libraries provide a custom UI element (an AdView) that the embedding application instantiates and embeds. As has been discussed extensively in prior work [17, 23], the library model for third-party advertisements comes with a number of security and privacy concerns. For example, the host application must trust the advertising library not to abuse the host’s permissions or otherwise exploit a buggy host application. Additionally, ad libraries ask their host applications to request permissions (such as location and Internet access) on their behalf; applications that request permissions not clearly relevant to their stated purpose can desensitize users to permission warnings [8].

Prior work [18] has also identified and experimentally demonstrated threats to the AdView. Parent applications can mount a programmatic clickfraud attack in which they programmatically click on embedded ads to increase their advertising revenue. Similarly, parent applications can mount clickjacking attacks by, for example, covering the AdView with another UI element that does not accept (and thus lets pass through) input events.

WebViews. One of the built-in UI elements provided by Android is the WebView, which can load local HTML content or an arbitrary URL from the Internet. Though WebViews appear conceptually similar to iframes, they do not provide many of the same security properties. In particular, WebViews—and more importantly, the contained webpage—can be completely manipulated by the containing application, which can mount attacks including programmatic clicking, clickjacking, and input eavesdropping [13]. Thus, for example, if an Android application embeds a WebView that loads a login page, that application can eavesdrop on the user’s password as he or she enters it into the WebView.

Facebook Social Plugins. On the Web, Facebook provides a set of social plugins [6] to third-party web developers. These plugins include the “Like” button, a comments widget, and a feed of friends’ activities on the embedding page (e.g., which articles they liked or

shared). These social plugins are generally implemented as iframes and thus isolated from the embedding page.

While Facebook also supplies an SDK for smart-phones (iOS and Android), this library—like all libraries, it runs in the host application’s process—does not provide embeddable plugins like those found on the Web. A possible reason for this omission is that Facebook’s SDK for Android cannot prevent, for example, applications from programmatically clicking on an embedded “Like” button or extracting private information from a recommendations plugin. Although developers can manually implement a social plugin using a Web-View, this implementation suffers from the security concerns described above. Thus, though embeddable social plugins on mobile may be desirable to Facebook, they cannot be achieved securely on stock Android.

Access Control Gadgets. Finally, recent work [19] has proposed access control gadgets (ACGs), secure embedded UI elements that are used to capture a user’s permission-granting intent (e.g., to grant an application access to the user’s current location). Authentically capturing a user’s intent relies on a set of UI-level security properties including clickjacking protection, display isolation, and user intent protection. As we describe in this paper, fundamental modifications to Android are required to enable secure embedded elements like ACGs.

4 Analysis of UI Embedding Systems

To assess the spectrum of solutions and to inform our implementation choices, we now step back and analyze prior Web and Android based solutions for cross-application embedded interfaces with respect to the set of security properties described in Section 2.3. This analysis is summarized in Figure 1.

4.1 Browsers

Browsers support third-party embedding by allowing web pages to include iframes from different domains. Like all pages, iframes are isolated from their parent pages based on the same-origin policy [27], and browsers do not allow pages from different origins to capture or generate input for each other.

However, an iframe’s parent has full control of its size, layout, and style, including the ability to make it transparent or overlay it with other content. These capabilities enable clickjacking attacks. While there are various “framebusting” techniques that allow a sensitive page to prevent itself from being framed in an attempt to prevent such attacks, these techniques are not foolproof [20]. More importantly, framebusting is a technique to prevent embedding, not one that supports secure embedding.

Additionally, while an iframe cannot read the URL(s) of its ancestor(s), it can change the top-level URL, redirecting the page without user consent. Newer version of

Category	Security Requirement	Browsers	Android	AdDroid [17]	AdSplit [23]	RFK [18]
Display Integrity	Prevents direct modification	✓	✗	✗	✓	✓*
	Prevents size manipulation	✗	✗	✗	✗	✓*
Input Integrity	Prevents programmatic input	✓	✗	✗	✓	✓*
Intent Integrity	Clickjacking protection	✗	✓	✓	✓	✓
	Prevents input denial-of-service	✓	✗	✗	✗	✓*
	Prevents focus stealing	✓	✗	✗	✓	✗
	Prevents ancestor redirection	✓	✗	✗	✗	✗
Data Isolation	Prevents access to display	✓	✗	✗	✓	✓*
	Prevents input eavesdropping	✓	✗	✗	✗	✓*
UI-to-API Links	APIs can verify caller	✓	✗	✗	✓	✓*

Figure 1: **Analysis of Existing Systems.** This table summarizes, to the best of our knowledge, the UI-level security properties (first defined in prior work [18] and expanded here) achieved by existing systems. Figure 2 similarly analyzes our implementation. * Checkmarks annotated with an asterisk require static analysis or hypothetical (not prototyped) changes to the Android framework.

some browsers allow parent pages to protect themselves by using the `sandbox` attribute for iframes; thus, we’ve indicated that the Web prevents such attacks in Figure 1. However, we observe that it may be desirable to allow user actions to override such a restriction, and we describe how to achieve such a policy in later sections.

Research browsers and browser operating systems (e.g., Gazelle [29] and IBOS [25]) provide similar embedded UI security properties as traditional browsers, and thus we omit them from Figure 1. Gazelle partially addresses clickjacking by allowing only opaque cross-origin overlays, but this policy is not backwards compatible. Furthermore, malicious parent pages can still obscure embedded content by partially overlaying additional content on top of sensitive iframes. We discuss additional work considering clickjacking in Section 9.

4.2 Android

Two recent research efforts [17, 23] propose privilege separation to address security concerns with Android’s advertising model (under which third-party ad libraries run in the context of the host application). AdDroid’s approach [17] introduces a system advertising service that returns advertisements to the AdDroid userspace library, which displays them in a new user interface element called an AdView. While this approach successfully removes untrusted ad libraries from applications, it does not provide any additional UI-level security properties for the embedded AdView beyond what is provided by stock Android (see Figure 1). For example, it does not prevent the host application from engaging in clickfraud by programmatically clicking on ads.

AdSplit [23], on the other hand, fully separates advertisements into distinct Android applications (one for each host application). AdSplit achieves the visual embedding of the ad’s UI into the application’s UI by overlaying the host application, with a transparent region for

the ad, on top of the ad application. It prevents programmatic clickfraud attacks by authenticating user input using Quire [3]. As summarized in Figure 1, AdSplit meets the majority of security requirements for embedded UIs. Indeed, the requirements it meets are sufficient for embedded advertisements. Because it does not meet all of the requirements, however—most importantly, it does not prevent input eavesdropping—AdSplit would not be well-suited as a generalized solution for embedded UIs.

Finally, the prototype implementation described in [18] to meet that work’s goals (upon which we build) also contains weaknesses. In particular, the isolation and identification of different principals (“trust groups” in the terminology of that paper) is insecure, undermining all of the security properties. Rather than truly supporting one Android application embedding UI from another application, it merely separates interfaces defined in the main application from those defined in included libraries. This separation relies on Java package names, static code analysis, and hypothetical changes to the Android framework (e.g., changing Android’s Java classloader to enable package sealing) that have not been implemented or verified in practice.

5 Implementation Experience: LayerCake

We now explore what it takes to support secure embedded UIs, under the definitions from Section 2.3, in the Android framework. As no existing Android-based solutions meet these goals, we view this implementation as an opportunity to consider secure embedding from scratch. While we adapt techniques from prior work, we find that previously published guidelines are not always directly applicable. For example, we found that we could simplify a prior approach [18] when overlaying cross-application content, but that we faced additional practical challenges, such as the need to propagate layout changes

Category	Security Requirement	LayerCake	(Section Number) Approach
Display Integrity	Prevents direct modification Prevents size manipulation	✓ ✓	(5.3) Embedded elements in isolated, overlaid windows. (5.6) User notifications on size conflicts.
Input Integrity	Prevents programmatic input	✓	(5.3) Embedded elements in isolated, overlaid windows.
Intent Integrity	Clickjacking protection Prevents input denial-of-service	✓ ✓	(5.7) No input delivered if view/window not fully visible. (5.3) Embedded windows attached to system root.
Data Isolation	Prevents access to display Prevents input eavesdropping Prevents focus stealing Prevents ancestor redirection	✓ ✓ ✓ ✓	(5.3) Embedded elements in isolated, overlaid windows. (5.3) Embedded windows attached to system root. (5.4) Focus changes only in response to real user clicks. (5.8) Prompts and (6.2) redirection ACG.
UI-to-API Links	APIs can verify caller	✓	(5.2) Elements from different principals run in separate calling processes (identifiable by package name).

Figure 2: **Techniques for Secure Embedded UI.** This table summarizes how LayerCake (our modified version of Android 4.2) achieves each of the desired security properties for embedded user interface elements.

and handle multiple levels of nesting. We further discuss these and other challenges and lessons in Section 8.

We thus created *LayerCake*, a modified version of the Android framework that supports cross-application embedding via changes to the ActivityManager, the WindowManager, and input dispatching. We added or modified 2400 lines of source code across 50 files in Android 4.2 (Jelly Bean). Figure 2 summarizes the implementation choices that achieve our desired security properties.

5.1 Android Background

Android user interfaces are focused around Activities, which present the user with a particular view (or screen) of an application. An application generally consists of multiple Activities (e.g., settings, comments, and news-feed Activities), each of which defines an interface consisting of built-in or custom UI elements (called Views).

Android’s ActivityManager keeps only one Activity in the foreground at a time. An application cannot embed an Activity from another application, and two applications cannot run side-by-side. While Android does provide support for ActivityGroups (deprecated in favor of Fragments) to improve UI code reuse within an application, these mechanisms do not provide true Activity embedding and are not applicable across application and process boundaries. The goal of our exploration is to allow one application to embed an Activity from another application (running in that other application’s process).

Each running Android application is associated with one or more windows, each of which serves as the root of an interface layout tree consisting of application-specified Views. Android’s WindowManager isolates these windows from each other—e.g., an application cannot access the status bar’s window (shown at the top of the screen)—and appropriately dispatches user input. Our implementation relies on these isolation properties.

While only one Activity can be in the foreground, multiple applications/processes may have visible windows.

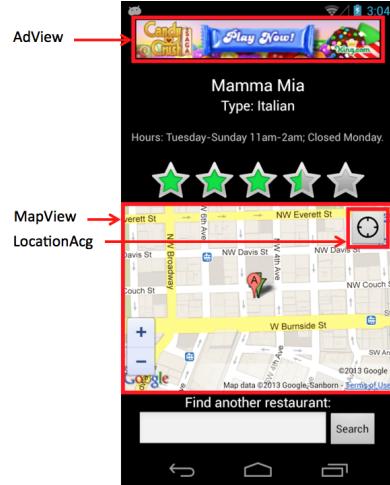


Figure 3: **Sample Application.** This restaurant review application embeds two third-party Activities, an advertisement and a map. The map Activity further embeds an access control gadget (ACG) for location access.

For example, the status bar runs in the system process, and the window of one application may be visible below the (partially) transparent window of another. As an example of the latter, AdSplit [23] achieves visual embedding by taking advantage of an application’s ability to make portions of its UI transparent. However, recall from Section 4 and Figure 1 that this approach is insufficient for generalized embedded UI security.

5.2 Supporting Embedded Activities

LayerCake introduces a new View into Android’s user interface toolkit (Java package `android.view`) called `EmbeddedActivityView`. It allows an application developer to embed another application’s Activity within her application’s interface by specifying in the parameters of the `EmbeddedActivityView` the package and class

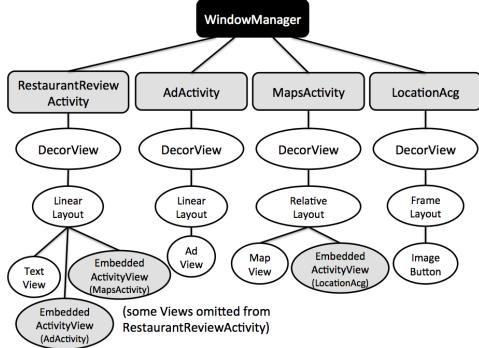


Figure 4: Window Management. This figure shows the Window/View tree for the Activities in Figure 3. Embedded Activities are not embedded in the View tree (circles) of their parent, but rather within a separate window. The WindowManager keeps track of a window (grey squares) for each Activity and visually overlays an embedded window on top of the corresponding EmbeddedActivityView in the parent Activity.

names of the desired embedded Activity. Figure 3 shows a sample application that embeds several Activities.

We extended Android’s ActivityManager (Java) to support embedded Activities, which are launched when an EmbeddedActivityView is created and displayed. Unlike ordinary Activities, embedded Activities are not part of the ActivityManager’s task stack or history list, but rather share the fate of their parent Activity. Crucially, this means that an embedded Activity’s lifecycle is linked to that of its parent: when the parent is paused, resumed, or destroyed, so are all of its embedded children.

An Activity may embed multiple other Activities, which themselves may embed one or more Activities (multiple nesting). Each embedded Activity is started as a new instance, so multiple copies of the same Activity are independent (although they run in the same application, allowing changes to the application’s global state to persist across different Activity instances).

5.3 Managing Windows

Properly displaying embedded Activities required modifications to the Android WindowManager (Java). One option for achieving embedded UI layouts is to literally nest them—that is, to add the embedded Activity’s Views (UI elements) as children in the parent Activity’s UI tree. However, this design would allow the parent Activity to mount input eavesdropping and denial-of-service attacks on the child Activity. Thus, following the interface layout tree invariants described in prior work [18], we do not literally nest the interface elements of embedded Activities inside the parent Activity. Instead, an embedded Activity is displayed in a new window, overlaid on top of the window to which it is attached (i.e., the window of the parent Activity). This overlay

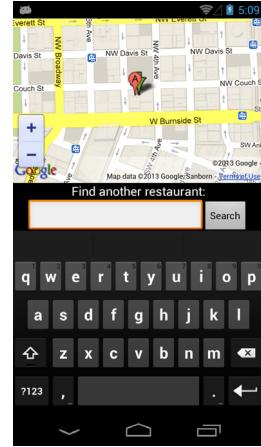


Figure 5: Panning for Software Keyboard. The restaurant review application (from Figure 3), including its overlaid embedded windows, must be panned upward to make room for the software keyboard underneath the in-focus text box.

achieves the same visual effect as literal embedding but prevents input manipulation attacks. Figure 4 shows an example of the interface layout trees associated with the Activities in the sample application in Figure 3. We note that we were able to simplify the proposed approach [18], which we found to be overly general (see Section 8).

By placing embedded Activities into their own windows instead of into the parent’s window, we also inherit the security properties provided by the isolation already enforced by the WindowManager. In particular, this isolation prevents a parent Activity from modifying or accessing the display of its child Activity (or vice versa).

The relative position and size of an overlaid window are specified by the embedding application in the layout parameters of the EmbeddedActivityView and are honored by the WindowManager. (Note that the specified size may violate size bounds requested by the embedded Activity, as we discuss in Section 5.6.)

The layout parameters of an embedded Activity’s window must remain consistent with those of the associated EmbeddedActivityView, a practical challenge not described in prior work. For example, when the user re-orientates the phone into landscape mode, the parent Activity will adjust its UI. Similarly, when the soft keyboard is shown, Android may pan the Activity’s UI upwards in order to avoid covering the in-focus text box with the keyboard (Figure 5). In both cases, the embedded Activity’s windows must be relocated appropriately. To support these dynamic layout changes, the EmbeddedActivityView reports its layout changes to the WindowManager, which applies them to the associated window.

Finally, since LayerCake supports multiple levels of embedding, it must appropriately display windows multiple levels down (e.g., grandchildren of the top-level Ac-

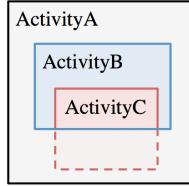


Figure 6: Cropping Further Nested Activities. If a grandchild (ActivityC) of the top-level Activity (ActivityA) is placed or scrolled partly out of the visible area of its immediate parent (ActivityB), it must be cropped accordingly.

tivity). For example, suppose ActivityA embeds ActivityB which embeds ActivityC. If the EmbeddedActivityView (inside ActivityB) that corresponds to ActivityC is not fully visible—e.g., because it is scrolled halfway out of ActivityB’s visible area—then the window corresponding to ActivityC must be cropped accordingly (Figure 6). This cropping is necessary because ActivityC is not literally nested within ActivityB, but rather overlaid on top of it, as discussed above.

5.4 Handling Focus

Both the parent and any embedded Activities must properly receive user input. While touch events are dispatched correctly even in the presence of visually overlapping windows, stock Android grants focus for key events only to the top-level window. As a result, only the window with the highest Z-order in an application with embedded Activities will ever receive key events. We thus modified Android to switch focus between windows belonging to the parent or any embedded Activities within an application, regardless of Z-order.

In particular, we changed the input dispatcher (C++) to deliver touch events to the WindowManager in advance of delivering them to the resolved target. When the user touches an unfocused window belonging to or embedded by the active application, the WindowManager redirects focus. Windows that might receive the redirected focus include that of the parent Activity, the window of any embedded Activity, or an attached window from the same process (e.g., the settings context menu, which Android displays in a new window). Switching focus only in response to user input (rather than an application’s request) prevents a parent or child window from stealing focus to eavesdrop on input intended for another principal.

5.5 Supporting Cross-Principal APIs

To support desired functionality, embedded UI elements and their parents must communicate. For example, an application embedding an ad may wish to communicate keywords to the ad provider, or a system-defined location button (ACG) may wish to pass the current location to the parent application in response to a user click. To en-

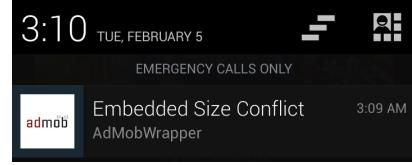


Figure 7: Size Conflict Notification. If the AdMobWrapper application specifies a minimum size that the RestaurantReviewActivity does not honor when it embeds the advertisement, a system notification is displayed to the user. Clicking on the notification displays a full-screen advertisement Activity.

able flexible communication between embedded Activities and their parents, we leverage the Android Interface Definition Language (AIDL), which lets Android applications define interfaces for interprocess communication. We thus define the following programming model.

Each embeddable Activity defines two AIDL interfaces, one that it (the child) will implement, and one that the parent application must implement. For example, the advertisement (child) may implement a `setKeywords()` method, and the ad’s parent application may be asked to implement an `onAdLoaded()` method to be notified that an ad has been successfully loaded. When an application wishes to embed a third-party Activity, it must keep copies of the relevant interface files in its own source files (as is standard with AIDL), and it must implement `registerChildBinder()`. This function allows the child Activity, once started, to make a cross-process call registering itself with the parent.

We note that this connection is set up automatically only between parents and immediate children, as doing so for siblings or farther removed ancestors may leak information about the UIs embedded by another principal.

5.6 Handling Size Conflicts

Recall from Section 5.3 that the WindowManager honors the parent application’s size specification for an `EmbeddedActivityView`. This policy prevents a child element from taking over the display (a threat discussed further in the context of ancestor redirection below). However, we also wish to prevent size manipulation by the parent.

We observe that it is only of concern if an embedded Activity is given a smaller size than requested, since it need not scale its contents to fill its (possibly too large) containing window. Thus, we modified the Activity descriptors to include only an optional minimum height and width (specified in density-independent pixels).

Prior work [18] describes different size conflict policies based on whether the embedded element is trusted or untrusted by the system. If it is trusted (e.g., a system-defined ACG), its own size request should be honored; if it is untrusted (e.g., an ad that requests a size filling

the entire screen), the parent’s size specification is honored. However, we observe that a malicious parent can mimic the effect of making a child element too small using other techniques, such as scrolling it almost entirely off-screen—and that doing so maliciously is indistinguishable from legitimate possible scroll placements. We thus further consider the failure to meet minimum size requirements in the context of clickjacking (Section 5.7).

Thus, since enforcing a minimum size for trusted embedded elements does provide additional security properties in practice, we use the same policy no matter whether mis-sized elements are trusted or untrusted by the system. That is, the WindowManager honors the size specifications of the parent Activity. If these values are smaller than the embedded Activity’s request, a status bar notification is shown to the user (Figure 7). Similar to a browser’s popup blocker, the user can click this notification to open a full-screen (non-embedded) version of the Activity whose minimum size was not met.

5.7 Support for Clickjacking Prevention

In a clickjacking attack [11], a malicious application forces or tricks a user into interacting with an interface, generally by hiding important contextual information from the user. For instance, a malicious application might make a sensitive UI element transparent or very small, obscure it with another element that allows input to pass through it, or scroll important context off-screen (e.g., the preview associated with a camera button).

To prevent such attacks, an interface may wish to discard user input if the target is not fully visible. Since it may leak information about the embedding application to let an element query its own visibility, LayerCake allows embedded Activities to request that the Android framework simply not deliver user input events if the Activity is:

1. *Covered (fully or partly) by another window.* This request is already supported by stock Android via `setFilterTouchesWhenObscured()`.
2. *Not the minimum requested size.* A parent application may not honor a child’s size request (see Section 5.6).
3. *Not fully visible due to window placement.* An embedded Activity’s current effective window may be cropped due to scrolling.

Note that an embedded Activity need not be concerned about a malicious parent making it transparent, because stock Android already does not deliver input to invisible windows. Similarly, an Activity need not be concerned about malicious visibility changes to UI elements within its own window, since process separation ensures that the parent cannot manipulate these elements. To prevent timing-based attacks, these criteria should be met for some minimum duration [11] before input is delivered, a check that we leave to future work.

We emphasize that embedded iframes on the Web today can neither discover if all of these criteria are met—due to the same-origin policy, they cannot know if the parent page has styled them to be invisible or covered them with other content—nor request that the browser discard input under these conditions.

5.8 Preventing Ancestor Redirection

Android applications use Intents to launch Activities either in their own execution context (e.g., to switch to a Settings Activity) or in another application (e.g., to launch a browser pointed at a specified URL). In response to a `startActivity(intent)` system call, Android launches a new top-level full-screen Activity. Recall that allowing an embedded element to redirect the ancestor UI without user consent is a security concern.

We thus make two changes to the Android framework. First, we introduce an additional flag for Intents that starts the resulting Activity inside the window of the embedded Activity that started it. Thus, for example, if an embedded music player wishes to switch from its `MusicSelection` Activity to its `NowPlaying` Activity without breaking out of its embedded window, it can do so by specifying `Intent.FLAG_ACTIVITY_EMBEDDED`. (If the music player is not embedded, this flag is simply ignored.)

Second, we introduce a prompt shown to users when an embedded Activity attempts to launch another Activity full-screen (i.e., not using the flag described above). This may happen either because it is a legacy application unaware of the flag, or for legitimate reasons (e.g., a user’s click on an embedded advertisement opens a new browser window). However, studies have shown that prompting users is disruptive and ineffective [16]; in Section 6.2 we discuss an access control gadget (ACG) that allows embedded applications to launch full-screen Intents in response to user clicks without requiring that the system prompt the user.

6 Case Studies

We now return to the case studies introduced in Section 3 and describe how LayerCake supports these and other scenarios. Figure 8 shows that implementation complexity is low, especially for parent applications.

6.1 Geolocation ACG

To support user-driven access for geolocation, we implemented a geolocation access control gadget (ACG) in the spirit of prior work [19]. We added a `LocationAcg` Activity to Android’s `SystemUI` (which runs in the system process and provides the status bar, the recent applications list, and more). This Activity, which other applications can embed, simply displays a location button (see Figure 3).

	Lines of Java	Parent Lines of Java
Geolocation ACG	111	14
Redirection Intent ACG	75	23
Secure WebView	133	13
Advertisement	562	37
FacebookWrapper	576	30

Figure 8: Implementation Complexity. *Lines of code for (1) the embedded Activity and (2) the parent’s implementation of the AIDL interface. We omit legacy applications because they required no modifications and expose no parent interfaces. Implementation complexity is low, especially for embedders.*

Following a user click, the SystemUI application, not the parent application, accesses Android’s location APIs. To then receive the current location, the parent application must implement the `locationAvailable()` method defined in the parent AIDL interface provided by the LocationAcg’s developers (us).

Security Discussion. LayerCake provides the security properties required to enable ACGs. In particular, the parent application of a LocationAcg cannot trick the user into clicking on the gadget, manipulate the gadget’s look, or programmatically click on it.

We emphasize again that this ACG provides location information to the parent application only when the user wishes to share that information; a well-behaving parent application will not need location permissions. In a system like Android, where applications can request location permissions in their manifest, it is an open question how to incentivize developers to use the corresponding ACG instead of requesting that permission. Prior work [19] has suggested incentives including increased scrutiny at app store review time of applications requesting sensitive permissions.

6.2 Redirection Intent ACG

In Section 5.8, we introduced a system prompt when an embedded Activity attempts to start a full-screen Activity. However, prompts are known to be disruptive and often ignored, especially following a user action intended to cause the effect about which the prompt warns [31]. For example, a user who clicks on an embedded ad in stock Android today expects it to open the ad’s target in a new (non-embedded) browser window. Following the philosophy of user-driven access control [19], we thus allow embedded Activities to start top-level Activities without a prompt if `startActivity()` is called in response to a user’s click.

To verify that the user has actually issued the click, we take advantage of our system’s support for ACGs and implement an ACG for top-level redirection. This `RedirectAcg` Activity again belongs to Android’s SystemUI application. It consists primarily of an `ImageView` that may be filled with an arbitrary Bitmap, al-

lowing the embedder to completely specify its look. An embedded Activity that embeds such an ACG (two levels of embedding) thus uses the cross-process API provided by the `RedirectAcg` to (1) provide a Bitmap specifying the look, and (2) specify an Intent to be supplied to the `startActivity()` system call when the user clicks on the `RedirectAcg` (i.e., the `ImageView`’s `onClick()` method is fired).

Security Discussion. The UI-level security properties provided by LayerCake ensure that the `RedirectAcg`’s `onClick()` method is fired only in response to real user clicks. In other words, the embedding application cannot circumvent the user intent requirement for launching a top-level Activity by programmatically clicking on the `RedirectAcg` or by tricking the user into clicking on it.

Unlike the `LocationAcg`, however, the embedding application is permitted to fully control the look of the `RedirectAcg`. This design retains backwards compatibility with the stock Android experience and relies on the assumption that a user’s click on anything within an embedded Activity indicates the user’s intent to interact with that application. However, alternate designs might choose to restrict the degree to which the redirecting application can customize the `RedirectAcg`’s interface. For example, the system could place a visual “full-screen” or “redirect” indicator on top of the application-provided Bitmap, or it could simply support a stand-alone “full-screen” ACG that applications wishing to open a new top-level view must display without customization.

Note that developers are incentivized to use the `RedirectAcg` because otherwise attempts to launch top-level Activities will result in a disruptive prompt (Section 5.8).

6.3 Secure WebView

We implemented a `SecureWebView` that addresses security concerns surrounding Android WebViews [12, 13]. The `SecureWebView` is an Activity in a new built-in application (`WebViewApp`) that consists solely of an ordinary `WebView` (inside a `FrameLayout`) that fills the Activity’s whole UI. Thus, when another Activity embeds a `SecureWebView`, the internal `WebView` takes on the dimensions of the associated `EmbeddedActivityView`.

The `SecureWebView` Activity exposes a safe subset (see below) of the underlying `WebView`’s APIs to its embedding process. The current version of LayerCake exposes only a subset of these APIs for demonstration purposes. A complete implementation will need to properly (de)serialize all complex data structures (e.g., `SslCertificate`) across process boundaries.

Security Discussion. Separating out the Android `WebView` into another process—that of the `WebViewApp`—provides important missing security properties. It is no longer possible to eavesdrop on input to the embedded

webpage, to extract content or programmatically issue input, or to manipulate the size, location, or transparency of the WebView to mount clickjacking attacks.

While the SecureWebView wraps the existing WebView APIs, it should avoid exposing certain sensitive APIs, such as those that mimic user input (e.g., scrolling via `pageUp()`) or that directly extract content from the WebView (e.g., screenshot via `capturePicture()`). Note, however, that APIs which redirect the SecureWebView to another URL are permitted, as the parent application could simply open a new SecureWebView instead.

Ideally, Android would replace the WebView with the SecureWebView, but this change would not be backwards compatible and may conflict with the goals of some developers in using WebViews. Thus, we observe that using a SecureWebView also benefits the embedding application: if it exposes an API to the webpage via an ordinary WebView (using `addJavascriptInterface()`), a malicious page could use this to manipulate the host application. Process separation protects the host application from such an attack, and since the WebViewApp has only the `INTERNET` permission, the attack’s effect is limited. Additionally, WebView cookies are not shared across processes; the SecureWebView allows applications to reuse (but not access) existing cookies, possibly providing a smoother user experience.

6.4 Advertisements

Recall that stock Android applications embedding third-party advertisements include an ad library that runs in the host application’s process and provides an AdView element. Our modifications separate the AdView out into its own process (see the advertisement in Figure 3). To do this, we create a wrapper application for the AdMob advertising library [10]. The wrapper application exposes an embeddable Activity (called `EmbeddedAd`) that instantiates an AdMob AdView with the specified parameters. This Activity exposes all of AdMob’s own APIs across the process boundary, allowing the embedding application to specify parameters for the ad.

Security Discussion. Moving ads into their own process (one process per ad library) addresses a number of the concerns raised in Section 3. In particular, an ad library can no longer abuse a parent application’s permissions or exploit a buggy parent application. Furthermore, the permissions needed by an ad library, such as Internet and location permissions, must no longer be requested by the parent application (unless it needs these permissions for other purposes).

Note that all ads from a given ad library—even if embedded by different applications—run in the same process, allowing that ad application to leverage input from different embedders. For example, if one appli-

cation provides the user’s age and another provides the user’s gender, the ad application can better target ads in *all* parent applications, without revealing additional information to applications that did not already have it. (However, we note that some users may prefer that ad applications not aggregate this information.)

LayerCake goes beyond process separation, providing UI-level security absent in most prior systems (except AdSplit [23]). Most importantly, the parent can no longer mount programmatic click fraud attacks.

6.5 Facebook Social Plugins

We can now support embedded Facebook social widgets in a secure manner. We achieve this by creating a Facebook wrapper application that exposes Activities for various Facebook social widgets (e.g., a Comments Activity and a Like Activity—see Figure 9). Each Activity displays a WebView populated with locally-generated HTML that references the Facebook JavaScript SDK to generate the appropriate plugin (as done ordinarily by web pages and as specified by Facebook [6]).

Security Discussion. LayerCake supports functionality that is impossible to achieve securely in stock Android and may be desirable to Facebook. This functionality was previously available only on the Web, due to the relative security of embedded iframes (though clickjacking, or “likejacking”, remains a problem on the Web). Our implementation protects the social widgets both by separating them into a different process (preventing data extraction, among others), and by enforcing other UI-level security properties (preventing clickjacking and programmatic clicking).

We observe that a malicious application might attempt to mimic the FacebookWrapper application by populating a local WebView with the HTML for a social plugin. To prevent this attack, we recommend that the FacebookWrapper application include a secret token in the HTML it generates (and that Facebook’s backend verify it), similar in approach to CSRF protections on the Web.

6.6 Legacy Applications

The applications discussed so far needed wrapper applications because the wrapped functionality was not previously available in a stand-alone fashion. However, this need is not fundamental—any legacy Android application (i.e., one that targets older versions of the Android SDK) can be embedded using the same techniques.

To demonstrate this, we created an application that embeds both the existing Pandora application and the existing Amazon application. To do so, we needed to discover the names of the corresponding Activities in the existing applications. This information is easy to discover from Android’s standard log, which prints information about Intent targets when they are launched. Figure 10



Figure 9: Facebook Social Plugins. This example blog application embeds both a Facebook “Like” button and a comments feed, both running in our FacebookWrapper application.

shows a screenshot of the resulting application.

Security Discussion. As in previous case studies, the embedded Activities are isolated from the parent. Thus, they cannot access sensitive information in or manipulate the UI or APIs in the parent application, or vice versa.

Legacy applications naturally do not use the new `FLAG_ACTIVITY_EMBEDDED` flag when launching internal Activities. While updated versions of Pandora and Amazon could use this flag to redirect within an embedded window, the experience with unmodified legacy applications is likely to be disruptive. Thus, a possible policy (perhaps subject to a user preference setting) for such applications is to internally modify all Activity launches to use the new flag, never allowing these applications to break out of their embedded windows.

Embedding arbitrary applications that were not intended by their developers to be embedded also raises the question of embedding permissions. Some Activities may wish never to be embedded, or to be embedded only by authorized parents. Future modifications to LayerCake should support such permissions.

7 Performance Evaluation

We evaluate the performance impact of our changes to Android by measuring the time it takes to start an application, i.e., the delay between a `startActivity()` system call and the `onCreate()` call for the last embedded Activity (or the parent Activity, if none are embedded). As shown in the top of Figure 11, applications with embedded Activities take longer to fully start. The reason for this is that the parent Activity’s layout must be created (in its `onCreate()`) before child Activities can be identified. Thus, an application with multiple nested Activities (e.g., `RestaurantReviewer`) requires linearly more time than an application with only one level

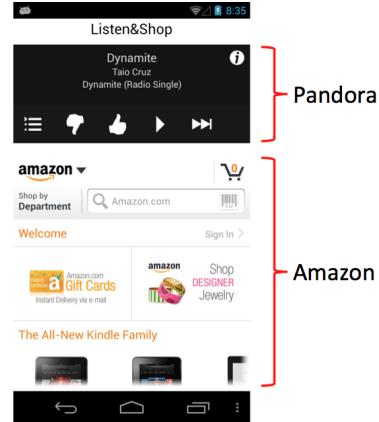


Figure 10: Embedded Pandora and Amazon Apps. Legacy applications can also be embedded, raising policy questions regarding top-level intents and embedding permissions.

of nesting (e.g., `FacebookDemo` or `Listen&Shop`). We note that the parent Activity’s own load time is unaffected by the presence of embedded content (e.g., the `FacebookDemo` Activity starts in 160 ms, even though the embedded Facebook components require 300 ms). Prior work [15] has argued that the time to display first content is more important than full load time.

We also measure input event dispatch time (e.g., the time it takes for Android to deliver a touch event to an application). Specifically, we evaluate the impact of dispatching input events first to the `WindowManager`, allowing it to redirect focus if appropriate (Section 5.4). The bottom of Figure 11 shows that involving the `WindowManager` in dispatch has a negligible performance impact over stock Android; changing focus has a greater impact, but it is not noticeable by the user, and focus change events are likely rare.

We can also report anecdotally that the effect of embedding on the performance of our case study applications was unnoticeable, except that the panning of embedded windows (for the software keyboard) appears to lag slightly. This case could likely be optimized by batching cross-process relay layout messages.

Finally, supporting embedded Activities may result in more applications running on a device at once, potentially impacting memory usage and battery life. The practical impact of this issue depends on the embedding behavior of real applications—for example, perhaps most applications will include ads from a small set of ad libraries, limiting the number of applications run in practice.

8 Discussion

Whereas existing systems—particularly browsers—have evolved security measures for embedded user interfaces over time, this paper has taken a principled ap-

Application	Load time (10 trial average)	
	No Embedding	With Embedding
RestaurantReviewer	163.1 ms	532.6 ms
FacebookDemo	157.5 ms	304.9 ms
Listen&Shop	159.6 ms	303.3 ms

Scenario	Event Dispatch Time (10 trial average)
Stock Android	1.9 ms
No focus change	2.1 ms
Focus change	3.6 ms

Figure 11: **Performance.** The top table shows the time it takes for the `onCreate()` method of all included Activities to be called. We note that the time to load the parent Activity remains the same whether or not it uses embedding, so the time for the parent to begin displaying native content is unaffected. The bottom table shows that the effect of intercepting input events in the WindowManager for possible focus changes is minor.

proach to defining a set of necessary security properties and building a system with full-fledged support for embedding interfaces based on these properties.

8.1 Lessons for Embedded Interfaces

From this process, we provide a set of techniques for systems that wish to support secure cross-application UI embedding. Figure 2 outlines the security properties provided by LayerCake and summarizes the implementation techniques used to achieve each property. While prior works [18, 19] have stated the need for many (though not all) of these properties, they have not provided detailed guidelines for implementation. We hope this work, in which we bring techniques from prior work together into a practical implementation, will serve that purpose.

Our implementation experience challenges several previous assumptions or choices. These lessons include:

User-driven ancestor redirection. Embedded applications should not be able to redirect an ancestor application/page without user consent. We argue that a reasonable tradeoff between security and usability is to prompt users only if the redirection attempt does not follow a user click (indicating the user’s intent to interact with the embedded content). While newer browsers prevent embedded iframes from redirecting the top-level page programmatically, they do not allow user actions (e.g., clicking on a link with target `_top`) or other mechanisms to override this restriction. In our case studies, we saw that this type of click-enabled redirection can be useful and expected (e.g., when a user clicks on an embedded ad, he or she likely expects to see full-screen content about the advertised product or service). In our system, we were able to apply ACGs in a novel way to capture a user’s redirection intent (Section 6.2).

Size manipulation as a subset of clickjacking. We ini-

tially considered size manipulation (by the parent of an embedded interface element) to be a stand-alone threat. A solution that we considered is to treat elements that are trusted or untrusted by the system differently (e.g., an access control gadget is trusted while an advertisement is not), letting the system enforce the minimum requested size for trusted elements. However, this solution provides no additional security, since a malicious parent can use other techniques to obscure the sensitive element (e.g., partially covering it or scrolling it partly off-screen). Thus, we consider size manipulation as a subset of clickjacking. We suggest that sufficient size be considered an additional criterion (in addition to traditional clickjacking prevention criteria like complete visibility [11, 19]) for the enabling of a sensitive UI element.

Simplification of secure UI layout tree. Prior work [18] proposes invariants for the interface layout tree that ensure a trusted path to every node and describes how to transform an invalid layout tree into a valid one. Our implementation experience shows this solution to be overly general. Embedded elements need not be attached to the layout tree in arbitrary locations; rather, they can always attach to the (system-controlled) root node and overlaid appropriately by the WindowManager (or equivalent). That is, the layout trees of separate principals need never be interleaved, but rather visually overlaid on top of each other, requiring no complex tree manipulations. Simplifying this approach is likely to make it easier and less error-prone for system developers to support secure embedded UI.

8.2 New Capabilities

We step back and consider the capabilities enabled by our implementation. In particular, the following scenarios were fundamentally impossible to support before our modifications to Android; LayerCake provides additional security properties and capabilities even beyond the Web, as we detail here.

Isolated Embedded UI. Most fundamentally, LayerCake allows Android applications to securely embed UI running in another process. Conceptually, this aligns the Android application model with the Web model, in which embedded cross-principal content is common. Especially as Android expands to larger devices like tablets, users and application developers will benefit from the ability to securely view and show content from multiple sources in one view.

Secure WebViews. It is particularly important that WebViews containing sensitive content run in their own process. While an Android WebView seems at first glance to be similar to an iframe, it does not provide the security properties to which developers are accustomed on the Web (as discussed in this paper and identified in prior

work [12, 13]). LayerCake matches and indeed exceeds the security of iframes—in particular, a SecureWebView can request that the system not deliver user input to it when it is not fully visible or sufficiently large, thereby preventing clickjacking attacks that persist on the Web.

Access Control Gadgets. Prior work [19] introduced ACGs for user-driven access control of sensitive resources like the camera or location, but that work does not provide concrete guidelines for how the necessary UI-level security properties should be implemented. This paper provides these details, and we hope that they will guide system developers to include ACGs in their systems. We particularly recommend that browser vendors consider ACGs in their discussions of how to allow users to grant websites access to sensitive resources [28].

8.3 Additional Issues

Finally, we discuss several issues unaddressed by LayerCake that must be considered in future work.

First is the issue of application dependencies, that is, how to handle the case when an application embeds an Activity from another application that is not installed. Possibilities include automatically bundling and installing dependencies (as also proposed by the authors of AdSplit [23]), giving the user the option of installing the missing application, or simply failing silently. This issue led the authors of AdDroid [17] to decide against running ads in their own process, but we argue that the security concerns of not doing so outweigh this issue. The concern that users might uninstall or replace ad applications to avoid seeing ads could be addressed by giving parent applications feedback when a requested embedded Activity cannot be displayed; applications relying on ads could then display an error message if the required ad library is not available. Updates and differences in library versions required by apps could be handled by Android by supporting multiple installed versions or simply by the ad libraries themselves.

Second is the issue of principal identification: a user cannot easily determine the source of an embedded interface (or even whether anything is embedded). This concern mirrors the Web today, where an iframe’s presence or source cannot be easily determined, and we consider this to be an important orthogonal problem.

9 Related Work

Finally, we consider additional related work not discussed inline.

In Section 4 and Figure 1, we explored existing implementations of embedded cross-application user interfaces [5, 17, 18, 23]. These systems have differing goals and employ a variety of techniques, but none fully meets the security requirements defined in [18] and expanded here. In particular, none of these approaches can, without

modification, support security-sensitive embedded user interfaces like ACGs [19]. The original ACG implementation built on interface-level security properties provided by the Gazelle browser operating system [29].

Others have explored the problem of clickjacking in more depth. One study [20] found that most framebusting techniques are circumventable, making them ineffective for preventing clickjacking. Other work [11] provides a comprehensive study of clickjacking attacks and defenses, presenting a solution (*InContext*) that relies on the browser to verify the visual context of sensitive UI elements. LayerCake could be extended to support *InContext* for additional clickjacking protection.

Our implementation relies on security properties provided by the Android WindowManager. Window system security has been explored previously by projects such as Trusted X [4] (an implementation of the X Window System [9] based on the Compartmented Mode Workstation requirements [30]) and the EROS Trusted Window System [22]. We extend this work by leveraging a secure window system to support secure cross-application UI embedding.

10 Conclusion

We have systematically considered the security requirements for embedded user interfaces, analyzing existing systems—including browsers, smartphones, and research systems—with respect to these requirements. While browsers have evolved to address many (though not all) of these requirements over time, Android-based implementations have not supported secure embedded interfaces. We thus created LayerCake, a modified version of the Android framework that supports cross-principal embedded interfaces in a way that meets our security goals. The resulting capabilities enable several important scenarios, including advertisement libraries, Facebook social plugins, and access control gadgets. Based on our exploration and implementation experience, we provide a concrete set of criteria and techniques that has to date been missing for system developers wishing to support secure interface embedding.

This paper, along with any updates, will be available at <https://layercake.cs.washington.edu/>.

11 Acknowledgements

We thank Roxana Geambasu, Alex Moshchuk, Bryan Parno, Helen Wang, and the anonymous reviewers for their valuable feedback on earlier versions. This work is supported in part by the National Science Foundation (Grant CNS-0846065 and a Graduate Research Fellowship under Grant DGE-0718124), by the Defense Advanced Research Projects Agency (under contract FA8750-12-2-0107), and by a Microsoft Research PhD Fellowship.

References

- [1] BARTH, A. Security in Depth: HTML5's @sandbox, 2010. <http://blog.chromium.org/2010/05/security-in-depth-html5s-sandbox.html>.
- [2] BEGEMANN, O. Remote View Controllers in iOS 6, Oct. 2012. <http://oleb.net/blog/2012/02/what-ios-should-learn-from-android-and-windows-8/>.
- [3] DIETZ, M., SHEKHAR, S., PISETSKY, Y., SHU, A., AND WALLACH, D. S. Quire: Lightweight Provenance for Smart Phone Operating Systems. In *20th USENIX Security Symposium* (2011).
- [4] EPSTEIN, J., MCHUGH, J., AND PASCALE, R. Evolution of a Trusted B3 Window System Prototype. In *IEEE Symposium on Security and Privacy* (1992).
- [5] ETTRICH, M., AND TAYLOR, O. XEmbed Protocol Specification, 2002. <http://standards.freedesktop.org/xembed-spec/xembed-spec-latest.html>.
- [6] FACEBOOK. Social Plugins. <https://developers.facebook.com/docs/plugins/>.
- [7] FACEBOOK. Like button requires confirm step, 2012. <https://developers.facebook.com/bugs/412902132095994/>.
- [8] FELT, A. P., HA, E., EGELMAN, S., HANEY, A., CHIN, E., AND WAGNER, D. Android permissions: user attention, comprehension, and behavior. In *8th Symposium on Usable Privacy and Security* (2012).
- [9] GETTYS, J., AND PACKARD, K. The X Window System. *ACM Transactions on Graphics* 5 (1986), 79–109.
- [10] GOOGLE. AdMob Ads SDK. <https://developers.google.com/mobile-ads-sdk/>.
- [11] HUANG, L.-S., MOSHCHUK, A., WANG, H. J., SCHECHTER, S., AND JACKSON, C. Clickjacking: Attacks and Defenses. In *21st USENIX Security Symposium* (2012).
- [12] LUO, T., HAO, H., DU, W., WANG, Y., AND YIN, H. Attacks on WebView in the Android system. In *27th Annual Computer Security Applications Conference* (2011).
- [13] LUO, T., JIN, X., ANANTHANARAYANAN, A., AND DU, W. Touchjacking Attacks on Web in Android, iOS, and Windows Phone. In *5th International Symposium on Foundations and Practice of Security* (2012).
- [14] MICROSOFT. User Account Control. microsoft.com/en-us/library/windows/desktop/aa511445.aspx.
- [15] MOSHCHUK, A., BRAGIN, T., DEVILLE, D., GRIBBLE, S. D., AND LEVY, H. M. SpyProxy: Execution-Based Detection of Malicious Web Content. In *16th USENIX Security Symposium* (2007).
- [16] MOTIEE, S., HAWKEY, K., AND BEZNOSOV, K. Do Windows Users Follow the Principle of Least Privilege?: Investigating User Account Control Practices. In *Symposium on Usable Privacy and Security* (2010).
- [17] PEARCE, P., FELT, A. P., NUNEZ, G., AND WAGNER, D. Ad-Droid: Privilege Separation for Applications and Advertisers in Android. In *ACM Symposium on Information, Computer and Communications Security (AsiaCCS)* (2012).
- [18] ROESNER, F., FOGARTY, J., AND KOHNO, T. User Interface Toolkit Mechanisms for Securing Interface Elements. In *25th ACM Symposium on User Interface Software and Technology* (2012).
- [19] ROESNER, F., KOHNO, T., MOSHCHUK, A., PARNO, B., WANG, H. J., AND COWAN, C. User-Driven Access Control: Rethinking Permission Granting in Modern Operating Systems. In *IEEE Symposium on Security and Privacy* (2012).
- [20] RYDSTEDT, G., BURSZTEIN, E., BONEH, D., AND JACKSON, C. Busting Frame Busting: A Study of Clickjacking Vulnerabilities on Popular Sites. In *IEEE Workshop on Web 2.0 Security and Privacy* (2010).
- [21] SCHECHTER, S., DHAMIJA, R., OZMENT, A., AND FISCHER, I. The Emperor's New Security Indicators. In *IEEE Symposium on Security and Privacy* (2007).
- [22] SHAPIRO, J. S., VANDERBURGH, J., NORTHUP, E., AND CHIZMADIA, D. Design of the EROS Trusted Window System. In *13th USENIX Security Symposium* (2004).
- [23] SHEKHAR, S., DIETZ, M., AND WALLACH, D. S. AdSplit: Separating Smartphone Advertising from Applications. In *21st USENIX Security Symposium* (2012).
- [24] SOPHOS LABS. Facebook Worm: Likejacking, 2010. <http://nakedsecurity.sophos.com/2010/05/31/facebook-likejacking-worm/>.
- [25] TANG, S., MAI, H., AND KING, S. T. Trust and Protection in the Illinois Browser Operating System. In *USENIX Symposium on Operating Systems Design and Implementation* (2010).
- [26] TEMPLEMAN, R., RAHMAN, Z., CRANDALL, D. J., AND KAPADIA, A. Placeraid: Virtual theft in physical spaces with smartphones. *CoRR abs/1209.5982* (2012).
- [27] W3C. Same Origin Policy. http://www.w3.org/Security/wiki/Same_Origin_Policy.
- [28] W3C. Device API Working Group, 2011. <http://www.w3.org/2009/dap/>.
- [29] 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).
- [30] WOODWARD, J. P. L. Security Requirements for System High and Compartmented Mode Workstations. Tech. Rep. MTR 9992, Revision 1 (also published by the Defense Intelligence Agency as DDS-2600-5502-87), The MITRE Corporation, Nov. 1987.
- [31] YEE, K.-P. Aligning Security and Usability. *IEEE Security and Privacy* 2(5) (Sept. 2004), 48–55.

Automatic Mediation of Privacy-Sensitive Resource Access in Smartphone Applications

*Benjamin Livshits and Jaeyeon Jung
Microsoft Research*

Abstract

Mobile app development best practices suggest that developers obtain opt-in consent from users prior to accessing potentially sensitive information on the phone. We study challenges that mobile application developers have with meeting such requirements, and highlight the promise of using new automated, static analysis-based solutions that identify and insert missing prompts in order to guard otherwise unprotected resource accesses. We find evidence that third-party libraries, incorporated by developers across the mobile industry, may access privacy-sensitive resources without seeking consent or even *against* the user’s choice. Based on insights from real examples, we develop the theoretical underpinning of the problem of mediating resource accesses in mobile applications. We design and implement a graph-theoretic algorithm to place mediation prompts that protect every resource access, while avoiding repetitive prompting and prompting in background tasks or third-party libraries.

We demonstrate the viability of our approach by analyzing 100 apps, averaging 7.3 MB in size and consisting of dozens of DLLs. Our approach scales well: once an app is represented in the form of a graph, the remaining static analysis takes under a second on average. Overall, our strategy succeeds in about 95% of all unique cases.

1 Introduction

Privacy on smartphones is far from being a theoretical issue: a popular iOS application, Path, had been found to upload the entire address book of an iPhone user by default; similarly, a number of high-profile incidents [1–3] show negative consequences for mobile applications that surreptitiously collected privacy-sensitive information about users without explicit consent. Furthermore, a recent survey of 714

cell phone users shows that 30% of the respondents had uninstalled an application because they discovered that the application in question was collecting personal information they did not wish to share [20].

Runtime *consent dialogs* (sometimes called runtime permission prompts) are commonly used by mobile applications to obtain a user’s explicit consent *prior* to accessing privacy-sensitive data. However, mobile operating systems differ in terms of their approach to raising these consent dialogs. iOS implements OS-level consent dialogs which are raised when accessing GPS location, contacts stored on the phone, as well as a few other key resources. These dialog boxes are far from being “no-ops” for the user: A recent study of hundreds of iPhone users shows that 85% of them exercised this control to deny at least one application from accessing location data [13]. However, in the absence of OS-level support, application developers can individually implement opt-in consent dialogs for enhancing the overall privacy for end-users.

This paper focuses on a number of technical challenges that arise when mobile application developers determine the right place to insert runtime prompts within an application. First, minimizing the runtime *frequency* of consent dialogs is important, as repetitive prompts tend to habituate users to blindly accept the terms [7]. However, to protect user privacy, every single attempt to access sensitive information should be guarded with a prompt. Second, apps should provide *just-in-time* prompts in order for it to make sense to the user within the application context. If prompts are placed early, e.g., at *install time*, users may forget about granted permissions, leading to unpleasant surprises because of data access performed by the app, especially when it runs in the background [21].

The aim of this paper is to formalize the problem of placing runtime consent dialogs within a mo-

bile application, and to propose a solution for automatic and correct prompt placement. We try to both 1) find missing prompts and 2) propose a valid prompt placement when prompts are missing.

1.1 Analysis Design Philosophy

While it is possible to use dynamic analysis to observe missing prompts at runtime, this approach is fraught with significant challenges. The traditional challenge is low path coverage, which can be alleviated with path exploration techniques such as symbolic execution, but never completely fixed. Other, more technical, challenges related to running UI-based mobile apps automatically also remain.

Because we aim to provide a technique that would err on the side of safety, we do not believe runtime analysis is suitable. To this end, we propose a new scalable static analysis algorithm to automatically find places for inserting prompts if they are missing. Our solution scales well with application size and does not require any changes to the underlying operating system.

Given the inherent nature of static analysis techniques and the complexity of both the applications and the execution environment, our tool may produce false positives. However, at the worst, these false positives will result in double-prompts that occur at most once per application. We believe this to be a considerable improvement over the current error-prone practice of manual prompt placement. Our approach in this paper may not be fully sound due to issues such as reflection (see Section 4); however, our goal is to be as sound as possible. Our evaluation in Section 6 does not reveal any false negatives.

Finally, note that our target is benign, but potentially buggy non-obfuscated apps. If the app writer tries to either obfuscate their code or take advantage of features that are not treated conservatively (such as reflection) to hide control flow, the precision and soundness of our analysis will suffer. Luckily, the presence of obfuscation is relatively easy to detect [22].

1.2 Contributions

Our contributions are three-fold:

- Using a set of .NET WP (Windows Phone) applications, we study how existing applications implement resource access prompts. We note that some advertising libraries access location data without a prompt.
- We propose a two-prong static analysis algorithm for correct resource access prompt placement.

ment. We first attempt to use a fast, dominator-based placement technique. If that fails, we resort to a slower but more exhaustive backward search.

- We evaluate our approach to both locating missing prompts and placing them when they are missing on 100 apps. Overall, our two-prong strategy of dominator-based and backward placement succeeds in about 95% of all unique cases. Our analyses run in seconds, making it possible to run them as part of the app submission process.

Our analysis reveals that some application developers fail to show the proper set of prompts, showing the difficulty and ineffectiveness of manual placement. Frequently, the issue that exacerbates this situation is that resource access takes place within third-party libraries shipped as bytecode, making them more difficult to reason about largely placing them outside developer’s control.

1.3 Paper Organization

The rest of this paper is organized as follows. We discuss case studies of real applications and challenges associated with proper prompt placement in Section 2. We then formulate the problem and provide much of the insight for our proposed solution in Section 3. We discuss the implementation of the algorithms in Section 4. Results from an experimental study are described in Section 5 and further discussed in Section 6. We summarize related work in Section 7 and conclude in Section 8.

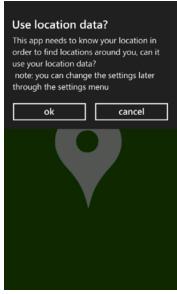
2 Background

We first provide three motivating case studies in Section 2.1 and then provide intuition for the complexity of the problem in Section 2.2.

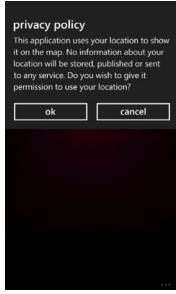
2.1 Motivating Case Studies

We begin by discussing several interesting real-world examples, which illustrate how existing WP apps mediate access to location data. One of the ways in which the WP SDK exposes location access API to applications is through the `GeoCoordinateWatcher` class in the `System.Device.Location` namespace. Prompts are created with a call to `MessageBox.Show`, with the text of the prompt provided by the developer.

Figure 1 shows screen-shots of three applications — `AroundMe`, `Burger King (inoffiziell)`, `LumiClock` — immediately before these applications in-



(a) AroundMe



(b) Burger King



(c) LumiaClock

Figure 1: Screen-shots of three examined applications. The first two applications display a location prompt prior to invoking location APIs. The third application never shows a location prompt; the screen-shot was captured when we detected the first time that a location API was invoked by the application.

App	Resource accesses	APIs used	Libraries
AroundMe	2	TryStart, get_Position	AroundMe.dll
Burger King	5	Start, get_Position	BurgerKing.dll, GART.dll
LumiaClock	2	Start, get_Position	SOMAWP7.dll

Figure 2: Location accesses found in three apps.

voked location access API. We picked these three apps from the WP Store, filtering for apps that use GPS location data. Each application consists of a set of DLLs and resources. We have disassembled the applications and inspected the code to find instances of location API invocations. Figure 2 shows (1) the number of location access points observed in each of the three applications; (2) which location API is used; and (3) which libraries call the location API.

As shown in Figure 2, location access happens both in application code and in third-party libraries. For instance, **GART.dll** is a library that provides augmented reality features and **SOMAWP7.dll** is a library that provides advertising to WP applications. Not surprisingly, the use of location data by third-party libraries complicates access mediation, as third-party libraries often come as a black box to application developers. The following in-depth analysis illustrates the issue.

Case 1 (proper protection): Location accesses are contained only in the application code and properly mediated by a runtime consent dialog. The code snippet in Figure 3(a) is from the **AroundMe** application. As shown in the code below, this application

```
public static bool AroundMe.App.CheckOptin() {
    if (((Option)Enum.Parse(typeof(Option), Config.GetSetting
        (SettingConstants.UseMyLocation), true)) == Option.Yes) {
        return GetCurrentCoordinates();
    }
    if (MessageBox.Show("This app needs ...",
        "Use location data?", MessageBoxButton.OKCancel) == MessageBoxResult.OK) {
        Config.UpdateSetting(new KeyValuePair<string, string>
            (SettingConstants.UseMyLocation, Option.Yes.ToString()));
        return GetCurrentCoordinates();
    }
    ...
}
```

(a) Illustration for Case 1

```
public BurgerKing.View.MapPage() {
    this.InitializeComponent();
    base.DataContext = new MapViewModel();
    this.BuildApplicationBar();
    if (AppSettings.Current.UseLocationService){
        this.watcher = new GeoCoordinateWatcher();
    }
    ...
}

protected virtual void GART.Controls.ARDisplay.
    OnLocationEnabledChanged(
        DependencyPropertyChangedEventArgs e)
{
    if (this.servicesRunning) {
        if (this.LocationEnabled) {
            this.StartLocation();
        } else {
            this.StopLocation();
        }
    }
}
```

(b) Illustration for Case 2

```
public SomaAd()
{
    ...
    this._locationUseOK = true;
    ...
    if (this._locationUseOK) {
        this.watcher = new GeoCoordinateWatcher
            (GeoPositionAccuracy.Default);
        this.watcher.MovementThreshold = 20.0;
        this.watcher.StatusChanged += new EventHandler<
            <GeoPositionStatusChangedEventArgs>(
            this.watcher_StatusChanged);
        this.watcher.Start();
    }
}
```

(c) Illustration for Case 3

Figure 3: Illustrative cases for Section 2.1.

invokes `GetCurrentCoordinates()` only after the user clicks the `OK` button as shown in Figure 1.

Case 2 (partial protection): Location accesses are spread across application and third-party code and only accesses by application code are protected by runtime consent dialog. The code snippet in Figure 3(b) is from the **BurgerKing** application. The consent dialog shown in Figure 1 only affects `AppSettings.Current.UserLocationService` and leaves `GART.Controls.ARDisplay.StartLocation()` unprotected. Using network packet inspection, we con-

```

while(P){
    11 = getLocation();
}
    (a) original

prompt();
while(P){
    11 = getLocation();
}
    (b) static prompt

while(P){
    if(not-yet-prompted-for-location){
        prompt();
    }
    11 = getLocation();
}
    (c) dynamic check

```

Figure 4: Resource access in a loop.

firmed that the application accesses and transmits location using the `GART` component *even* when the `Cancel` button is clicked.

Case 3 (no protection): Location accesses are only present in third-party code and the application provides no consent dialogs. The following code snippet is from the `LumiaClock` application. The application has no location features. Although the third-party code `SomaAd` exposes a flag to protect location access, the application appears unaware of it. Moreover, the `SomaAd` component enables the flag, `_locationUseOK` by default, as shown in Figure 3(c).

Summary: In summary, the case studies above demonstrate that properly protecting location access is challenging because multiple components, including third-party libraries, are involved in accessing sensitive resources. The current practice often fails in providing adequate privacy protection, as some applications do not honor the user’s choice (as shown in case 2) or do not obtain the user’s consent prior to acquiring privacy-sensitive information.

2.2 Challenges

Next, we dive into the properties that we want to ensure, while deciding where to place missing prompts via static analysis. Naively, one might suspect that prompt placement is a fairly trivial task, reducing to (1) finding resource access points and (2) inserting prompts right in front of them. In reality, situation is considerably more complex. In this section, we systematically investigate the challenges we need to overcome in order to provide a satisfactory solution.

1) Avoiding double-prompts: We need to avoid prompting the user for access to resource R more than once on a given execution path. This is a harder

problem than it might initially seem; indeed, consider the following code:

```

if(P) 11 = getLocation();
12 = getLocation();

```

There are two location access points and two ways to avoid duplicate prompts. One is to introduce a boolean flag to keep track of whether we have prompted for the location already:

```

flag = true;
if(P){
    prompt();
    flag = true;
    11 = getLocation();
}
if(!flag){
    prompt();
    12 = getLocation();
}

```

The disadvantage of this approach is that it requires introducing extra runtime instrumentation to perform this sort of bookkeeping. A fully static approach involves rewriting the original code by “folding” the second prompt into the `if`:

```

if(P){
    prompt();
    11 = getLocation();
    12 = getLocation();
} else{
    prompt();
    12 = getLocation();
}

```

This approach has the advantage of not having to introduce extra bookkeeping code. The disadvantage is replication of the existing code across the branches of the `if`, which leads to extra code growth.

The problem of double-prompts can be exacerbated. Figure 4a illustrates the challenge of placing a prompt within a loop. Placing the prompt before the loop as in Figure 4b is not valid if the loop never executes. Placing the prompt within the loop body will lead to execution on every iteration. However, a simple dynamic check will ensure that the location prompt is not shown more than once (Figure 4c).

2) Sticky prompts: Applications frequently make user-granted permissions persistent and avoid duplicate prompts, by saving the prompt status to the app’s isolated storage, as illustrated in Figure 5. Here the challenge comes in both recognizing existing “sticky” prompts in app code and in making inserted prompts sticky, as discussed in Section 4.3.

3) Avoiding weaker prompts: Suppose there are two resources r_1, r_2 such that r_2 is less sensitive than r_1 . If an app has already prompted the user for

access to r_1 , it should avoid prompting the user for access to resource r_2 . For instance, if an app already has requested access to fine-grained location, there is no need to prompt for access to coarse-grained location. Note that in the current version of the WP operating system, there is no difference in capabilities between fine- and coarse-grained locations; both require the `ID_CAP_LOCATION` capability in the app manifest. However, in the future more fine-grained capabilities subsuming one another may evolve, as they have on Android. Moreover, it is still possible and perhaps even desirable to distinguish between fine- and coarse-grained locations when prompting at runtime, even though they are treated the same at installation time.

4) Avoiding prompts in background tasks: WP apps provide non-interactive background tasks. These are often used for polling remote servers and other tasks that do not require access to the user’s screen beyond, perhaps, a live tile of the app. We cannot raise dialog boxes within background tasks. To properly determine where the prompts should be located, we should compute the call graph and determine what foreground code precedes the code within background tasks.

5) Avoiding prompts in libraries: Given that libraries are often shipped in the form of bytecode and are updated separately from the rest of the applications, we choose to avoid placing prompts in library code. This approach allows developers to examine prompt placement within their own code, and to alleviate the need to keep custom-modified versions of third-party libraries such as `SOMAWP7.dll`, which can make error reporting, debugging, and sharing libraries across apps a challenge.

```

if (MessageBox.Show(
    "This app needs to know your location
    in order to find locations
    around you, can it use your location data?
    note: you can change the settings later
    through the settings menu",
    "Use location data? ", 1) == 1)
{
    Config.UpdateSetting(
        new KeyValuePair<string, string>(
            SettingConstants.UseMyLocation,
            Option.Yes.ToString()));
    return
        GetCurrentCoordinates();
}

```

Figure 5: Sticky location prompt.

3 Overview

A recent spate of research efforts is centered around detecting undesirable *information flows*, i.e. sensitive data like contacts leaving the phone, usually via the network (e.g., [9, 10]). Reasoning about these kinds of leaks involves understanding inter-procedural data flow within the app and perhaps even across different apps. Data flow analysis of this kind is a known difficult problem which, despite a great deal of work on both the static and runtime sides has not yet found widespread practical deployment [24].

In the context of mobile apps, there is another aspect further complicating this problem. Even if there is in fact a perfect *mechanism* for precisely and efficiently tracking inter-procedural data flow, a viable *policy* is hard to come by. Indeed, how does a tool automatically distinguish between a Yelp app that shares GPS location information with a back-end server to obtain restaurant listings from (a potentially malicious) flashlight app that obtains the same GPS information and shares it with an ad server? Constructing a robust policy is not trivial. Our paper rather focuses on providing a method for assisting application developers in checking their apps against the currently accepted practice of obtaining consent prior to accessing potentially sensitive user data on the phone and in fixing problems before submitting apps to a marketplace. Note that our work in this paper is in the control flow, *not* data flow space; we want to reason about whether the acquisition points for sensitive content are well-protected. In this section we first formulate the problem of prompt placement and then discuss some approaches for computing a valid placement.

3.1 Graph Representation

As is typical in static analysis, it is helpful to represent the program in the form of a graph, to abstract away many unnecessary features of the original source or bytecode representation.

Since our goal is to reason about prompts “guarding” resource access points, we choose a representation similar to a control-flow graph. Because both prompts and resource accesses take the form of method calls, we find it convenient to augment the traditional notion of *basic blocks* to treat call sites specially. We use the term *enhanced basic block* to emphasize the difference in construction. An enhanced basic block is different from a basic block in that only the first and last of its instructions can be (method) calls. Consequently, call instructions exist

in a block of their own. (First and last instructions can also be jumps, just as in the case of regular basic blocks.)

Our representations also need to be inter-procedural: we need to be able to handle prompts that are located outside of the method in which the resource access takes place. This is especially necessary given that WP apps are written in .NET, where methods generally tend to be small. We therefore augment the control flow graph with call and return edges denoted as C below.

Definition 1 A resource access prompt placement problem is defined as follows. Let $\mathcal{P} = \langle N, A, B, E, C, \mathcal{L} \rangle$ be a tuple with the following components:

- N : set of enhanced basic blocks in the program consisting of a sequence of instructions $N = n_1, n_2, \dots, n_k$. For simplicity, we assume that graph G has unique entry and exit nodes $N_{\text{entry}}, N_{\text{exit}} \in N$.
- $A \subset N$: set of resource access points;
- $B \subset N$: set of enhanced basic blocks located within background tasks and (third-party) libraries; we assume that N_{entry} and N_{exit} are outside background tasks and libraries;
- E : intra-procedural control flow edges;
- C : inter-procedural call and return edges.
- $\mathcal{L} = \langle R, \wedge \rangle$: the semi-lattice of access permissions with meet operator \wedge^1 .

Intuitively, this representation is an expanded inter-procedural control flow graph $G = \langle N, E \cup C \rangle$.

3.2 Valid Placement

Based on the challenges described in Section 2.2, we proceed to formulate what it means to have a *valid* placement of resource access prompts.

Definition 2 We say that placement $P \subset N$ is a valid placement for a prompt placement problem $\mathcal{P} = \langle N, A, B, E, C, \mathcal{L} \rangle$ if the following conditions hold for every runtime execution of the app:

- **Safe:** Every access to resource $r \in R$ is preceded by a prompt check for r .
- **Visible:** No prompt is located within a background task or a library.

¹We assume that in the general case it is possible for permissions to subsume one another, like in the case of fine- and coarse-grained GPS locations, giving rise to a partial order, although we currently do not strictly need this kind of support in our implementation.

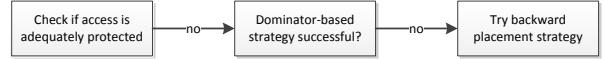


Figure 6: Analysis steps.

- **Frugal:** Prompt for $r \in R$ is never invoked unless it is either followed by a call to `get(r)` or an exception occurs².
- **Not-repetitive:** Prompt for permission $r_2 \in R$ is never invoked if permissions for $r_1 \in R$ have already been granted and $r_2 \sqsubseteq r_1$ (that is, r_1 is at least as or more permissive than r_2).

3.3 Solution Outline

We provide intuition for our solution in the remaining sections; Section 4 gives the actual algorithms. Figure 6 shows the overall flow of our analysis. Given a graph with well-identified resource access points, a *safe* placement is relatively easy to come up with. The main obstacle is the fact that we cannot always put prompts right before accesses, because sometimes accesses are within background tasks or, more frequently, in libraries (violating the *visible* requirement).

Intuitively, we can start with resource access points A and move the prompts up until we are outside of background tasks. The downside of this approach is a possibility of moving these prompts too far (to the beginning of the app in the most extreme case), which would violate the *frugal* requirement. This gives rise to a notion of a prompt being *needed* at a particular point, for which we use the term *anticipating*, common in compiler literature [4]. By way of example, for the code snippet in Figure 7, location access is anticipating before line 3, but it is *not* anticipating before the `if` on line 2, because of the `else` branch. So placing the prompt on line 1 leads to unnecessary prompting, violating the requirement of being *frugal*.

```

1.
2. if(P){
3.     var l = getLocation();
4. } else {
5.     x++;
6. }
  
```

Figure 7: Conditional location access.

²Note that this notion of frugality is optimized for runtime savings, not necessarily savings in terms of code size.

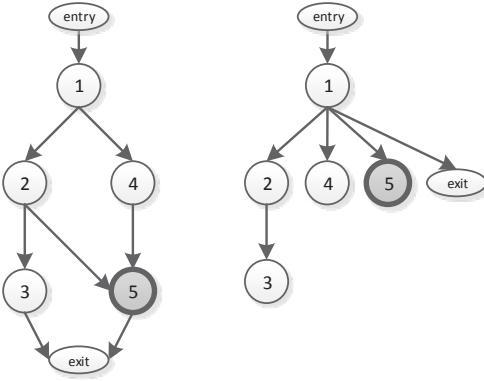


Figure 8: Graph (left) and its dominator tree (right). Node 5 is a resource access node within a library.

Definition 3 We say that basic block $B \in N$ is r -anticipating if every path from B to N_{exit} passes through a resource access of type r .

Intuitively, placing prompts for resource accesses of type r at r -anticipating nodes is necessary because these nodes are guaranteed to require them eventually; in other words, these placements will be *frugal*.

Finally, the discussion so far has not considered the case of prompts granting permissions of different “strength”, resulting in potentially unnecessary prompts. This suggests that the notion of being anticipating should be defined not globally, but with respect to a particular kind of resource, taking into account the lattice of resource access permissions.

Dominator-based Approach: Using the notion of *dominators* in the graph [4] we can abstract away unnecessary details. Recall that we say that node $d \in N$ dominates node $n \in N$ if every path from $N_{\text{entry}} \rightarrow n$ passes through d . Dominator relationships induce a dominator tree over the set of nodes N . An example of such a dominator tree for a graph in Figure 8a is shown in Figure 8b.

By this definition, dominator-based placement is an easy way to “block” access to a particular resource access. The most immediate approach is to place prompts on the nodes dominating the resource access node.

Of course, since we want a placement as close as possible to the access point, we will prefer the *immediate* dominator of the resource access node. By definition, we will have a *safe* placement, because $\forall a \in A$, every path from $N_{\text{entry}} \rightarrow a$ must pass through $\text{idom}(a)$, the immediate dominator of a . This simple approach suffers from two problems:

- Background and library nodes can invalidate immediate dominator-based placement. To deal with the issue of visibility, we can shift the prompts up in the dominator tree.
- Immediate dominator-based placement can violate the *frugal* condition. Indeed, consider the location access at line 3 in Figure 7. Its immediate dominator is the *if(P)* node. However, this node is not location-anticipating, because the *else* branch is *not* accessing the location.

A viable approach is therefore to start at the resource access node and walk up the dominator tree until we encounter a node that is not in the background or a library. We are guaranteed to encounter such a node eventually, because sooner or later we will encounter N_{entry} , which is a foreground non-library node by Definition 1.

For the graph in Figure 8, node 5 is a library node. Nodes 1 and N_{entry} are in the cover for node 5. Node 1 is the immediate cover of 5. Our approach, therefore, will choose node 1 for a prompt protecting node 5, but, unfortunately, this placement will violate the frugality condition.

Backward Placement: Sometimes dominator-based placement will backtrack “too far” in the graph to become unnecessary — in other words, not *frugal*. In these cases, we propose an alternative strategy called *backward placement*, which often avoids this problem. Backward placement explores the predecessors of the resource access node and find an individual *separate* place for a prompt for each of them. For node 5 in Figure 8, both predecessors 2 and 4 present valid placement opportunities, which are also frugal. Frequently, the backward placement approach will yield a valid placement. The concern with this strategy is two-fold:

- This approach may not scale well, as it involves an exponential graph search. While this is true in general, in practice we frequently find a valid placement within several nodes, as detailed in Section 5.
- More prompts will be created compared to the dominator-based approach. (Indeed, in our Figure 8 example, we inserted two nodes instead of one.) More inserted prompts may increase the size of the rewritten app and may also make manual validation of placement results more challenging and time-consuming.

Unlike dominator-based placement, there is a possibility of passing through prompt placement nodes multiple times at runtime. To see this, consider adding a backward edge from 3 → 1 in Figure 8.

```

1: function INSERTPROMPT( $G, a, ant, idom$ )
2: if  $\neg$ HasPrompt( $G, a.Type, a$ ) then
3:   // Try dominator-based first
4:    $Placement \leftarrow \emptyset$ 
5:   success  $\leftarrow$  InsertPrompt-D( $G, a, ant, idom$ )
6:   if  $\neg$ success then
7:     // Try backward placement next
8:      $Placement \leftarrow \emptyset$ 
9:     InsertPrompt-B( $G, a, ant$ )
10:    end if
11:   end if
12: end function
13:
14: //Dominator-based placement
15: function INSERTPROMPT-D( $G, a, ant, idom$ )
16:    $n \leftarrow a$ 
17:   while  $n \neq N_{entry}$  do
18:     if IsAnticipating( $n, a.Type, ant$ )  $\wedge$ 
19:        $n \notin G.Background \wedge n \notin G.Libraries$ 
20:     then
21:        $Placement \leftarrow Placement \cup \{n\}$ 
22:       return true
23:     else
24:        $n \leftarrow idom(n)$  ▷ Proceed to the immediate dominator
25:     end if
26:   end while
27:   return false
28: end function
29:
30: //Backward search placement
31: function INSERTPROMPT-B( $G, a, ant$ )
32:   Occurs-check( $a$ ) ▷ Prevent infinite recursion
33:   if  $\neg$ IsReachable( $a$ )  $\vee$  (IsAnticipating( $a, a.Type, ant$ )
34:      $\wedge a \notin G.Background \wedge a \notin G.Libraries$ )
35:   then
36:      $Placement \leftarrow Placement \cup \{a\}$ 
37:     return true
38:   else
39:     for all  $p \in G.predecessors(a)$  do ▷ Predecessors
40:       success  $\leftarrow$  InsertPrompt-B( $G, p, ant$ )
41:       if  $\neg$ success then
42:         return false ▷ One of the predecessors failed
43:       end if
44:     end for
45:     return true ▷ All predecessors succeeded
46:   end if
47: end function
48:
49: //Helper function to check if  $n$  is anticipating for  $r \in R$ 
50: function ISANTICIPATING( $n, r, ant$ )
51:    $r' \leftarrow ant(n)$  ▷ Computed prompt type at  $n$ 
52:   return  $r \sqsubseteq r'$  ▷ True if  $r'$  is more permissive
53: end function

```

Figure 9: Insertion of resource access prompts. G is the graph; a is the access node; $ant : N \rightarrow 2^R$ is the anticipating lookup map computed as specified in Figure 10, and, finally, $idom$ is the immediate dominator relation.

This edge does not affect the dominator tree or dominator-based placement. If we place prompts at nodes 2 and 4 for resource access at node 5, there is a possibility of encountering the prompt at node 2 multiple times as we go through the loop $1 \rightarrow 2 \rightarrow 3$. This kind of double-prompting violates the non-repetitive condition in Definition 2. A simple way to address this is to record user consent in app’s isolated storage for both the current runtime session and future app invocations, as shown in Section 4.3.

Semi-lattice	L	2^R , the power set of R
Top	\top	\emptyset
Initial value	$init(n)$	\emptyset
Transfer func.	$TF(n)$	$\begin{cases} \text{add } r \text{ to set} & \text{if } n \text{ is an access} \\ \text{identity} & \text{for } r \in R \\ & \text{otherwise} \end{cases}$
Meet operator	$\wedge(x, y)$	$\text{union } x \cup y$
Direction		backward

Figure 10: Dataflow analysis formulation for computing anticipating nodes: $\forall n \in N$, we compute the set of resource types that node n is anticipating.

3.4 Placement Algorithm

In our evaluation section, we will examine the trade-offs between the dominator-based and backward placement strategies. To summarize, this is an outline of our placement approach:

1. For every $r \in R$ and every node $n \in N$, compute its r -anticipating value $A_r(n)$.
 2. Merge values by meeting them in the semi-lattice $\mathcal{L} = \langle R, \wedge \rangle$ for all resource types:
- $$A(n) = \bigwedge_{r \in R} A_r(n)$$
3. For every resource access a of type r , use a backward search to find if it is adequately covered by existing prompts of type r' such that $r \sqsubseteq r'$.
 4. If not, proceed to insert a prompt of type $A(n)$ using either a dominator-based or a backward placement strategy.

Anticipating values can be calculated using a simple data-flow computation, in the style of the Dragon book [4]. A formulation of this analysis is shown in Figure 10 in the form of a table traditional for succinctly representing data-flow problems. The advantage of such a formulation is that it runs in linear time, given a lattice of finite height (and size), and that most compiler frameworks already provide a data-flow framework into which this kind of analysis can be “dropped”.

There is some flexibility when it comes to the last step. Indeed, we can choose to use a dominator-based, or a backward placement strategy, or some combination. In our implementation, we try the dominator strategy first to see if it yields a valid placement and, failing that, resort to the backward strategy. This hybrid approach is shown in the function INSERTPROMPT in Figure 9. Note that if placement is successful, the outcome is stored in the $Placement \subset N$ set.

INSERTPROMPT-B has an occurs-check on line 32 to avoid the possibility of infinite recursion for

```

1: //Checks for existing prompts
2: function HASPROMPT( $G, r, a$ )
3:   Occurs-check( $a$ )                                ▷ Prevent infinite recursion
4:   if  $a \in G.\text{Prompts}$  then
5:      $r' \leftarrow a.\text{Type}$ 
6:      $\text{adequate} \leftarrow (r \sqsubseteq r')$       ▷ Existing prompt at least as
    permissive as needed?
7:     if  $\text{adequate}$  then
8:       return true          ▷ Check if adequately protected
9:     end if
10:    end if
11:  end if
12:  //Explore all predecessors in turn
13:  for all  $p \in G.\text{predecessors}(a)$  do
14:     $\text{success} \leftarrow \text{HasPrompt}(G, r, p)$ 
15:    if  $\neg \text{success}$  then
16:      return false          ▷ One of the predecessors failed
17:    end if
18:  end for
19:  return true          ▷ All predecessors succeeded
20: end function

```

Figure 11: Checking for resource access prompts. G is the graph; r is the resource type; a is the access node.

graphs with loops, which are encountered in the process of backward exploration. If the current node is not reachable from non-library code as indicated by *IsReachable*, we return *true*. We discuss the challenges of fast backward computation in Section 4.2.

3.5 Checking For Existing Prompts

Note that before we choose to insert prompts we need to make sure they are in fact missing as shown on line 2 of Figure 9. Doing so requires a backward search, as shown in Figure 11. Note that in practice, HASPROMPTS frequently returns *false*, failing quickly without exploring the entire set of predecessors. Section 4.2 demonstrates how this search can be made faster.

```

1: function CREATEPLACEMENT( $G, ant, idom$ )
2:   for all  $a \in G.\text{Accesses}$  do
3:      $\text{success} \leftarrow \text{InsertPrompt}(G, a, ant, idom)$ 
4:     if  $\neg \text{success}$  then
5:       return false
6:     else
7:       for all  $p \in Placement$  do
8:          $\text{Prompts} \leftarrow \text{Prompts} \cup \langle p, ant(a) \rangle$ 
9:       end for
10:      end if
11:    end for
12:  end if
13:  // All clear: proceed with the placement
14:  for all  $\langle n, t \rangle \in \text{Prompts}$  do
15:    InsertAtNode( $n, t$ )
16:  end for
17:  return true
18: end function

```

Figure 12: Putting it all together: creating an overall prompt placement for graph G .

3.6 Proof Sketch

The algorithm that pulls everything together to create a placement is shown in Figure 12. We first check that whether there is indeed a valid placement for all resource accesses. Once this is ensured, we proceed to modify the underlying graph by inserting prompts at appropriate places. Note that prompt insertion is only attempted if they are in fact missing, as ensured by the check on line 2 of Figure 9. The details of runtime instrumentation are given in Section 4.3. The structure of the algorithm allows us to reason about the resulting placement.

Theorem 1 *The placement of prompts above is in fact valid if the placement routine CREATEPLACEMENT returns true.*

Proof sketch: It is easier to consider each correctness property in turn. We will refer to code lines in Figure 9 unless indicated otherwise.

Safe: We need to ensure that every access a to resource r is preceded by a prompt check for r . The call to INSERTPROMPT must have returned true for resource access a . This is because either the dominator-based or backward strategy was successful. If the dominator-based strategy succeeded, there was a non-background, non-library node dominating a which is also anticipating for $a.\text{Type}$. The check on line 18 maintains this invariant. If the dominator-based strategy failed and the backward strategy succeeded, this is because *every* path from a to N_{entry} has encountered a placement point which satisfied the check on line 33, providing adequate protection for the access at a .

Visible: No prompt is placed within a background task or library code. This is true by construction because of checks on lines 19 and 34.

Frugal: Placement only occurs at anticipating nodes because of checks on lines 18 and 33.

Not-repetitive: Prompt for $r_2 \in R$ is never invoked if permissions for r_1 have already been granted and $r_2 \sqsubseteq r_1$. This property is maintained by a combination of three steps: (1) merging in Step 2 on the overall algorithm, (2) check on line 52 and (3) the runtime “sticky” treatment of prompts that avoids double-prompting for the same resource type further explained in Section 4.3.

4 Implementation Details

Our current implementation of the static analysis described in this paper involves dealing with a variety of practical details, some of which are fairly common in bytecode-based static analysis tools, whereas

others are quite specific to our setting of WP apps written in .NET.

A significant part of the implementation involves building a graph on which to perform our analysis. Intra-procedurally, we parse the .NET bytecode to construct basic blocks; we terminate them at method calls to simplify analysis. For call graph construction, we use a simple class hierarchy analysis (CHA) to resolve virtual calls within the program. We also construct a dominator tree as part of graph construction, as we need it later. In many cases, the resulting graphs have enough precision for our analysis.

4.1 Reflection & Analysis Challenges

WP applications are distributed as XAP files, which are archives consisting of code in the form of bytecode DLLs, resources such as images and XAML, and the app manifest, which specifies requested capabilities, etc. Unsurprisingly, various reflective constructs found in WP apps create challenges for our analysis. While we outline some of the details of our solutions below, constructing precise static call graphs for mobile apps remains an ongoing challenge, and require further research.

Analysis imprecision usually does not stem from the underlying call graph construction approach, which could be alleviated through pointer analysis, which generally provides sufficient precision for call graph construction, but in challenges specific to complex WP apps, as discussed below.

Event handlers: The code below illustrates some complications posed by event handlers.

```
static void Main(string[] args) {
    AppDomain.CurrentDomain.ProcessExit += 
        new EventHandler(OnProcessExit);
}

// library code
static void OnProcessExit(object sender, EventArgs e) {
    // location access
    var watcher =
        new System.Device.Location.GeoCoordinateWatcher();
    var pos = watcher.Position;
}
```

By default, method `OnProcessExit` does not have any predecessors in the call graph. At runtime, it may in fact be called from a variety of places, which is not easy to model as part of call graph construction. However, it may not be called *before* the event handler is registered in method `Main`. Our solution is to augment the call graph construction code to create a special invocation edge from the registration site to `OnProcessExit`. The analysis will then be able to place the prompt right before the registration in

method `Main`, which makes a significant difference in our ability to find successful placements.

Actions and asynchronous wrappers: Another similar form of delayed execution in WP apps is `actions` (`System.Action`) and its asynchronous cousin `System.AsyncCallback`, which are effectively wrappers around delegates registered for later execution. We deal with actions in a way that is similar to event handlers.

XAML: A particular difficulty for analysis stems from the use of declarative UIs specified in XAML, an XML-like language that combines an easy-to-read UI specification with “hooks” into code. XAML is compiled into special resources that are embedded into an app’s DLLs. When the method `InitializeComponent()` is called on the class specified in XAML, it proceeds to register events that are specified declaratively, as shown in a XAML snippet below:

```
1   <phone:PhoneApplicationPage.ApplicationBar>
2       <shell:AppBar IsVisible="True">
3           <shell:AppBar.MenuItems>
4               <shell:AppBarMenuItem Text="Settings"
5                   Click="SettingsClick" />
6           </shell:AppBar.MenuItems>
7       </shell:AppBar>
8   </phone:PhoneApplicationPage.ApplicationBar>
```

Event handler `SettingsClick` should be properly registered so that it can later be invoked.

Alas, some aspects of declarative app specification defy static analysis. A typical example is navigation between an app’s pages.

```
1   base.NavigationService.
2       Navigate(new Uri(
3           "/VenueByGeo.xaml?mc=" + this.strMenuCode +
4           "&t=" + this.strToken,
5           UriKind.RelativeOrAbsolute));
```

Statically, we do not know which page will be navigated to, and, consequently, which `OnNavigatedTo` event handler will be called. To avoid polluting the call graph, we only link up page navigation when the destination is a string constant. Unfortunately, this approach is unsound. A more robust technique would be to integrate a string analysis [8, 19, 33] into our implementation.

Summary: Reflective coding constructs are the Achilles heel of static analysis. While this is true as it applies to applications written in .NET and Java, this is especially so given the declarative programming style often used in WP apps, where code is “glued together” with declarative specification. Several approaches to handling reflection have been proposed and used in the literature [6, 18, 26, 28, 35]. Alas, all of them require a certain degree of customization to the problem and APIs at hand. Additionally, reflection analysis tends to be intertwined

with a heavyweight analysis such as a points-to. We instead opt for a lightweight analysis that pattern-matches for the easily-to-resolve case, potentially introducing unsoundness. We evaluate the effects of this treatment in Section 6.

```
[SomaAd..ctor() @ 0134) bg      // resource access
[SomaAd..ctor() @ 0120) bg
[SomaAd..ctor() @ 0118) bg
[SomaAd..ctor() @ 0000) bg
[SomaAdViewer.StartAds() @ 00a6) bg
[SomaAdViewer.StartAds() @ 009e) bg
[SomaAdViewer.StartAds() @ 0000) bg
[CollectHome.g_AdFailed(object, ...) @ 00f7) fg
[CollectHome.g_AdFailed(object, ...) @ 0052) fg
[CollectHome.g_AdFailed(object, ...) @ 000a) fg
[CollectHome.g_AdFailed(object, ...) @ 0040) fg
[CollectHome.g_AdFailed(object, ...) @ 0030) fg
[CollectHome.g_AdFailed(object, ...) @ 0008) fg
[CollectHome.g_AdFailed(object, ...) @ 004a) fg
[CollectHome.g_AdFailed(object, ...) @ 00df) fg
[CollectHome.g_AdFailed(object, ...) @ 006c) fg
[CollectHome.g_AdFailed(object, ...) @ 0066) fg
```

Figure 13: A backward exploration tree of depth 20. Method names and signatures are abbreviated for brevity. `bg` and `fg` stands for background/library vs. foreground/non-library methods, respectively.

4.2 Fast Backward Placement

Recall from Section 3 that our approach resorts to a search for both checking if a resource access is already protected with a prompt and for inserting prompts if the dominator-based strategy fails. In implementing backward search, we need to be concerned with preventing infinite recursion (the occurs-check from Section 3). There is also the possibility of exponential path explosion, which is quite real given that we are dealing with graphs that typically have tens of thousands of nodes. It is therefore imperative to design an efficient exploration strategy.

Our approach for both checking for prompts and inserting them relies on first building a *spanning tree* rooted at the access node, computed using a depth-first search. Figure 13 gives an example of such a tree. The tree allows us to classify underlying graph edges as either forward, backward, or cross edges. Further analysis is performed on the tree as a series of downward passes, implemented as recursive procedures, starting at the resource access and exploring the predecessors³. In summary, we perform three recursive passes over the spanning tree. Each pass computes a boolean value for each of the visited nodes to represent the checking or placement

³To avoid stack overflow issues stemming from deep trees, once the tree has been constructed, we make sure that the size is below a fixed threshold (set to 250 for our experiments).

status; values are maintained across the passes in a map called v .

The advantage of this multi-pass approach is its simplicity and guaranteed runtime complexity. We start with all spanning tree nodes as unvisited and then perform three recursive traversals of the tree, as shown in Figure 14 and described below. In our implementation, we reuse the same spanning tree for the prompt checking and placement analysis stages. This approach is linear in the size of the graph, and is generally quite fast, even when there are hundreds of nodes reachable from a resource access.

1. **Traverse:** For each non-library non-background node, declare it as a valid placement point and set $v[n]$ to *true*⁴. For other nodes, if *all* their children have their v as *true*, set $v[n]$ to *true*; otherwise, set $v[n]$ to *false*.
2. **Patch-up:** Traverse the tree considering cross-edges originating at the current node. If all cross-edges emanating from nodes have valid placements (v value is *true*), set $v[n]$ to *true*.
3. **Collect:** Propagate (newly) *true* values up to the root: set $v[n]$ to *true* if the v value is *true* for all of n 's children.

The final result is computed by running all three steps in order and examining the result at the root of the spanning tree.

4.3 Runtime Considerations

While much of the focus of this paper is on statically locating placement points, choosing the right kind of runtime instrumentation presents some interesting challenges. We need to ensure that we are not going to induce double-prompting, as discussed in Section 3. To do so, we maintain a “sticky” app-global setting value in the app’s isolated storage, as illustrated by the following example for the fine-grained GPS location resource type:

```
1 var setting = IsolatedStorageSettings.
2     get_ApplicationSettings();
3     get_Item("UserLocationSettings");
4 if (setting == null){
5     int result = MessageBox.Show(
6         "Is it okay to access your fine-grained GPS location?",
7         "Allow "+Assembly.GetExecutingAssembly().FullName()+
8             " to access and use your location.",
9     1);
10    {
11        settings.set_Item("UserLocationSettings",
12            (result == 1) ? "Y" : "N");
13    }
14 }
```

⁴Note that to maximize backward placement opportunities, for all unreachable nodes, we set $v[n]$ to *true*, as shown in Figure 9. This is because the presence of dead code should not prevent prompt placement.

```

1: function TRAVERSE(n)
2:   for all c in children(n) do
3:     Traverse(c)
4:   end for
5:   v[n]  $\leftarrow$  true
6:   if n  $\notin$  G.Background  $\wedge$  n  $\notin$ 
    G.Libraries return
7:
8:   for all c in children(n) do
9:     if  $\neg v[c]$  then
10:      v[n]  $\leftarrow$  false
11:      return
12:    end if
13:   end for
14: end function
15:
16: function PATCHUP(n)
17:   if  $\neg v[n]$  then
18:     crossEdges  $\leftarrow$  CrossEdges(n)
19:     if |crossEdges| > 0 then
20:       v[n]  $\leftarrow$  true
21:       for all (n'  $\rightarrow$  n) in crossEdges do
22:         if  $\neg v[n']$  then
23:           v[n']  $\leftarrow$  false
24:           break
25:         end if
26:       end for
27:     end if
28:   end function
29:
30: function COLLECT(n)
31:   for all c in children(n) do
32:     if  $\neg v[c]$  then
33:       v[n]  $\leftarrow$  false
34:     return
35:   end if
36:   end for
37:   v[n]  $\leftarrow$  true
38: end function

```

Step 1: Traverse

```

1: function TRAVERSE(n)
2:   for all c in children(n) do
3:     Traverse(c)
4:   end for
5:   v[n]  $\leftarrow$  true
6:   if n  $\notin$  G.Background  $\wedge$  n  $\notin$ 
    G.Libraries return
7:
8:   for all c in children(n) do
9:     if  $\neg v[c]$  then
10:      v[n]  $\leftarrow$  false
11:      return
12:    end if
13:   end for
14: end function
15:
16: function PATCHUP(n)
17:   if  $\neg v[n]$  then
18:     crossEdges  $\leftarrow$  CrossEdges(n)
19:     if |crossEdges| > 0 then
20:       v[n]  $\leftarrow$  true
21:       for all (n'  $\rightarrow$  n) in crossEdges do
22:         if  $\neg v[n']$  then
23:           v[n']  $\leftarrow$  false
24:           break
25:         end if
26:       end for
27:     end if
28:   end function
29:
30: function COLLECT(n)
31:   for all c in children(n) do
32:     if  $\neg v[c]$  then
33:       v[n]  $\leftarrow$  false
34:     return
35:   end if
36:   end for
37:   v[n]  $\leftarrow$  true
38: end function

```

Step 2: Patch-up

```

1: function COLLECT(n)
2:   for all c in children(n) do
3:     if  $\neg v[c]$  then
4:       v[n]  $\leftarrow$  false
5:     return
6:   end if
7:   end for
8:
9:   v[n]  $\leftarrow$  true
10: end function

```

Step 3: Collect

Figure 14: Three-stage backward placement algorithm explained in Section 4.2.

```

13   }
14 }else{
15   if(setting.ToString().Equals("Y")){
16     // proceed with the prompt
17   }
18 }

```

Because the prompt remains sticky application-wide and persists across application invocations, even if we conservatively *insert* an extra prompt, we will only *show* it at most once per app.

5 Evaluation

We have analyzed 100 WP 7 apps from the WP Store to collect our results. To make the analysis more meaningful, we have selected only apps with `LOCATION` and `NETWORKING` capabilities. Such apps constitute about a fifth of a larger set of about 2,000, from which we drew our 100 app sample. The goal of our evaluation is to understand how frequently prompts are omitted and to attempt to insert prompts in a fully automatic manner.

Characterizing the input: We first present some aggregate statistics of the analysis results in Figure 15. WP applications are quite substantial in size, constituting about 3,528 methods on average. This is in part because they rely (and therefore recursively include within their call graph) large libraries, some of which are part of the operating system SDK, and others are included .NET libraries. The average size for our apps is 7.3 MB; many consist of dozens of DLLs.

We discovered that the libraries shown in the inlined figure are included most frequently. These libraries provide advertising functionality, and many request location data.

About 7% of all methods are contained in background tasks or libraries, which presents a significant challenge for prompt placement. Out of these, most are in

fact in third-party libraries. Recall that we do not want to place prompts in libraries. To recognize third-party libraries in our experiments, we used a list of 100 common advertising libraries, identified by the DLL in which they are contained; these include `Microsoft.Advertising.Mobile.dll`, `AdRotator.dll`, `MobFox.Ads.LocationAware.dll`, `FlurryWP7SDK.dll`, `Inneractive.Nokia.Ad.dll`, `MoAds.dll`, `adMob7.dll`, `Photobucket.Ads.dll` and many others. Our analysis is parameterized with respect to this list. Frameworks such as these may access GPS location deep within library code, making prompt placement analysis particularly difficult.

Our analysis represents each application as 13,330 nodes on average. Out of these, about 12% are considered to be anticipating by our analysis. In other words, about 88% of nodes are *not* eligible prompt placement points.

The last section of Figure 15 describes the resource accesses found in these 100 applications. Across all apps, there are 227 resource accesses we analyze. Overall, apps have an average of 2.27 resource accesses, with a maximum of 9 for one of the apps. The figure shown inline in this paragraph shows how frequent individual

Component	Count
SOMAWP7	42
NetDragon.PandaReader	13
EchoEchoBackgroundAgent	10
Utilities	10
BMSApp	10
MobFox.Ads.LocationAware	8
XIMAD_Ad_Client	7
EchoEcho	5
DirectRemote	5
DCMetroApp	5

Location	95.15%
Contacts	4.41%
Calendar	0.44%

apps analyzed	100
processed methods	352,816
background/library methods	26,033
library methods	25,898
nodes	1,333,056
anticipating	171,253
accesses	227
accesses in background/library methods	78

Figure 15: Apps analyzed: summary of input statistics.

succeeded	202
failed	19
succeeded unique	143
failed unique	7
dominator-based succeeded	150
naïve	143
backward succeeded	56
regular	150
dead code	2,094
backward placements	(40,270, 56)
depth exceeded	15

Figure 16: Prompt placement: summary of results of applying analysis to 100 apps.

resource types are. We find that the majority of sensitive resource accesses are to GPS location data, with occasional accesses to user contacts and calendar.

Inserting prompts: Figure 16 provides statistics describing the prompt placement process. Overall, our two-prong strategy of dominator-based and backward placement succeeds in about 91% of all cases. However, it is important to observe that many cases, including challenging resource accesses deep in library code, are shared by many applications. To avoid double-counting, we show the number of *unique* placement attempts that have succeeded and failed. Considering these numbers of unique accesses, we are able to successfully place prompts

	Average	Max	#
app loading	1,779	24,585	100
call graph construction	18,152	147,287	100
placement graph construction	15,103	293,480	100
anticipating computation	158	3826	86
finding missing prompts	123	649	100
prompt insertion, per app	942	70,228	103
dominator-based, per access	0.05	1	221
backward, per access	1,366	49,277	71

Figure 17: Timing, in *ms*. All measurements are per app, unless stated otherwise.

in 95% of cases (143 out of 150), a higher success percentage. Several other lessons can be drawn from the rest of the table:

- When dominator-based placement succeeds, it is usually immediate (95% of all dominator-based successes are *naïve* successes).
- Backward placement is helpful for cases where dominator-based placement fails. However, some of these cases are still too hard, leading to 7 unique failures.

Timing: Figure 17 provides a summary of timing information for our analysis. For each measurement, we provide the average timing across 100 apps, the maximum observed time and the number of observations. Each measurement is given in *ms*. Overall, the most time goes into initial processing of the application, which involves reading it from disk, constructing a representation of the app’s assemblies in memory (1.7 seconds on average), traversing it to create a call graph and control flow graphs (CFGs) (18 seconds on average), dominator calculation, and reachability calculation, resulting in a graph suitable for analysis. Computing anticipating nodes only takes 158 *ms* on average.

Finding missing prompts takes about 123 *ms* on average, in part because many instructions need to be examined in search of existing prompts. Prompt insertion, on average, is fast, only about .9 seconds per application. Dominance-based placement is virtually instantaneous. Backward placement is slower, at 1.3 seconds per resource access, raising the average. Based on these performance numbers, we are optimistic that prompt insertion can be done entirely automatically over a large number of applications.

6 Discussion

We have selected static analysis as a method of choice to avoid code coverage issues inherent with runtime analysis and for analysis speed (end-to-end processing is several minutes per app). In this section we discuss some of the limitations of our current static analysis approach. There are two potential sources of errors in our analysis. Our analysis may classify a resource access as *unprotected* whereas it is properly protected with runtime prompts; we call these cases *false positives*. By the same token, our analysis may classify a resource access as *protected* whereas in fact at runtime there are no preceding prompts that protect the resource access; we call these cases *false negatives*.

Manual inspection: We examined a subset of applications to manually check for these errors. The

verification process includes running these applications in the emulator to collect network packets and to collect API calls invoked by each application at runtime. We manually exercise as much functionality of each application as possible. If the application presented a runtime prompt, we inspected the text of the message and clicked through each “allow” (to use my location) and “don’t allow” button to determine how the choice affects application behavior.

Once the runtime inspection was complete, we examined network packets and invoked API lists, correlating them with the app’s disassembled code to verify the observed behavior. Although this verification process is thorough, it requires significant manual efforts, thus limiting the number of cases that can be examined. Next, we discuss findings from 10 applications. These apps contain 27 resource access points, among which 21 are classified as *unprotected* by our analysis.

6.1 False Negatives

Our manual analysis found no false negatives. On a close examination of each of the 27 resource accesses, we find 10 accesses that are not protected. Our analysis correctly identifies all of these accesses as unprotected and finds proper placements.

These unprotected accesses are found in third-party libraries included across 5 apps. Interestingly, in an effort to maximize revenue, one app embeds *two* advertising related third-party libraries (`SOMAWP7.dll` and `AdRotatorXNA.dll`) and *both* contain unprotected location accesses. Two placements are made via dominator-based placement; the other eight through backward placement. Backward placements result in 40 inserted prompts in application code, which upon casual examination appear to be correct. We find these results promising, as users express increasing concern about data sharing with third parties [21], and our analysis properly detects and fixes such unprotected accesses.

6.2 False Positives

Eleven out of 21 accesses flagged as unprotected turn out to be properly protected. Although the number of false positives is somewhat high, with manual inspection, we found the following reasons for them:

Sticky location prompt: Seven false positives are due to our analysis’s inability to analyze sticky location prompts, as shown in Figure 5. Three cases are similar to the example in Figure 18(a). The rest are caused by one application that uses the location flag to enable or disable the button that allows the

```

private void mapLocaitons() {
    if (this.avisAppUnitService.UseLocationsMapping) {
        this.watcher=null;
        GeoCoordinateWatcher watcher=new GeoCoordinateWatcher
            (GeoPositionAccuracy.Default){MovementThreshold = 20.0};
        this.watcher = watcher;
        ...
        this.watcher.Start();
    }
}

(a) Sticky prompt example #1: This app
saves the result of the prompt response in
Athis.avisAppUnitService.UseLocationsMapping.

public MapPage() {
    this.InitializeComponent();
    base.DataContext = new MapViewModel();
    this.BuildApplicationBar();
    if (AppSettings.Current.UseLocationService) {
        this.watcher = new GeoCoordinateWatcher();
    }
    ((ApplicationBarIconButton)base.ApplicationBar.Buttons[0]).IsEnabled = AppSettings.Current.UseLocationService;
    ((ApplicationBarIconButton)base.ApplicationBar.Buttons[2]).IsEnabled = AppSettings.Current.UseLocationService;
    this.UpdatePushpinsBackground();
}

(b) Sticky prompt example #2: This app disables page
navigation based on the location access depending on
AppSettings.Current.UseLocationService.

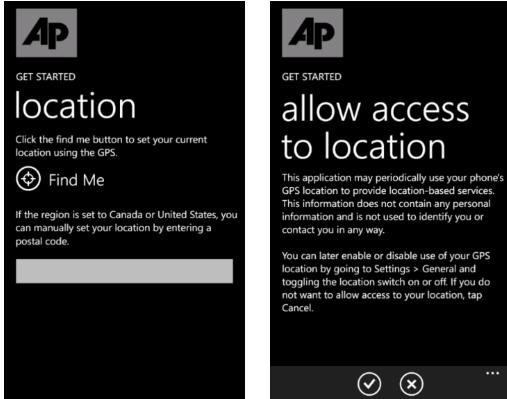
```

Figure 18: Sticky prompt examples.

user to navigate to the page (that invokes location access) as shown in Figure 18(b). WP apps can use several different storage mechanisms; we are looking into ways to detect them statically.

Consent dialog implementation: Two false positives are due to the limitation of identifying existing prompts. Both result from a single app that implements a custom consent dialog page instead of `MessageBox()`, as shown in Figure 19. We are looking into ways to parse a blocking page with buttons to detect such custom-made consent dialog pages, although this is obviously a difficult problem. However, such cases are not common and we find that five out of six applications that show prompts employ `MessageBox()`, as expected.

Async calls and XAML files: Two false positives are due to limitations of call graph construction. Figure 20(a) shows an expanded example of the case discussed in Section 4.1. Applications may use multiple types of `EventHandlers` to be called asynchronously. In our current implementation, we parse `EventHandlers` and add links when handlers are registered. However, the current implementation fails when multiple delegates and `EventHandlers` are used in a tricky way, as shown in Figure 20(b). We are investigating ways to extend our call graph construction to support these cases.



(a) App page with location access. (b) Prompt (consent dialog).

Figure 19: False positive due to a custom prompt: A prompt is customized as a separate WP UI page.

6.3 Effect of False Positives

Like most practical static analysis tools, our analysis is potentially vulnerable to false positives, primarily because of program representation challenges. Unlike most static analysis tools for bug detection, our analysis is two-phase: if it detects that a resource access is not adequately protected, it tries to propose a placement of prompts that would protect it. Our analysis errs on the safe side, introducing false positives and not false negatives.

False positives, however, may lead to double-prompts, since our analysis will inject a prompt to protect already protected resource accesses. Because our inserted prompts are sticky, our approach introduces at most one extra runtime prompt per app during the entire app’s lifecycle, which we believe will not lead to prompt fatigue. Nonetheless, double-prompts can trigger confusion in end-users and therefore should be minimized. Our experience with the ten test applications shows that in all cases, resource accesses get triggered quickly, with several clicks, so runtime checking of this kind is unlikely to require excessive effort. If desired, runtime testing by the developer or App Store maintainers can accompany our analysis to detect and eliminate potential double-prompts.

7 Related Work

The requirement of protecting privacy-sensitive resource accesses with runtime prompts or consent dialogs has only recently been introduced to mobile applications. To our knowledge, no previous work has investigated static analysis approaches to detect

```
private void GPS_MouseLeftButtonDown(object sender,
    MouseEventArgs e) {
    ...
    else if (MessageBox.Show("Sharing this info allows us to
        find theaters and events near you. We won't share
        this information.", "Allow BookMyShow to access and
        use your location.", MessageBoxButton.OKCancel)==
    MessageBoxResult.OK) {
        ...
        base.NavigationService.Navigate(
            new Uri("/VenueByGeo.xaml?mc=" +
            this.strMenuCode + "&t=" + this.strToken,
            UriKind.RelativeOrAbsolute));
    }
    ...
}
```

(a) Complex CFG #1: Function `Navigate()` internally calls `BMSApp.VenueByGeo.OnNavigatedTo()` as defined in `VenueByGeo.xaml`.

```
public static bool GetCurrentLocation() {
    ...
    Observable.FromEvent
        <GeoPositionStatusChangedEventArgs>
        (delegate
            (EventHandler<GeoPositionStatusChangedEventArgs> ev){
                GeoCoordinateWatcher.StatusChanged += ev;
            }, delegate (EventHandler<GeoPositionStatusChangedEventArgs> ev) {
                GeoCoordinateWatcher.StatusChanged -= ev;
            }).Where<IEvent
                <GeoPositionStatusChangedEventArgs>>
                (delegate ... args){
                    if (args.EventArgs.Status != GeoPositionStatus.Ready)
                    {
                        return (args.EventArgs.Status ==
                            GeoPositionStatus.Disabled);
                    }
                    return true;
                }).Take<IEvent<GeoPositionStatusChangedEventArgs>>(1).
                    Subscribe<IEvent<GeoPositionStatusChangedEventArgs>>
                    (delegate
                        (IEvent<GeoPositionStatusChangedEventArgs> args) {
                            if (args.EventArgs.Status == GeoPositionStatus.Ready)
                            {
                                RaiseCurrentLocationAvailable(
                                    new CurrentLocationAvailableEventArgs(
                                        GeoCoordinateWatcher.Position.Location));
                            }
                        }
                    );
    ...
}
```

(b) Complex CFG #2: This code generates a compiler-generated function `<GetCurrentLocation>b_3` in `Eventful.Helpers.LocationHelper`, which is called within `GetCurrentLocation()`, as defined in `VenueByGeo.xaml`.

Figure 20: Complex CFG cases.

unprotected resource accesses in mobile application binaries. This section discusses previous research in three related areas: automatic hook placement, graph-based analysis for information security, and user studies of consent dialogs.

Automatic hook placement: A number of previous studies examine the issues of protecting security-sensitive operations with authorization hooks (e.g., checking permissions for file operations). Ganapathy *et al.* [14] use a static program analysis over the Linux kernel source code to identify previously unspecified sensitive operations and find the right set

of hooks that need to protect them. AutoISES by Tan *et al.* [34] is designed for the similar goal as [14] but the ways that AutoISES infers access to sensitive data structure are different from [14]. Muthukumaran *et al.* [27] focus on server code such as the X server and postgresql and use their insight concerning object access patterns in order to identify sensitive operations that require authorization.

In comparison to these efforts, our work begins with a set of known APIs that access sensitive resources. Such a set is easy to mine from developer documentation for most mobile operating systems. In particular, our work focuses on algorithms to find placements that meet the four important conditions specific to user prompts on mobile devices, whereas the previous work concentrates of placement being safe [14, 34] or safe and not-repetitive [27].

Graph-based analysis: Program dependence graphs are used for analyzing information security of programs in several projects [16, 17, 32]. Program dependence graphs include both data dependencies and control dependencies whereas the dataflow graphs that we use in this work typically contain just data dependencies. Hammer *et al.* [15] consider the enforcement of declassification [30] using program dependence graphs. Recent efforts focus on automating security-critical decisions for application developers [31, 36]. The use of a security type system for enforcing correctness is another case of cooperating with the developer to achieve better code quality and correctness guarantees [29]. Livshits and Chong [25] address the problem of sanitizer placement through static analysis and partially inspire our work on consent dialog placement. In our work, we use a backwards traversal to find the closest valid node to insert a missing prompt. Au *et al.* [5] use a similar backward reachability analysis over a call graph constructed from the Android framework. However, their goal is to create a mapping between API calls and permission checks and therefore their analysis need not consider the four conditions.

Mobile user privacy and consent dialogs: Several recent studies have investigated the effectiveness of existing consent dialogs used on mobile devices at informing users about which privacy-sensitive data can be accessed by apps. Felt *et al.* [12] show that only 17% of study participants paid attention to the permissions when installing Android applications. This finding may indicate that placing consent dialogs at install time (far removed from when the data is actually being accessed) renders these dialogs ineffective. On the contrary, a study by Fisher *et al.* focus on iPhone users' responses to *runtime* consent dialogs to location access and shows that 85% of

study participants actually denied location requests for at least one app on their phone [13].

Although orthogonal to our work, previous studies have explored ways to improve the presentation of consent dialogs in mobile devices. Lin *et al.* measure users' "expectations" of apps' access to phone resources [23]. By highlighting unexpected behaviors in the Android permissions interface, the authors show that the new permission interface is more easily understood and efficient than the existing one. Felt *et al.* propose a framework for requesting permissions on smartphones [11]. Findings of these studies can inform a better usable privacy design of a consent dialog, which our analysis can automatically insert in mobile apps.

8 Conclusions

In this paper, we have explored the problem of missing prompts that should guard sensitive resource accesses. Our core contribution is a graph-theoretic algorithm for placing such prompts automatically. The approach balances the execution speed and few prompts inserted via dominator-based placement with a comprehensive nature of a more exhaustive backward analysis.

Overall, our two-prong strategy of dominator-based and backward placement succeeds in about 95% of all unique cases. Our approach is highly scalable; once the application has been represented in the form of a graph, analysis usually takes under a second on average.

References

- [1] Pandora discloses privacy-related US inquiry into phone apps. <http://www.nytimes.com/2011/04/05/technology/05pandora.html>, April 2011.
- [2] Daily report: Social app makes off with address books. <http://bits.blogs.nytimes.com/2012/02/08/daily-report-social-app-makes-off-with-address-books/>, February 2012.
- [3] LinkedIn's iOS app collects and transmits names, emails and notes from your calendar, in plain text. <http://thenextweb.com/insider/2012/06/06/linkedin-ios-app-collects-and-sends-names-emails-and-meeting-notes-from-your-calendar-back-in-plain-text/>, June 2012.
- [4] A. V. Aho, M. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 2007.
- [5] K. W. Y. Au, Y. F. Zhou, Z. Huang, and D. Lie. Pscout: analyzing the android permission specification. In *ACM CCS*, 2012.
- [6] E. Bodden, A. Sewe, J. Sinschek, H. Oueslati, and M. Mezini. Taming reflection: Aiding static analysis in the presence of reflection and custom class loaders. In *Software Engineering (ICSE), 2011 33rd International Conference on*, pages 241–250, 2011.

- [7] R. Böhme and S. Köpsell. Trained to accept?: a field experiment on consent dialogs. In *Proceedings of CHI*, 2010.
- [8] A. S. Christensen, A. Møller, and M. Schwartzbach. Precise analysis of string expressions. In *International Conference on Static analysis*, 2003.
- [9] M. Egele, C. Kruegel, E. Kirda, and G. Vigna. PiOS: Detecting privacy leaks in iOS applications. In *Proceedings of the Annual Network and Distributed System Security Symposium*, Feb. 2011.
- [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*, 2010.
- [11] A. P. Felt, S. Egelman, M. Finifter, D. Akhawe, and D. Wagner. How to ask for permission. In *Proceedings of HotSec*, 2012.
- [12] A. P. Felt, E. Hay, S. Egelman, A. Haney, E. Chin, and D. Wagner. Android permissions: User attention, comprehension, and behavior. In *Proceedings of SOUPS*, 2012.
- [13] D. Fisher, L. Dorner, and D. Wagner. Short paper: location privacy: user behavior in the field. In *Proceedings of SPSM*, 2012.
- [14] V. Ganapathy, T. Jaeger, and S. Jha. Automatic placement of authorization hooks in the linux security modules framework. In *ACM CCS*, 2005.
- [15] C. Hammer, J. Krinke, and F. Nodes. Intransitive noninterference in dependence graphs. In *2nd International Symposium on Leveraging Application of Formal Methods, Verification and Validation*, Nov. 2006.
- [16] C. Hammer, J. Krinke, and G. Snelting. Information flow control for java based on path conditions in dependence graphs. In *IEEE International Symposium on Secure Software Engineering*, Mar. 2006.
- [17] C. Hammer and G. Snelting. Flow-sensitive, context-sensitive, and object-sensitive information flow control based on program dependence graphs. *International Journal of Information Security*, 8(6):399–422, Dec. 2009.
- [18] M. Hirzel, D. von Dincklage, A. Diwan, and M. Hind. Fast online pointer analysis. *ACM Trans. Program. Lang. Syst.*, 29(2), 2007.
- [19] P. Hooimeijer and M. Veanes. An evaluation of automata algorithms for string analysis. In *Verification, Model Checking, and Abstract Interpretation*, pages 248–262. Springer, 2011.
- [20] P. Internet. Privacy and data management on mobile devices. <http://pewinternet.org/Reports/2012/Mobile-Privacy.aspx>, September 2012.
- [21] J. Jung, S. Han, and D. Wetherall. Short paper: Enhancing mobile application permissions with runtime feedback and constraints. In *Proceedings of SPSM*, 2012.
- [22] S. Kaplan, B. Livshits, B. Zorn, C. Seifert, and C. Curtsinger. “nofus: Automatically detecting” + string.fromcharcode(32) + “obfuscated”.tolowercase() + “javascript code”. Technical Report MSR-TR-2011-57, Microsoft Research, May 2011.
- [23] J. Lin, S. Amini, J. Hong, N. Sadeh, J. Lindqvist, and J. Zhang. Expectation and purpose: Understanding users’ mental models of mobile app privacy through crowdsourcing. In *Proceedings of UbiComp 2012*, 2012.
- [24] B. Livshits. Dynamic taint tracking in managed runtimes. Technical Report MSR-TR-2012-114, Microsoft Research, 2012.
- [25] B. Livshits and S. Chong. Towards fully automatic placement of security sanitizers and classifiers. In *Proceedings of the Sympolism on Principles of Programming Languages (POPL)*, Jan. 2013.
- [26] B. Livshits, J. Whaley, and M. S. Lam. Reflection analysis for Java. In *Asian Symposium on Programming Languages and Systems*, Nov. 2005.
- [27] D. Muthukumaran, T. Jaeger, and V. Ganapathy. Leveraging “choice” to automate authorization hook placement. In *ACM CCS*, 2012.
- [28] G. Richards, S. Lebresne, B. Burg, and J. Vitek. An analysis of the dynamic behavior of javascript programs. In *ACM Sigplan Notices*, volume 45, pages 1–12. ACM, 2010.
- [29] W. Robertson and G. Vigna. Static enforcement of Web application integrity through strong typing. In *Proceedings of the Usenix Security Symposium*, Aug. 2009.
- [30] A. Sabelfeld and D. Sands. Dimensions and principles of declassification. In *Proceedings of the 18th IEEE Computer Security Foundations Workshop*, pages 255–269. IEEE Computer Society, June 2005.
- [31] M. Samuel, P. Saxena, and D. Song. Context-sensitive auto-sanitization in web templating languages using type qualifiers. In *Proceedings of the Conference on Computer and Communications Security*, Oct. 2011.
- [32] B. Scholz, C. Zhang, and C. Cifuentes. User-input dependence analysis via graph reachability. Technical Report 2008-171, Sun Microsystems Labs, 2008.
- [33] D. Shannon, S. Hajra, A. Lee, D. Zhan, and S. Khurshid. Abstracting symbolic execution with string analysis. In *Testing: Academic and Industrial Conference Practice and Research Techniques-MUTATION*, 2007. TAICPART-MUTATION 2007, pages 13–22, 2007.
- [34] L. Tan, X. Zhang, X. Ma, W. Xiong, and Y. Zhou. Autoises: Automatically inferring security specification and detecting violations. In *USENIX Security Symposium*, 2008.
- [35] O. Tripp, M. Pistoia, S. J. Fink, M. Sridharan, and O. Weisman. TAJ: effective taint analysis of Web applications. In *Proceedings of the Conference on Programming Language Design and Implementation*, 2009.
- [36] J. Weinberger, P. Saxena, D. Akhawe, R. Shin, and D. Song. A systematic analysis of XSS sanitization in Web application frameworks. In *Proceedings of the European Symposium on Research in Computer Security*, Sept. 2011.

Flexible and Fine-Grained Mandatory Access Control on Android for Diverse Security and Privacy Policies

Sven Bugiel*

bugiel@cs.uni-saarland.de

Saarland University, Germany

Stephan Heuser

stephan.heuser@sit.fraunhofer.de

Fraunhofer SIT, Germany

Ahmad-Reza Sadeghi

ahmad.sadeghi@trust.cased.de

Technische Universität Darmstadt / CASED, Germany

Abstract

In this paper we tackle the challenge of providing a generic security architecture for the Android OS that can serve as a flexible and effective ecosystem to instantiate different security solutions. In contrast to prior work our security architecture, termed *FlaskDroid*, provides mandatory access control simultaneously on both Android’s middleware and kernel layers. The alignment of policy enforcement on these two layers is non-trivial due to their completely different semantics. We present an efficient policy language (inspired by SELinux) tailored to the specifics of Android’s middleware semantics. We show the flexibility of our architecture by policy-driven instantiations of selected security models such as the existing work *Saint* as well as a new privacy-protecting, user-defined and fine-grained per-app access control model. Other possible instantiations include *phone booth mode*, or *dual persona* phone. Finally we evaluate our implementation on SE Android 4.0.4 illustrating its efficiency and effectiveness.

1 Introduction

Mobile devices such as smartphones and tablets have become very convenient companions in our daily lives and, not surprisingly, also appealing to be used for working purposes. On the down side, the increased complexity of these devices as well as the increasing amount of sensitive information (private or corporate) stored and processed on them, from user’s location data to credentials for online banking and enterprise VPN, raise many security and privacy concerns. Today the most popular and widespread smartphone operating system is Google’s Android [4].

Android’s vulnerabilities. Android has been shown to be vulnerable to a number of different attacks such as malicious apps and libraries that misuse their privileges [57, 40, 25] or even utilize root-exploits [55, 40] to extract security and privacy sensitive information; taking advantage of unprotected interfaces [14, 12, 53, 32] and files [49]; confused deputy attacks [16]; and collusion attacks [46, 34].

Solutions. On the other hand, Android’s open-source nature has made it very appealing to academic and industrial security research. Various extensions to Android’s access control framework have been proposed to address particular problem sets such as protection of the users’ privacy [19, 28, 15, 52, 7, 30]; application centric security such as *Saint* enabling developers to protect their application interfaces [39]; establishing isolated domains (usage of the phone in private and corporate context) [9]; mitigation of collusion attacks [8], and extending Android’s Linux kernel with Mandatory Access Control [48].

Observations. Analyzing the large body of literature on Android security and privacy one can make the following observations: First, almost all proposals for security extensions to Android constitute mandatory access control (MAC) mechanisms that are tailored to the specific semantics of the addressed problem, for instance, establishing a fine-grained access control to user’s private data or protecting the platform integrity. Moreover, these solutions fall short with regards to an important aspect, namely, that protection mechanisms operate only at a specific system abstraction layer, i.e., either at the middleware (and/or application) layer, or at the kernel-layer. Thus, they omit the peculiarity of the Android OS design that each of its two software layers (middleware and kernel) is important within its respective semantics for the desired overall security and privacy.

* Author was affiliated with Technische Universität Darmstadt/CASED at the time this work was conducted.

Only few solutions consider both layers [8, 9], but they support only a very static policy and lack the required flexibility to instantiate different security and privacy models.

The second observation concerns the distinguishing characteristic of application development for mobile platforms such as Android: The underlying operating systems provide app developers with clearly defined programming interfaces (APIs) to system resources and functionality – from network access over personal data like SMS/contacts to the onboard sensors. This clear API-oriented system design and convergence of functionality into designated service providers [54, 36] is well-suited for realizing a security architecture that enables fine-grained access control to the resources exposed by the API. As such, mobile systems in general and Android in particular provide better opportunities to more efficiently establish a higher security standard than possible on current commodity PC platforms [31].

Challenges and Our Goal. Based on the observations mentioned above, we aim to address the following challenges in this paper: 1) Can we design a generic and practical mandatory access control architecture for Android-based mobile devices, that operates on both kernel and middleware layer, and is flexible enough to instantiate various security and privacy protecting models just by configuring security policies? More concretely, we want to create a generic security architecture which supports the instantiation of already existing proposals such as *Saint* [39] or privacy-enhanced system components [58], or even new use-cases such as a *phone booth mode*. 2) To what extent would the API-oriented design of Android allow us to minimize the complexity of the desired policy? Note that policy complexity is an often criticized drawback of generic MAC solutions like SELinux [33] on desktop systems [54].

Our Contribution. In this paper, we present the design and implementation of a security architecture for the Android OS that addresses the above mentioned challenges. Our design is inspired by the concepts of the *Flask* architecture [50]: a modular design that decouples policy enforcement from the security policy itself, and thus provides a generic architecture where multiple and dynamic security policies can be supported by the system. In particular, our contributions are:

1. *System-wide security framework.* We present an Android security framework that operates on both the middleware and kernel layer. It addresses many

problems of the stock Android permission framework and of related solutions which target either the middleware or the kernel layer. We base our implementation on SE Android [48], which has already been partially merged into the official Android source-code by Google¹.

2. *Security policy and type enforcement at middleware layer.* We extended Android’s middleware layer with type enforcement and present our policy language, which is specifically designed for the rich semantics at this layer. The alignment of middleware and kernel layer policies in a system-wide security framework is non-trivial, particularly due to the different semantics of both layers.

3. *Use-cases.* We show how our security framework can instantiate selected use-cases. The first one is an attack-specific related work, the well-known application centric security solution *Saint* [39]. The second one is a privacy protecting solution that uses fine-grained and user-defined access control to personal data. We also mention other useful security models that can be instantiated with FlaskDroid.

4. *Efficiency and effectiveness.* We successfully evaluate the efficiency and effectiveness of our solution by testing it against a testbed of known attacks and by deriving a basic system policy which allows for the instantiation of further use-cases.

2 Background

In this section, we first present a short overview of the standard Android software stack, focusing on the relevant security and access control mechanisms in place. Afterwards, we elaborate on the SE Android Mandatory Access Control (MAC) implementation.

2.1 Android Software Stack

Android is an open-source software stack tailored to mobile devices, such as smartphones and tablets. It is based on a modified Linux kernel responsible for basic operating system services (e.g. memory management, file system support and network access).

Furthermore, Android consists of an application framework implementing (most of) the Android API. System Services and libraries, such as the radio interface layer, are implemented in C/C++. Higher-level services, such as *System settings*, the *Location-* and *Audiomanager*, are implemented in Java. Together, these components comprise the middleware layer.

¹http://www.osnews.com/story/26477/Android_4_2_alpha_contains_SELinux

Android applications (apps) are implemented in Java and may contain native code. They are positioned at the top of the software stack (application layer) and use kernel and middleware Services. Android ships with standard apps completing the implementation of the Android API, such as a **Contacts (database) Provider**. The user can install additional apps from, for example, the Google Play store.

Android apps consist of certain components: Activities (user interfaces), Services (non user-interactive tasks), ContentProviders (SQL-like databases), and Broadcast Receivers (mailboxes for broadcast messages). Apps can communicate with each other on multiple layers: 1) Standard Linux Inter-Process Communication (IPC) using, e.g., domain sockets; 2) Internet sockets; 3) *Inter-Component Communication* (ICC) [21], a term abstractly describing a lightweight IPC mechanism between app components, called *Binder*. Furthermore, predefined actions (e.g., starting an Activity) can be triggered using an Intent, a unicast or broadcast message sent by an application and delivered using the Android ICC mechanism.

2.2 Security Mechanisms

Sandboxing. Android uses the Linux discretionary access control (DAC) mechanism for application sandboxing by assigning each app a unique user identifier (UID) during installation². Every process belonging to the app is executed in the context of this UID, which determines access to low level resources (e.g. app-private files). Low-level IPC (e.g. using domain sockets) is also controlled using Linux DAC.

Permissions. Access control is applied to ICC using *Permissions* [21]: Labels assigned to apps at install-time after being presented to and accepted by the user. These labels are checked by reference monitors at middleware- and application level when security-critical APIs are accessed. In addition to Android’s default permissions, app developers can define their own permissions to protect their applications’ interfaces. However, it should be noted that the permission model is *not* mandatory access control (MAC), since callees must discretely deploy or define the required permission check and, moreover, permissions can be freely delegated (e.g., URI permissions).

Permissions are also used to restrict access to some low level resources, such as the world read-/writeable external storage area (e.g. a MicroSD card) or network access. These permissions are mapped to Linux group identifiers (GIDs) assigned to an app’s UID

²Developers may use the same UID (Shared UID, SUID) for their own apps. These apps will share the same sandbox.

during installation and checked by reference monitors in the Linux kernel at runtime.

2.3 SELinux

Security Enhanced Linux (SELinux) [33] is an instantiation of the Flask security architecture [50] and implements a policy-driven mandatory access control (MAC) framework for the Linux kernel. In SELinux, policy decision making is decoupled from the policy enforcement logic. Various access control enforcement points for low-level resources, such as files, IPC, or memory protection enforce policy decisions requested from a *security server* in the kernel. This security server manages the policy rules and contains the access decision logic. To maintain the security server (e.g., reload the policy), SELinux provides a number of userspace tools.

Access Control Model. SELinux supports different access control models such as *Role-Based Access Control* and *Multilevel Security*. However, *Type Enforcement* is the primary mechanism: each object (e.g., files, IPC) and subject (i.e., processes) is labeled with a *security context* containing a *type* attribute that determines the access rights of the object/subject. By default, all access is denied and must be explicitly granted through policy rules—*allow rules* in SELinux terminology. Using the notation introduced in [26], each rule is of the form

$$\textit{allow } T_{\textit{Sub}} \; T_{\textit{Obj}} : C_{\textit{Obj}} \; O_C$$

where $T_{\textit{Sub}}$ is a set of subject types, $T_{\textit{Obj}}$ is a set of object types, $C_{\textit{Obj}}$ is a set of object classes, and O_C is a set of operations. The object classes determine which kind of objects this rule relates to and the operations contain specific functions supported by the object classes. If a subject whose type is in $T_{\textit{Sub}}$ wants to perform an operation that is in O_C on an object whose class is in $C_{\textit{Obj}}$ and whose type is in $T_{\textit{Obj}}$, this action is allowed. Otherwise, if no such rule exists, access is denied.

Dynamic policies. SELinux supports some extent dynamic policies based on *boolean flags* which affect *conditional policy* decisions at runtime. These booleans and conditions have to be defined prior to policy deployment and new booleans/conditions can *not* be added after the policy has been loaded without recompiling and reloading the entire policy. The simplest example for such dynamic policies are booleans to switch between “enforcing mode” (i.e., access denials are enforced) and “permissive mode” (i.e., access denials are not enforced).

Userspace Object Managers. A powerful feature of SELinux is that its access control architecture can

be extended to security-relevant userspace daemons and services, which manage data (objects) independently from the kernel. Thus, such daemons and services are referred to as **Userspace Object Managers (USOMs)**. They are responsible for assigning security contexts to the objects they manage, querying the SELinux security server for access control decisions, and enforcing these decisions. A prominent example for such USOMs on Linux systems is *GConf* [13].

2.4 SE Android

SE Android [48] prototypes SELinux for Android’s Linux kernel and aims to demonstrate the value of SELinux in defending against various root exploits and application vulnerabilities. Specifically, it confines system Services and apps in different kernelspace security domains even isolating apps from one another by the use of the Multi-Level Security (MLS) feature of SELinux. To this end, the SE Android developers started writing an Android-specific policy from scratch. In addition, SE Android provides a few key security extensions tailored for the Android OS. For instance, it labels application processes with SELinux-specific security contexts which are later used in type enforcement. Moreover, since (in the majority of cases) it is *a priori* unknown during policy writing which apps will be installed on the system later, SE Android employs a mechanism to derive the security context of an app at install-time. Based on criteria, such as the requested permissions, apps are assigned a security type. This mapping from application meta-information to security types is defined in the SE Android policy.

Additionally, SE Android provides *limited* support for MAC policy enforcement at the Android middleware layer (MMAC) and we explain these particular features in Section 7.2 and provide a comparison to our FlaskDroid architecture.

3 Requirements Analysis for Android Security Architectures

3.1 Adversary Model

We consider a strong adversary with the goal to get access to sensitive data as well as to compromise system or third-party apps. Thus, we consider an adversary that is able to launch *software* attacks on different layers of the Android software stack.

3.1.1 Middleware Layer

Recently, different attacks operating at Android’s middleware layer have been reported:

Overprivileged 3rd party apps and libraries threatening user privacy by adopting questionable privacy practices (e.g. WhatsApp [6] or Path [23]). Moreover, advertisement libraries, frequently included in 3rd party apps have been shown to exploit the permissions of their host app to collect information about the user [25].

Malicious 3rd party apps [22] leverage dangerous permissions to cause financial harm to the user (e.g., sending premium SMS) and exfiltrate user-private information [57, 40].

Confused deputy attacks concern malicious apps, which leverage unprotected interfaces of benign system [20, 41] and 3rd party [16, 56] apps (denoted deputies) to escalate their privileges.

Collusion attacks concern malicious apps that collude using covert or overt channels [8, 34] in order to gain a permission set which has not been approved by the user (e.g. the Soundcomber attack [46]).

Sensory malware leverages the information from onboard sensors, like accelerometer data, to derive privacy sensitive information, like user input [53, 12].

3.1.2 Root Exploits

Besides attacks at Android’s middleware layer, various privilege escalation attacks on lower layers of the Android software stack have been reported [55, 40] which grant the attacker root (i.e., administrative) privileges and can be used to bypass the Android permission framework. For instance, he can bypass the `ContactsProvider` permission checks by accessing the contacts database file directly. Moreover, processes on Android executing with root privileges inherit all available permissions at middleware layer.

It should be noted that attacks targeting vulnerabilities of the Linux kernel are out of scope of this paper, since SE Android is a building block in our architecture (see Section 4) and as part of the kernel it is susceptible to kernel exploits.

3.2 Requirements

Based on our adversary model we derive the necessary requirements for an efficient and flexible access control architecture for mobile devices, focusing on the Android OS.

Access Control on Multiple Layers. Mandatory access control solutions at kernel level, such as SE Android [48] or Tomoyo [27], help to defend against or to constrain privilege escalation attacks on

the lower-levels of the OS [48]. However, kernel level MAC provides insufficient protection against security flaws in the middleware and application layers, and lacks the necessary high-level semantics to enable a fine-grained filtering at those layers [48, 47]. Access control solutions at middleware level [28, 15, 39, 9, 8] are able to address these shortcomings of kernel level MAC, but are, on the other hand, susceptible to low-level privilege escalation attacks.

Thus, a first requirement is to provide simultaneous MAC defenses at the two layers. Ideally, these two layers can be dynamically synchronized at run-time over mutual interfaces. At least, the kernel MAC is able to preserve security *invariants*, i.e., it enforces that any access to sensitive resources/functionality is always first mediated by the middleware MAC.

Multiple stakeholders policies. Mobile systems involve multiple stakeholders, such as the end-user, the device manufacturer, app developers, or other 3rd parties (e.g., the end-user’s employer). These stakeholders also store sensitive data on the device. Related work [39, 9] has proposed special purpose solutions to address the security requirements and specific problems of these parties. Naturally, the assets of different stakeholders are subject to different security requirements, which are not always aligned and might conflict. Thus, one objective for a generic MAC framework that requires handling policies of multiple stakeholders is to support (basic) policy reconciliation mechanisms [43, 35].

Context-awareness. The security requirements of different stakeholders may depend on the current context of the device. Thus, our architecture shall provide support for context-aware security policies.

Support for different Use-Cases. Our architecture shall serve as a basis for different security solutions applicable in a variety of use cases. For instance, by modifying the underlying policy our solution should be able to support different use cases (as shown in Section 5), such as the selective and fine-grained protection of app interfaces [39] or privacy-enhanced system Services and ContentProviders.

4 FlaskDroid Architecture

In this section, we provide an overview of our FlaskDroid architecture, elaborate in more detail on particular design decisions, and present the policy language employed in our system. Due to space constraints, we focus on the most important aspects and refer to our technical report [11] for more detailed information.

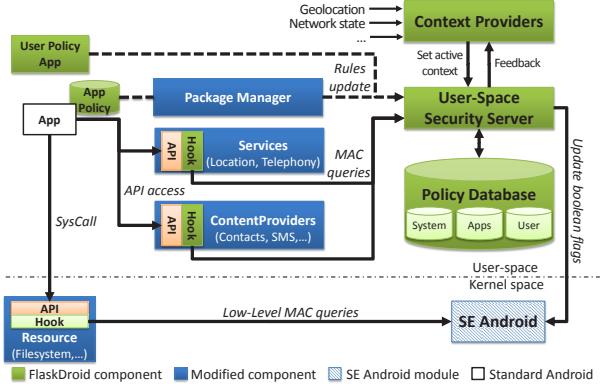


Figure 1: FlaskDroid Architecture

4.1 Overview

The high-level idea of FlaskDroid is inspired by the Flask security architecture [50], where various Object Managers at middleware and kernel-level are responsible for assigning security contexts to their objects. Objects can be, for instance, kernel resources such as Files or IPC and middleware resources such as Service interfaces, Intents, or ContentProvider data. On access to these objects by subjects (i.e., apps) to perform a particular operation, the managers enforce an access control decision that they request from a security server at their respective layer. Thus, our approach implements a *userspace security server*. Access control in FlaskDroid is implemented, as in SE Android (cf. Section 2), as *type enforcement*. However, in contrast to SE Android we extend our policy language with new features that are tailored to the Android middleware semantics (cf. Section 4.3). Moreover, to enable more dynamic policies, the policy checks in FlaskDroid depend also on the *System State*, which determines the actual security context of the objects and subjects at runtime.

Each security server is also responsible for the policy management for multiple stakeholders such as app developers, end-user, or 3rd parties. A particular feature is that the policies on the two layers are synchronized at runtime, e.g., a change in enforcement in the middleware, must be supported/reflected at kernel-level. Thus, by decoupling the policy management and decision making from the enforcement points and consolidating the both layers, the goal of FlaskDroid’s design is to provide fine-grained and highly flexible access control over operations on both middleware and kernel-level.

4.2 Architecture Components

Figure 1 provides an overview of our architecture. In the following, we will explain the individual components that comprise the FlaskDroid architecture.

4.2.1 SE Android Module

At the kernel-level, we employ stock SE Android [48] as a building block primarily for the following purposes: First, it is essential for hardening the Linux kernel [48] thereby preventing malicious apps from (easily) escalating their privileges by exploiting vulnerabilities in privileged (system) services. Even when an attack, usually with the intent of gaining *root* user privileges, is successful, SE Android can constrain the file-system privileges of the app by restricting the privileges of the root account itself. Second, it complements the policy enforcement at the middleware level by preventing apps from bypassing the middleware enforcement points (in Flask terminology defined as **Userspace Object Managers (USOMs)**), for example, accessing the contacts database file directly instead of going through the `ContactsProvider` app. **Dynamic policies.** Using the dynamic policy support of SELinux (cf. Section 2.3) it is possible to reconfigure the access control rules at runtime depending on the current system state. Our **Userspace Security Server** (cf. Section 4.2.2) is hereby the trusted user space agent that controls the SELinux dynamic policies and can map system states and contexts to SELinux boolean variables (cf. Section 4.3). To this end, SE Android provides user space support (in particular `android.os.SELinux`).

4.2.2 Userspace Security Server

In our architecture, the **Userspace Security Server** is the central policy decision point for all userspace access control decisions, while the SE Android kernelspace security server is responsible for all kernelspace policy decisions. This approach provides a clear separation of security issues between the userspace and the kernelspace components. Furthermore, it enables at middleware level the use of a more dynamic policy schema (different from the more static SELinux policy language) which takes advantage of the rich semantics (e.g., contextual information) at that layer. Access control is implemented as type enforcement based on (1) the subject type (usually the type associated with the calling app), (2) the object type (e.g., `contacts_email` or the type associated with the callee app UID), (3) the object class (e.g., `contacts_data` or `Intent`), and (4) the operation on the object (e.g. `query`). The **Userspace Security**

Server (USSS) is implemented as part of the Android system server (`com.android.server`) and comprises 3741 lines of Java code. It exposes an interface to the USOMs for requesting access control decisions over ICC (cf. Figure 1).

4.2.3 Userspace Object Managers

In FlaskDroid, middleware services and apps act as **Userspace Object Managers (USOMs)** for their respective objects. These services and apps can be distinguished into system components and 3rd party components. The former, i.e., pre-installed services and apps, inevitably have to be USOMs to achieve the desired system security and privacy, while the latter can use interfaces provided by the **Userspace Security Server** to optionally act as USOMs.

Table 4 in Appendix B provides an overview of exemplary *system* USOMs in FlaskDroid and shows some typical operations each object manager controls. Currently, the USOMs implemented in FlaskDroid comprise 136 policy enforcement points. In the following, we explain how we instrumented selected components as **Userspace Object Managers**.

PackageManagerService is responsible for (un)installation of application packages. Furthermore, it is responsible for finding a preferred component for doing a task at runtime. For instance, if an app sends an `Intent` to display a PDF, the **PackageManagerService** looks for a preferred **Activity** able to perform the task.

As a **Userspace Object Manager**, we extend the **PackageManagerService** to assign consolidated middleware- and kernel-level app types to all apps during installation using criteria defined in the policy (cf. Section 4.3). This is motivated by the fact that at the time a policy is written, one cannot predict which 3rd party apps will be installed in the future. Pre-installed apps are labeled during the phone’s boot cycle based on the same criteria. More explicitly, we assign app types to the (shared) UIDs of apps, since (shared) UIDs are the smallest identifiable unit for application sandboxes. In addition, pre-defined UIDs in the system are reserved for particular system components³ and we map these UIDs to pre-defined types (e.g., `aid_root_t` or `aid_audio_t`). Furthermore, we extend the logic for finding a preferred component to only consider apps which are allowed by the policy to perform the requested task.

ActivityManagerService is responsible for managing the stack of **Activities** of different apps, Activity life-cycle management, as well as providing the `Intent`

³These pre-defined UIDs on Android 4.0.4 are found in `system/core/include/private/android_filesystem_config.h`

broadcast system. As a USOM, the `ActivityManagerService` is responsible for labeling `Activity` and `Intent` objects and enforcing access control on them. Activities are labeled according to the apps they belong to, i.e., the UID of the application process that created the `Activity`. Subsequently, access control on the `Activity` objects is enforced in the `ActivityStack` subsystem of the `ActivityManagerService`. During operations that manipulate `Activities`, such as moving `Activities` to the foreground/background or destroying them, the `ActivityStack` queries the `USSS` in order to verify that the particular operations are permitted to proceed depending on the subject type (i.e., the calling app) and object type (i.e., the app owning the `Activity` being modified).

Similar to apps, `Intents` are labeled based on available meta-information, such as the action and category string or the sender app (cf. Section 4.3.1). To apply access control to `Broadcast Intents`, we followed a design pattern as proposed in [39, 9]. We modified the `ActivityManagerService` to filter out receivers which are not allowed to receive `Intents` of the previously assigned type (e.g., to prevent apps of lower security clearance from receiving `Broadcasts` by an app of a higher security clearance).

Content Providers are the primary means for apps to share data. This data can be accessed over a well-defined, SQL-like interface. As `Userspace Object Managers`, `ContentProviders` are responsible for assigning labels to the data entries they manage during insertion/creation of data and for performing access control on update, query, or deletion of entries. Two approaches for access control are supported: 1) at the API level by controlling access to the provider as a whole or 2) integrating it into the storage back-end (e.g., `SQLite` database) for more fine-grained per-data access control.

For approach 2), we implemented a design pattern for `SQLite`-based `ContentProviders`. Upon insertion or update of entries, we verify that the subject type of the calling app is permitted to perform this operation on the particular object type. To filter queries to the database we create one `SQL View` for each subject type and redirect the query of each calling app to the respective `View` for its type. Each `View` implements a filtering of data based on an access control table managed by the `USSS` which represents the access control matrix for subject/object types. This approach is well-suited for any `SQLite`-based `ContentProvider` and scales well to multiple stakeholders by using nested `Views`.

Service components of an app provide a particular functionality to other (possibly remote) components, which access the `Service` interface via `ICC`. To instant-

iate a `Service` as a `Userspace Object Manager`, we add access control checks when a (remote) component connects to the `Service` and on each call to `Service` functions exposed by the `Service API`. The developer of the `Service` can set the types of the service and its functions by adding type-tags to their definitions.

`Service` interfaces are exposed as `Binder IPC` objects that are generated based on an interface specification described in the `Android Interface Definition Language (AIDL)`. We extended the lexer and parser of `Android's AIDL` tool to recognize (developer-defined) type tags on `Service` interfaces and function declarations. The `AIDL` code generator was extended to automatically insert policy checks for these types in the auto-generated `Service` code. Since the `AIDL` tool is used during build of the system as well as part of the `SDK` for app development, this solution applies to both system `Services` and *3rd* party app `Services` in the same way.

4.2.4 Context Providers

A context is an abstract term that represents the current security requirements of the device. It can be derived from different criteria, such as physical criteria (e.g., the location of the device) or the state of apps and the system (e.g., the app being currently shown on the screen). To allow for flexible control of contexts and their definitions, our design employs **Context Providers**. These providers come in form of plugins to our `Userspace Security Server` (see Figure 1) and can be arbitrarily complex (e.g., use machine learning) and leverage available information such as the network state or geolocation of the device to determine which contexts apply. `Context Providers` register Listener threads in the system to detect context changes similar to the approach taken in [15]. Each `Context Provider` is responsible for a distinct set of contexts, which it activates/deactivates in the `USSS`. Decoupling the context monitoring and definition from our policy provides that context definitions do not affect our policy language except for very simple declarations (as we will show in Section 4.3.1).

Moreover, the `USSS` provides feedback to `Context Providers` about the performed access control decisions. This is particularly useful when instantiating security models like [8, 15] in which access control decisions depend on previous decisions.

4.3 Policy

4.3.1 Policy Language and Extensions

We extend SELinux's policy semantics for *type enforcement* (cf. Section 2.3) with new default classes

Listing 1: Assigning types to apps and Intents

```

1 defaultAppType untrustedApp_t;
2 defaultIntentType untrustedIntent_t;
3
4 appType app_telephony_t {
    Package:package_name=com.android.phone; };
5
6 intentType intentLaunchHome_t {
    Action:action_string=android.intent.action.MAIN;
    Categories:category=android.intent.category.HOME;};
8

```

and constructs for expressing policies on both middleware and kernel-level. A recapitulation of the SELinux policy language is out of scope of this paper and we focus here on our extensions.

New default classes. Similar to classes at the kernel-level, like *file* or *socket*, we introduce new default classes and their corresponding operations to represent common objects at middleware level, such as **Activity**, **Service**, **ContentProvider**, and **Intent**. Operations for these classes are, for example, *query* a **ContentProvider** or *receive* an **Intent**.

Application and Intent Types. A further extension is the possibility to define criteria by which applications and **Intents** are labeled with a security type (cf. Listing 1). The criteria for apps can be, for instance, the application package name, the requested permissions or the developer signature. Criteria for assigning a type to **Intent** objects can be the **Intent** action string, category or receiving component. If no criteria matched, a default type is assigned to apps (line 1) and **Intents** (line 2), respectively.

Context definitions and awareness. We extend the policy language with an option to declare *contexts* to enable context-aware policies. Each declared context can be either *activated* or *deactivated* by a dedicated Context Provider (cf. Section 4.2).

To actually enable context-aware policies, we introduce in our policy language *switchBoolean* statements which map contexts to booleans, which in turn provide dynamic policies. Listing 2 presents the definition of booleans and *switchBoolean* statements. For instance, the *switchBoolean* statement in lines 4-9 defines that as soon as the context *phoneBooth_con* is active, the boolean *phoneBooth_b* has to be set to *true*. As soon as the *phoneBooth_con* context is deactivated, the *phoneBooth_b* boolean should be reset to its initial value (line 6). To map contexts to the kernel-level, we introduce *kbool* definitions (line 2), which point to a boolean at kernel level instead of adding a new boolean at middleware level. Changes to such kernel-mapped boolean values by *switchBoolean* statements trigger a call to the SELinux kernel module to update the corresponding

Listing 2: Linking booleans with contexts

```

1 bool phoneBooth_b = false;
2 kbool allowIPTablesExec_b = true;
3
4 switchBoolean {
5     context=phoneBooth_con;
6     auto_reverse=true;
7     phoneBooth_b=true;};
7

```

SELinux boolean.

4.3.2 Support for Multiple Stakeholders

A particular requirement for the design of FlaskDroid is the protection of interests of different stakeholders. This requires that policy decisions consider the policies of all involved stakeholders. These policies can be pre-installed (i.e., system policy), delivered with apps (i.e., app developer policies), or configured by the user (e.g., *User Policy App* in Figure 1).

In FlaskDroid, 3rd party app developers may optionally ship app-specific policies with their application packages and additionally choose to instrument their app components as Userspace Object Managers for their own data objects. FlaskDroid provides the necessary interfaces to query the Userspace Security Server for policy decisions as part of the SDK. These decisions are based on the app-specific 3rd party policy, which defines custom **appType** statements to label subjects (e.g., other apps) and declares app-specific object types. To register app-specific policies, the **PackageManagerService** is instrumented such that it extracts policy files during app installation and injects them into the USSS.

A particular challenge when supporting multiple stakeholders is the reconciliation of the various stakeholders' policies. Different strategies for reconciliation are possible [43, 35] and generally supported by our architecture, based on namespaces and global/local type definitions. For instance, as discussed in [43], *all-allow* (i.e., all stakeholder policies must allow access), *any-allow* (i.e., only one stakeholder policy must allow access), *priority* (i.e., higher ranked stakeholder policies override lower ranked ones), or *consensus* (i.e., at least one stakeholder policy allows and none denies or vice versa). However, choosing the right strategy strongly depends on the use-case. For example, on a pure business smartphone without a user-private domain, the system (i.e., company) policy always has the highest priority, while on a private device a consensus strategy may be preferable.

We opted for a consensus approach, in which the *system* policy check is mandatory and must always consent for an operation to succeed.

5 Use-cases / Instantiations

In the following we will show how FlaskDroid can instantiate certain privacy and security protecting use-cases. More use-cases and concrete examples are provided in our technical report [11].

5.1 Privacy Enhanced System Services and Content Providers

System Services and ContentProviders are an integral part of the Android application framework. Prominent Services are, for instance, the `LocationManager` or the `Audio Services` and prominent ContentProviders are the contacts app and SMS/MMS app. By default, Android enforces permission checks on access to the interfaces of these Services and Providers.

Problem description: The default permissions are non-revocable and too coarse-grained and protect access only to the entire Service/Provider but not to specific functions or data. Thus, the user cannot control in a fine-grained fashion which sensitive data can be accessed how, when and by whom. Apps such as Facebook and WhatsApp have access to the entire contacts database although only a subset of the data (i.e., email addresses, phone numbers and names) is required for their correct functioning. On the other hand, recent attacks demonstrated how even presumably privacy-unrelated and thus unprotected data (e.g. accelerometer readings) can be misused against user's security and privacy [53, 12].

Solution: Our modified AIDL tool automatically generates policy checks for each Service interface and function in the system. We tagged selected query functions of the system `AudioService`, `LocationManager`, and `SensorManager` with specific security contexts (e.g., `fineGrainedLocation_t` as `object_type`, `locationService_c` as `object_class`, and `getLastKnownLocation` as `operation`) to achieve fine-grained access control on this information. Our policy states that calling functions of this object type is prohibited while the phone is in a security sensitive state. Thus, retrieving accelerometer information or recording audio is not possible when, e.g., the virtual keyboard/PIN pad is in the foreground or a phone call is in progress.

In Section 4.2.3 we explained how ContentProviders (e.g. the `ContactsProvider`) can act as User-space Object Managers. As an example, users can refine the system policy to further restrict access to their contacts' data. A user can, for instance, grant the Facebook app read access to their "friends" and "family" contacts' email addresses and names, while prohibiting it from reading their postal addresses and any data of other groups such as "work".

5.2 App Developer Policies (Saint)

Ongtang et al. present in [39] an access control framework, called *Saint*, that allows app developers to ship their apps with policies that regulate access to their apps' components.

Problem description: The concrete example used to illustrate this mechanism consists of a shopping app whose developer wants to restrict the interaction with other 3rd party apps to only specific payment, password vault, or service apps. For instance, the developer specifies that that the password vault app must be at least version 1.2 or that a personal ledger app must not hold the Internet permission.

The policy rules for the runtime enforcement of Saint on Inter-Component communication (ICC) are defined as the tuple (`Source`, `Destination`, `Conditions`, `State`). `Source` defines the source app component of the ICC and optional parameters for an `Intent` object (e.g., action string). `Destination` describes similarly the destination app component of the ICC. `Conditions` are optional conjunctional conditions (e.g., permissions or signature key of the destination app) and `State` describes the system state (e.g., geolocation or bluetooth adapter state).

Solution: Instantiating Saint's runtime access control on FlaskDroid is achieved by mapping Saint's parameters to the type enforcement implemented by FlaskDroid. Thus, `Source`, `Destination`, and `Conditions` are combined into security types for the subject (i.e., source app) and object (i.e., destination app or Intent object). For instance, a specific type is assigned to an app with a particular signature and permission. If this app is source in the Saint policy, it is used as subject type in FlaskDroid policy rules; and if it is used as destination, it is used as object type. The object class and operation are directly derived from the destination app. The *system state* can be directly expressed by *booleans* and *switchBoolean* statements in the policy and an according Context Provider. Appendix A provides a concrete policy example for the instantiation of the above shopping app example.

6 Evaluation and Discussion

In this section we evaluate and discuss our architecture in terms of policy design, effectiveness, and performance overhead.

6.1 Policy

To evaluate our FlaskDroid architecture, we derived a basic policy that covers the pre-installed system

USOMs that we introduced in Section 4.2.3.

Policy Assessment. For FlaskDroid we are for now foremost interested in generating a *basic policy* to estimate the access control complexity that is inherent to our design, i.e., the number of new types, classes, and rules required for the *system Userspace Object Managers*. This basic policy is intended to lay the foundation for the development of a *good* policy, i.e., a policy that covers *safety*, *completeness*, and *effectiveness* properties. However, the development of a security policy that fulfills these properties is a highly complex process. For instance, on SELinux enabled systems the policies were incrementally developed and improved after the SELinux module had been introduced, even inducing research on verification of these properties [24]. A similar development can be currently observed for the SE Android policies which are written from scratch [48] and we envision inducing a similar research on development and verification of FlaskDroid policies.

Basic Policy Generation. To generate our basic policy, we opted for an approach that follows the concepts of TOMOYO Linux’ learning phase⁴ and other semi-automatic methods [42]. The underlying idea is to derive policy rules directly from observed application behavior. To generate a log of system application behavior, we leveraged FlaskDroid’s audit mode, where policy checks are logged but not enforced. Under the assumption, that the system contained in this auditing phase only trusted apps, this trace can be used to derive policy rules.

To achieve a high coverage of app functionality and thus log all required access rights, we opted for testing with human user trials for the following reasons: First, automated testing has been shown to exhibit a potentially very low code coverage [24] and, second, Android’s extremely event-driven and concurrent execution model complicates static analysis of the Android system [56, 24]. However, in the future, static analysis based (or aided) generation of access control rules is more preferable in order to cover also corner-cases of applications’ control-flows.

The users’ task was to thoroughly use the pre-installed system apps by performing various everyday tasks (e.g., maintaining contacts, writing SMS, browsing the Internet, or using location-based services). To analyze interaction between apps, a particular focus of the user tasks was to leverage inter-app functionality like sharing data (e.g., copying notes from a website into an SMS). For testing, the users were handed out Galaxy Nexus devices running FlaskDroid with a *No-allow-rule* policy. This is a

manually crafted policy containing only the required subject/object types, classes and operations for the USOMs in our architecture, but no allow rules. The devices were also pre-configured with test accounts (e.g., EMail) and test data (e.g., fake contacts).

Using the logged access control checks from these trials, we derived 109 access control rules required for the correct operation of the system components (as observed during testing), which we learned to be partially operationally dependent on each other. Our pre-installed middleware policy contained 111 types and 18 classes for a fine-granular access control to the major system Services and ContentProviders (e.g., ContactsProvider, LocationManager, PackageManagerService, or SensorManager). These rules (together with the above stated type and object definitions) constitute our *basic policy*. Although SELinux policies cannot be directly compared to our policy, since they target desktop operating systems, the difference in policy complexity (which is in the order of several magnitudes [11]) underlines that the design of mobile operating systems facilitates a clearer mandatory access control architecture (e.g., separation of duties). This profits an easier policy design (as supported by the experiences from [54, 36]).

3rd Party Policies. The derived basic policy can act as the basis on top of which additional user, 3rd party, and use-case specific policies can be deployed (cf. Section 5). In particular, we are currently working on extending the basic policy with types, classes and allow rules for popular apps, such as WhatsApp or Facebook, which we further evaluated w.r.t. user’s privacy protection (cf. Section 6.2). A particular challenge is to derive policies which on the one hand protect the user’s privacy but on the other hand preserve the intended functionality of the apps. Since the user privacy protection strongly depends on the subjective security objectives of the user, this approach requires further investigation on how the user can be involved in the policy configuration [58].

However, as discussed in Sections 3 and 4.2.2, multiple policies by different stakeholders with potentially conflicting security objectives require a reconciliation strategy. Devising a general strategy applicable to all use-cases and satisfying all stakeholders is very difficult, but use-case specific strategies are feasible [44, 29]. In our implementation, we opted for a consensus approach, which we successfully applied during implementation of our use-cases (cf. Section 5). We explained further strategies in Section 4.3.2.

⁴<http://tomoyo.sourceforge.jp/2.2/learning.html.en>

Attack	Test
Root Exploit	mempodroid Exploit
App executed by root	Synthetic Test App
Over-privileged and Information-Stealing Apps	Known malware Synthetic Test App WhatsApp v2.8.4313 Facebook v1.9.1
Sensory Malware	Synthetic Test App [53, 12, 46]
Confused Deputy	Synthetic Test App
Collusion Attack	Synthetic Test Apps [46]

Table 1: List of attacks considered in our testbed

6.2 Effectiveness

We decided to evaluate the effectiveness of FlaskDroid based on empirical testing using the security models presented in Section 5 as well as a testbed of known malware retrieved from [55, 3] and synthetic attacks (cf. Table 1). Alternative approaches like static analysis [18] would benefit our evaluation but are out of scope of this paper and will be addressed separately in future work.

Root exploits. SE Android successfully mitigates the effect of the *mempodroid* attack. While the exploit still succeeds in elevating its process to root privileges, the process is still constrained by the underlying SE Android policy to the limited privileges granted to the root user [48].

Malicious apps executed by root. While SE Android constrains the file-system privileges of an app process executed with root UID, this process still inherits all Permissions at middleware level. In FlaskDroid, the privileges of apps running with this omnipotent UID are restricted to the ones granted by the system policy to root (cf. `aid_root_t` in Section 4.2.3). During our user tests, we had to define only one allow rule for the `aid_root_t` type on the middleware layer, which is not surprising, since usually Android system or third-party apps are not executed by the root user. Thus, a malicious app gaining root privileges despite SE Android, e.g., using the *mempodroid* exploit [48], is in FlaskDroid restricted at both kernel and middleware level.

Over-privileged and information stealing apps. We verified the effectiveness of FlaskDroid against over-privileged apps using a) a synthetic test app which uses its permissions to access the `ContactsProvider`, the `LocationManager` and the `SensorManager` as 3rd party apps would do; b) malware such as `Android.Loozfon` [2] and `Android.Enesoluty` [1] which steal user private information; and c) unmodified apps from Google Play, including the popular WhatsApp messenger and Facebook apps. In all cases, a corresponding policy on FlaskDroid successfully and

gracefully prevented the apps and malware from accessing privacy critical information from sources such as the `ContactsProvider` or `LocationManager`.

Sensory malware. To mitigate sensory malware like *TapLogger* [53] and *TouchLogger* [12], we deployed a context-aware FlaskDroid policy which causes the `SensorManager USOM` to filter acceleration sensor information delivered to registered `SensorListeners` while the on-screen keyboard is active. Similarly, a second policy prevents the *SoundComber* attack [46] by denying any access to the audio record functionality implemented in the `MediaRecorderClient USOM` while a call is in progress.

Confused deputy and collusion attacks. Attacks targeting confused deputies in system components (e.g. `SettingsAppWidgetProvider` [41]) are addressed by fine-grained access control rules on ICC. Our policy restricts which app types may send (broadcast) `Intents` reserved for system apps.

Collusion attacks are in general more challenging to handle, especially when covert channels are used for communication. Similar to the mitigation of confused deputies, a FlaskDroid policy was used to prohibit ICC between colluding apps based on specifically assigned app types. However, to address collusion attacks *efficiently*, more flexible policies are required. We already discussed in Section 4.2.4 a possible approach to instantiate *XManDroid* [8] based on our Context Providers and we elaborate in the subsequent Section 6.3 on particular challenges for improving the mitigation of collusion attacks.

6.3 Open Challenges and TCB

Information flows within apps. Like any other access control system, e.g., SELinux, exceptions for which enforcement falls short concern attacks which are legit within the policy rules. Such shortcomings may lead to unwanted information leakage. A particular challenge for addressing this problem and controlling access and separation (*non-interference*) of security relevant information are information flows within apps. Access control frameworks like FlaskDroid usually operate at the granularity of application inputs/outputs but do not cover the information flow within apps. For Android security, this control can be crucial when considering attacks such as collusion attacks and confused deputy attacks. Specifically for Android, taint tracking based approaches [19, 28, 45] and extensions to Android’s IPC mechanism [17] have been proposed. To which extend these approaches could augment the coverage and hence effectiveness of FlaskDroid has to be explored in future work.

User-centric and scalable policies. While

FlaskDroid is a sophisticated access control framework for enforcing security policies and is already now valuable in specific scenarios with fixed policies like business phones or locked-down devices [11], a particular challenge of the forthcoming policy engineering are user-centric and scalable policies for off-the-shelf end-user devices. Although expert-knowledge can be used to engineer policies for the static components of the system, similar to common SELinux-enabled distributions like Fedora, an orthogonal, open research problem is how to efficiently determine the individual end-users' security and privacy requirements and how to map these requirements scalable to FlaskDroid policy rules w.r.t. the plethora of different apps available. To this end, we started exploring approaches to provide the end-user with tools that abstract the underlying policies [10]. Furthermore, the policy-based classification of apps at install-time applied in FlaskDroid could in the future be augmented by different or novel techniques from related fields, e.g., role-mining for RBAC systems [51], to assist the end-user in his decision processes.

Trusted Computing Base. Moreover, while SE Android as part of the kernel is susceptible to kernel-exploits, our middleware extensions might be compromised by attacks against the process in which they execute. Currently our `SecurityServer` executes within the scope of the rather large Android system server process. Separating the `SecurityServer` as a distinct system process with a smaller attack surface (smaller TCB) can be efficiently accomplished, since there is no strong functional inter-dependency between the system server and FlaskDroid's `SecurityServer`.

6.4 Performance Overhead

Middleware layer. We evaluated the performance overhead of our architecture based on the *No-allow-rule* policy and the basic policy presented in Section 6.1 using a Samsung Galaxy Nexus device running FlaskDroid. Table 2 presents the mean execution time μ and standard deviation σ for performing a policy check at the middleware layer in both policy configurations (measured in μs) as well as the average memory consumption (measured in MB) of the process in which our `Userspace Security Server` executes (i.e., the system server). Mean execution time and standard deviation are the amortized values for both cached and non-cached policy decisions.

In comparison to permission checks on a vanilla Android 4.0.4 both the imposed runtime and memory overhead are acceptable. The high standard deviation is explained by varying system loads, however,

	μ (in μs)	σ (in μs)	memory (in MB)
FlaskDroid			
No-allow-rule	329.505	780.563	15.673
Basic policy	452.916	4887.24	16.184
Vanilla Android 4.0.4			
Permission check	330.800	8291.805	15.985

Table 2: Runtime and memory overhead

	μ (in ms)	σ (in ms)
FlaskDroid (Basic policy)	0.452	4.887
XManDroid [8] (Amortized)	0.532	2.150
TrustDroid [9]	0.170	1.910

Table 3: Performance comparison to related works

Figure 2 presents the cumulative frequency distribution for our policy checks. The shaded area represents the 99.33% confidence interval for our basic policy with a maximum overhead of $2ms$.

In comparison to closest related work [8, 9] (cf. Section 7), FlaskDroid achieves a very similar performance. Table 3 provides an overview of the average performance overhead of the different solutions. *TrustDroid* [9] profits from the very static policies it enforces, while FlaskDroid slightly outperforms *XManDroid* [8]. However, it is hard to provide a completely fair comparison, since both TrustDroid and XManDroid are based on Android 2.2 and thus have a different baseline measurement. Both [8, 9] report a baseline of approximately $0.18ms$ for the default permission check, which differs from the $0.33ms$ we observed in Android 4.0.4 (cf. Table 2).

Kernel layer. The impact of SE Android on Android system performance has been evaluated previously by its developers [48]. Since we only minimally add/modify the default SE Android policy to cater for our use-cases (e.g., new booleans), the negligible performance overhead presented in [48] still applies to our current implementation.

7 Related Work

7.1 Mandatory Access Control

The most prominent MAC solution is SELinux [33] and we elaborated on it in detail in our Background and Requirements Sections 2 and 3. Specifically for mobile platforms, related work [54, 36] has investigated the placement of SELinux enforcement hooks in the operating system and user-space services on OpenMoko [36] and the LiMo (Linux Mobile) platform [54]. Our approach follows along these ideas but for the Android middleware.

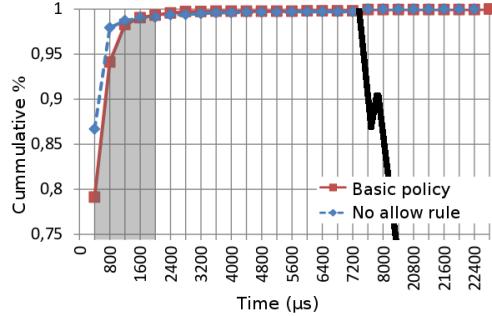


Figure 2: CDF of the performance overhead. Shaded area represents the 99.33 confidence interval for checks with *Basic policy*.

Also TOMOYO Linux [27], a path-based MAC framework, has been leveraged in Android security extensions [8][9]. Although TOMOYO supports more easily policy updates at runtime and does not require extended file system attributes, SELinux is more sophisticated, supports richer policies, and covers more object classes [5].

However, as we state in Section 3, low-level MAC alone is insufficient. In this paper we show how to extend the SE Android security architecture into the Android middleware layer for policy enforcement.

7.2 SE Android MMAC

The SE Android project was recently extended by different mechanisms for mandatory access control at Android’s middleware layer [47], denoted as **MMAC**: **Permission revocation** is a simple mechanism to dynamically revoke permissions by augmenting the default Android permission check with a policy driven check. When necessary, this additional check overrules and negates the result of the default check.

However, this permission revocation is in almost all cases unexpected for app developers, which rely on the fact that if their app has been installed, it has been granted all requested permissions. Thus, developers very often omit error handling code for permission denials and hence unexpectedly revoking permissions easily leads to application crashes.

In FlaskDroid, policy enforcement also effectively revokes permissions. However, we use **USOMs** which integrate the policy enforcement into the components which manage the security and privacy sensitive data. Thus, our **USOMs** apply enforcement mechanisms that are *graceful*, i.e., they do not cause unexpected behavior that can cause application crashes. Related work (cf. Section 7.3) introduced some of these graceful enforcement mechanisms, e.g., filtering table rows and columns from **ContentProvider**

responses [58, 15, 28, 8, 9].

Intent MAC protects with a white-listing enforcement the delivery of **Intents** to **Activities**, **Broadcast Receivers**, and **Services**. Technically, this approach is similar to prior work like [58, 8, 9]. The white-listing is based on attributes of the **Intent** objects (e.g., the value of the action string) and the security type assigned to the **Intent** sender and receiver apps.

In FlaskDroid, we apply a very similar mechanism by assigning **Intent** objects a security type, which we use for type enforcement on **Intents**. While we acknowledge, that access control on **Intents** is important for the overall coverage of the access control, Intent MAC alone is insufficient for policy enforcement on inter-app communications. A complete system has to consider also other middleware communications channels, such as Remote Procedure Calls (RPC) to **Service** components and to **ContentProviders**. By instrumenting these components as **USOMs** and by extending the AIDL compiler (cf. Section 4.2) to insert policy enforcement points, we address these channels in FlaskDroid and provide a non-trivial complementary access control to Intent MAC.

Install-time MAC performs, similar to *Kirin* [20], an install-time check of new apps and denies installation when an app requests a defined combination of permissions. The adverse permission combinations are defined in the SE Android policy.

While FlaskDroid does not provide an install-time MAC, we consider this mechanism orthogonal to the access control that FlaskDroid already provides and further argue that it could be easily integrated into existing mechanisms of FlaskDroid (e.g., by extending the install-time labeling of new apps with a blacklist-based rejection of prohibited app types).

7.3 Android Security Extensions

In the recent years, a number of security extensions to the Android OS have been proposed.

Different approaches [38, 37, 15, 39] add mandatory access control mechanisms to Android, tailored for specific problem sets such as providing a DRM mechanism (*Porscha* [38]), providing the user with the means to selectively choose the permissions and runtime constraints each app has (*APEX* [37] and *CREPE* [15]), or fine-grained, context-aware access control to enable developers to install policies to protect the interfaces of their apps (*Saint* [39]). Essentially all these solutions extend Android with MAC at the middleware layer. The explicit design goal of our architecture was to provide an ecosystem that is flexible enough to instantiate those related works based on policies (as demonstrated in Section 5 at

the example of *Saint*) and additionally providing the benefit of a consolidated kernel-level MAC.

The pioneering framework *TaintDroid* [19] introduced the tracking of tainted data from sensible sources on Android and successfully detected unauthorized information leakage. The subsequent *AppFence* architecture [28] extended TaintDroid with checks that not only detect but also prevent such unauthorized leakage. However, both TaintDroid and AppFence do not provide a generic access control framework. Nevertheless, future work could investigate their applicability in our architecture, e.g., propagating the security context of data objects. The general feasibility of such “context propagation” has been shown in the *MOSES* [45] architecture.

Inlined Reference Monitors (IRM) [52, 7, 30] place policy enforcement code for access control directly in 3rd party apps instead of relying on a system centric solution. An unsolved problem of *inlined* monitoring in contrast to a system-centric solution is that the reference monitor and the potentially malicious code share the same sandbox and that the monitor is *not* more privileged than the malicious code and thus prone to compromise.

The closest related work to FlaskDroid with respect to a two layer access control are the *XManDroid* [8] and *TrustDroid* [9] architectures. Both leverage TOMOYO Linux as kernel-level MAC to establish a separate security domain for business apps [9], or to mitigate collusion attacks via kernel-level resources [8]. Although they cover MAC enforcement at both middleware and kernel level, both systems support only a very static policy tailored to their specific purposes and do not support the instantiation of different use-cases. In contrast, FlaskDroid can instantiate the XManDroid and TrustDroid security models by adjusting policies. For instance, different security types for business and private apps could be assigned at installation time, and boolean flags can be used to dynamically prevent two apps from communicating if this would form a collusion attack.

8 Conclusion

In this paper, we present the design and implementation of FlaskDroid, a policy-driven generic two-layer MAC framework on Android-based platforms. We introduce our efficient policy language that is tailored for Android’s middleware semantics. We show the flexibility of our architecture by policy-driven instantiations of selected security models, including related work (*Saint*) and privacy-enhanced system components. We demonstrate the applicability of our design by prototyping it on Android 4.0.4. Our

evaluation shows that the clear API-oriented design of Android benefits the effective and efficient implementation of a generic mandatory access control framework like FlaskDroid.

Availability

The source code for FlaskDroid is available online at <http://www.flaskdroid.org>.

References

- [1] Android.Enesoluty | Symantec. http://www.symantec.com/security_response/writeup.jsp?docid=2012-082005-5451-99.
- [2] Android.Loozfon | Symantec. http://www.symantec.com/security_response/writeup.jsp?docid=2012-082005-5451-99.
- [3] Contagio Mobile. <http://contagiominidump.blogspot.de/>.
- [4] Gartner Says Worldwide Mobile Phone Sales Declined 1.7 Percent in 2012. <http://www.gartner.com/newsroom/id/2335616>.
- [5] TOMOYO Linux Wiki: How is TOMOYO Linux different from SELinux and AppArmor? <http://tomoyo.sourceforge.jp/wiki-e/?WhatIs#comparison>.
- [6] WhatsApp reads your phone contacts and is breaking privacy laws. <http://www.digitaltrends.com/mobile/whatsapp-breaks-privacy-laws/>.
- [7] BACKES, M., GERLING, S., HAMMER, C., AND VON STYPEREKOWSKY, P. Appguard - enforcing user requirements on android apps. In *19th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)* (2013), Springer-Verlag.
- [8] BUGIEL, S., DAVI, L., DMITRIENKO, A., FISCHER, T., SADEGHI, A.-R., AND SHAstry, B. Towards taming privilege-escalation attacks on android. In *NDSS* (2012), The Internet Society.
- [9] BUGIEL, S., DAVI, L., DMITRIENKO, A., HEUSER, S., SADEGHI, A.-R., AND SHAstry, B. Practical and lightweight domain isolation on android. In *1st ACM CCS Workshop on Security and Privacy in Mobile Devices (SPSM)* (2011), ACM.
- [10] BUGIEL, S., HEUSER, S., AND SADEGHI, A.-R. myTunes: Semantically Linked and User-Centric Fine-Grained Privacy Control on Android. Tech. Rep. TUD-CS-2012-0226, Center for Advanced Security Research Darmstadt, Nov. 2012.
- [11] BUGIEL, S., HEUSER, S., AND SADEGHI, A.-R. Towards a Framework for Android Security Modules: Extending SE Android Type Enforcement to Android Middleware. Tech. Rep. TUD-CS-2012-0231, Center for Advanced Security Research Darmstadt, December 2012.
- [12] CAI, L., AND CHEN, H. Touchlogger: inferring keystrokes on touch screen from smartphone motion. In *6th USENIX conference on Hot topics in security (HotSec)* (2011), USENIX Association.
- [13] CARTER, J. Using gconf as an example of how to create an userspace object manager, 2007.

- [14] CHIN, E., FELT, A. P., GREENWOOD, K., AND WAGNER, D. Analyzing inter-application communication in Android. In *MobiSys* (2011), ACM.
- [15] CONTI, M., NGUYEN, V. T. N., AND CRISPO, B. CRePE: Context-related policy enforcement for Android. In *13th Information Security Conference (ISC)* (2010), Springer-Verlag.
- [16] DAVI, L., DMITRIENKO, A., SADEGHI, A.-R., AND WINANDY, M. Privilege escalation attacks on Android. In *13th Information Security Conference (ISC)* (2010), Springer-Verlag.
- [17] DIETZ, M., SHEKHAR, S., PISETSKY, Y., SHU, A., AND WALLACH, D. S. Quire: Lightweight provenance for smartphone operating systems. In *USENIX Security* (2011), USENIX Association.
- [18] EDWARDS, A., JAEGER, T., AND ZHANG, X. Runtime verification of authorization hook placement for the Linux security modules framework. In *CCS* (2002), ACM.
- [19] ENCK, W., GILBERT, P., CHUN, B.-G., COX, L. P., JUNG, J., McDANIEL, P., AND SHETH, A. N. TaintDroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *OSDI* (2010), USENIX Association.
- [20] ENCK, W., ONGTANG, M., AND McDANIEL, P. On lightweight mobile phone application certification. In *CCS* (2009), ACM.
- [21] ENCK, W., ONGTANG, M., AND McDANIEL, P. Understanding Android security. *IEEE Security and Privacy Magazine* 7 (2009), 50–57.
- [22] F-SECURE LABS. Mobile Threat Report: Q3 2012, 2012.
- [23] FEDERAL TRADE COMMISSION. Path social networking app settles FTC charges it deceived consumers and improperly collected personal information from users' mobile address books. <http://www.ftc.gov/opa/2013/02/path.shtm>, Jan. 2013.
- [24] GILBERT, P., CHUN, B.-G., COX, L. P., AND JUNG, J. Vision: automated security validation of mobile apps at app markets. In *2nd international workshop on Mobile cloud computing and services (MCS)* (2011), ACM.
- [25] GRACE, M. C., ZHOU, W., JIANG, X., AND SADEGHI, A.-R. Unsafe exposure analysis of mobile in-app advertisements. In *WiSec* (2012), ACM.
- [26] GUTTMAN, J. D., HERZOG, A. L., RAMSDELL, J. D., AND SKORUPKA, C. W. Verifying information flow goals in security-enhanced linux. *Journal on Computer Security* 13, 1 (Jan. 2005), 115–134.
- [27] HARADA, T., HORIE, T., AND TANAKA, K. Task Oriented Management Obviates Your Onus on Linux. In *Linux Conference* (2004).
- [28] HORNYACK, P., HAN, S., JUNG, J., SCHECHTER, S., AND WETHERALL, D. These aren't the droids you're looking for: retrofitting android to protect data from imperious applications. In *CCS* (2011), ACM.
- [29] HU, H., AHN, G.-J., AND KULKARNI, K. Detecting and resolving firewall policy anomalies. *IEEE Transactions on Dependable and Secure Computing* 9, 3 (2012), 318–331.
- [30] JEON, J., MICINSKI, K. K., VAUGHAN, J. A., FOGL, A., REDDY, N., FOSTER, J. S., AND MILLSTEIN, T. Dr. Android and Mr. Hide: Fine-grained security policies on unmodified Android. In *2nd ACM CCS Workshop on Security and Privacy in Mobile Devices (SPSM)* (2012), ACM.
- [31] KOSTIAINEN, K., RESHETOVA, E., EKBERG, J.-E., AND ASOKAN, N. Old, new, borrowed, blue – a perspective on the evolution of mobile platform security architectures. In *CODASPY* (2011), ACM.
- [32] LINEBERRY, A., RICHARDSON, D. L., AND WYATT, T. These aren't the permissions you're looking for. BlackHat USA 2010. <http://dtors.files.wordpress.com/2010/08/blackhat-2010-slides.pdf>, 2010.
- [33] LOSCOCCO, P., AND SMALLEY, S. Integrating flexible support for security policies into the Linux operating system. In *FREENIX Track: USENIX Annual Technical Conference* (2001), USENIX Association.
- [34] MARFORIO, C., RITZDORF, H., FRANCILLON, A., AND CAPKUN, S. Analysis of the communication between collocating applications on modern smartphones. In *ACSAC* (2012), ACM.
- [35] McDANIEL, P., AND PRAKASH, A. Methods and limitations of security policy reconciliation. In *S&P* (2002), IEEE Computer Society.
- [36] MUTHUKUMARAN, D., SCHIFFMAN, J., HASSAN, M., SAWANI, A., RAO, V., AND JAEGER, T. Protecting the integrity of trusted applications in mobile phone systems. *Security and Communication Networks* 4, 6 (2011), 633–650.
- [37] NAUMAN, M., KHAN, S., AND ZHANG, X. Apex: Extending Android permission model and enforcement with user-defined runtime constraints. In *ASIA CCS* (2010), ACM.
- [38] ONGTANG, M., BUTLER, K., AND McDANIEL, P. Porscha: Policy oriented secure content handling in Android. In *ACSAC* (2010), ACM.
- [39] ONGTANG, M., McLAUGHLIN, S., ENCK, W., AND McDANIEL, P. Semantically rich application-centric security in Android. In *ACSAC* (2009), IEEE Computer Society.
- [40] PORTER FELT, A., FINIFTER, M., CHIN, E., HANNA, S., AND WAGNER, D. A survey of mobile malware in the wild. In *1st ACM workshop on Security and privacy in smartphones and mobile devices (SPSM)* (2011), ACM.
- [41] PORTER FELT, A., WANG, H., MOSCHUK, A., HANNA, S., AND CHIN, E. Permission re-delegation: Attacks and defenses. In *USENIX Security* (2011), USENIX Association.
- [42] PROVOS, N. Improving host security with system call policies. In *USENIX Security* (2003), USENIX Association.
- [43] RAO, V., AND JAEGER, T. Dynamic mandatory access control for multiple stakeholders. In *SACMAT* (2009), ACM.
- [44] REEDER, R. W., BAUER, L., CRANOR, L. F., REITER, M. K., AND VANIEA, K. More than skin deep: measuring effects of the underlying model on access-control system usability. In *International Conference on Human Factors in Computing Systems (CHI)* (2011), ACM.
- [45] RUSSELLO, G., CONTI, M., CRISPO, B., AND FERNANDES, E. MOSES: supporting operation modes on smartphones. In *SACMAT* (2012), ACM.
- [46] SCHLEGEL, R., ZHANG, K., ZHOU, X., INTWALA, M., KAPADIA, A., AND WANG, X. Soundcomber: A stealthy and context-aware sound trojan for smartphones. In *NDSS* (2011), The Internet Society.
- [47] SMALLEY, S. Middleware MAC for android. <http://kernsec.org/files/LSS2012-MiddlewareMAC.pdf>, Aug. 2012.

- [48] SMALLEY, S., AND CRAIG, R. Security Enhanced (SE) Android: Bringing Flexible MAC to Android. In *NDSS* (2013), The Internet Society.
- [49] SMITH, C. Privacy flaw in skype android app exposed. <http://www.t3.com/news/privacy-flaw-in-skype-android-app-exposed/>.
- [50] SPENCER, R., SMALLEY, S., LOSCOCCO, P., HIBLER, M., ANDERSEN, D., AND LEPREAU, J. The Flask security architecture: System support for diverse security policies. In *USENIX Security* (1999), USENIX Association.
- [51] VAIDYA, J., ATLURI, V., AND WARNER, J. RoleMiner: mining roles using subset enumeration. In *CCS* (2006), ACM.
- [52] XU, R., SAIDI, H., AND ANDERSON, R. Aurasium: Practical policy enforcement for android applications. In *USENIX Security* (2012), USENIX Association.
- [53] XU, Z., BAI, K., AND ZHU, S. Taplogger: inferring user inputs on smartphone touchscreens using on-board motion sensors. In *WiSec* (2012), ACM.
- [54] ZHANG, X., SEIFERT, J.-P., AND ACHIQMEZ, O. SEIP: simple and efficient integrity protection for open mobile platforms. In *International conference on Information and communications security (ICICS)* (2010), Springer-Verlag.
- [55] ZHOU, Y., AND JIANG, X. Dissecting android malware: Characterization and evolution. In *S&P* (2012), IEEE Computer Society.
- [56] ZHOU, Y., AND JIANG, X. Detecting passive content leaks and pollution in android applications. In *NDSS* (2013), The Internet Society.
- [57] ZHOU, Y., WANG, Z., ZHOU, W., AND JIANG, X. Hey, You, Get Off of My Market: Detecting Malicious Apps in Official and Alternative Android Markets. In *NDSS* (2012), The Internet Society.
- [58] ZHOU, Y., ZHANG, X., JIANG, X., AND FREEH, V. W. Taming information-stealing smartphone applications (on android). In *TRUST* (2011), Springer-Verlag.

A Concrete Instantiation of Saint policies with FlaskDroid

Listing 3 shows an instantiation of the developer policy in [39] on FlaskDroid. This policy is deployed by the shopping app and thus `self_t` refers to the shopping app. We define types `app_trustedPayApp_t`, `app_trustedPayApp_t`, `app_noInternetPerm_t` (lines 1-3 and lines 8-16) for the specific apps the shopping app is allowed to interact with and describe some of the allowed interactions by means of Intent types `intent_actionPay_t` and `intent_recordExpense_t` (lines 5-6 and lines 18-24). Afterwards, we declare access control rules that reflect the policy described in [39] (lines 26-28). For instance, the rule in line 26 defines that the shopping app is allowed to send an Intent with action string `ACTION_PAY` only to an app with type `app_trustedPayApp_t` (line 20), which in turn is only assigned to apps with the developer signature `308201...` (line 9).

Listing 3: Policy deployed by the shopping app, showing an instantiation of the Saint [39] runtime policy example.

```

1 type app_trustedPayApp_t;
2 type app_trustedPWVault_t;
3 type app_noInternetPerm_t;
4
5 type intent_actionPay_t;
6 type intent_recordExpense_t;
7
8 appType app_trustedPayApp_t {
9   Developer:signature=308201...; };
10
11 appType app_trustedPWVault_t {
12   Package:package_name=com.secure.passwordvault;
13   Package:min_version=1.2; };
14
15 appType app_noInternetPerm_t {
16   Package:permission=~android.permission.INTERNET; };
17
18 intentType intent_actionPay_t {
19   Action:action_string=ACTION_PAY;
20   Components:receiver_type=app_trustedPayApp_t; };
21
22 intentType intent_recordExpense_t {
23   Action:action_string=RECORD_EXPENSE;
24   Components:receiver_type=app_noInternetPerm_t; };
25
26 allow self_t intent_actionPay_t: intent_c { send };
27 allow self_t app_trustedPWVault_t: any { any };
28 allow self_t intent_recordExpense_t: intent_c { send };

```

B Userspace Object Managers

USOM	Example operations	
	Service USOMs	
PackageManagerService	getPackageInfo findPreferredActivity getInstalledApplications installPackage	findPreferredActivity getInstalledApplications installPackage
ActivityManagerService	startActivity moveTask grantURIPermission sendBroadcast registerBroadcastReceiver	moveTask grantURIPermission sendBroadcast registerBroadcastReceiver
AudioService	setStreamVolume setVibrateSetting	setVibrateSetting
PowerManagerService	acquireWakeLock isScreenOn reboot preventScreenOn	isScreenOn reboot preventScreenOn
SensorManager	getSensorList getDefaultSensor	getSensorList getDefaultSensor
LocationManagerService	requestLocationUpdates addProximityAlert getLastKnownLocation	requestLocationUpdates addProximityAlert getLastKnownLocation
SMSManager	copyMessageToIcc deleteMessageFromIcc sendTextMessage	copyMessageToIcc deleteMessageFromIcc sendTextMessage
TelephonyManager	getCellLocation getDeviceId getCellLocation	getCellLocation getDeviceId getCellLocation
ContentProvider USOMs		
ContactsProvider2	query insert update delete writeAccess readAccess	query insert update delete writeAccess readAccess
MMSSMSProvider	query insert update delete	query insert update delete
TelephonyProvider	query insert update delete	query insert update delete
SettingsProvider	query insert update delete	query insert update delete

Table 4: Exemplary System USOMs

Proactively Accountable Anonymous Messaging in Verdict

Henry Corrigan-Gibbs, David Isaac Wolinsky, and Bryan Ford
Yale University

Abstract

Among anonymity systems, DC-nets have long held attraction for their resistance to traffic analysis attacks, but practical implementations remain vulnerable to internal disruption or “jamming” attacks, which require time-consuming detection procedures to resolve. We present Verdict, the first practical anonymous group communication system built using *proactively verifiable* DC-nets: participants use public-key cryptography to construct DC-net ciphertexts, and use zero-knowledge proofs of knowledge to detect and exclude misbehavior *before* disruption. We compare three alternative constructions for verifiable DC-nets: one using bilinear maps and two based on simpler ElGamal encryption. While verifiable DC-nets incur higher computational overheads due to the public-key cryptography involved, our experiments suggest that Verdict is practical for anonymous group messaging or microblogging applications, supporting groups of 100 clients at 1 second per round or 1000 clients at 10 seconds per round. Furthermore, we show how existing symmetric-key DC-nets can “fall back” to a verifiable DC-net to quickly identify misbehavior, speeding up previous detections schemes by two orders of magnitude.

1 Introduction

A right to anonymity is fundamental to democratic culture, freedom of speech [3, 46], peaceful resistance to repression [39], and protecting minority rights [45]. Anonymizing relay tools, such as Tor [18], offer practical and scalable anonymous communication but are vulnerable to traffic analysis attacks [4, 34, 38] feasible for powerful adversaries, such as ISPs in authoritarian states.

Dining cryptographers networks [13] (*DC-nets*) promise security even against traffic analysis attacks, and recent systems such as Herbivore [24, 44] and Dissent [14, 52] have improved the scalability of DC-net-style systems. However, these systems are still vulnerable to internal *disruption* attacks in which a misbehaving member anonymously “jams” communication, either

completely or selectively. Dissent includes a *retrospective blame* procedure that can eventually exclude disruptors, but at high cost: tracing a disruptor in a 1,000-member group takes over 60 minutes [52], and the protocol makes no communication progress until it restarts “from scratch.” An adversary who infiltrates such a group with f colluding members can “sacrifice” them one at a time to disrupt *all* communication for f contiguous hours at any time—long enough time to cause a communications blackout before or during an important mass protest, for example.

Verdict, a novel but practical group anonymity system, thwarts such disruptions while maintaining DC-nets’ resistance to traffic analysis. At Verdict’s core lies a *verifiable* DC-net primitive, derived from theoretical work proposed and formalized by Golle and Juels [25], which requires participating nodes to prove *proactively* the well-formedness of messages they send. The first working system we are aware of to implement verifiable DC-nets, Verdict supports three alternative schemes for comparison: a pairing scheme using bilinear maps similar to the Golle-Juels approach, and two schemes based on ElGamal encryption in conventional integer or elliptic curve groups. Verdict incorporates this verifiable core into a client/server architecture like Dissent’s [52], to achieve scalability and robustness to client churn. As in Dissent, Verdict maintains security as long as *at least one* of the participating servers is honest, and participants need not know or guess which servers are honest.

Due to their reliance on public-key cryptography, verifiable DC-nets incur higher computation overheads than traditional DC-nets, which primarily use symmetric-key cryptography (e.g., AES). We expect this CPU cost to be acceptable in applications where messages are usually short (e.g., chat or microblogging), where costs are dominated by network delays, or in groups with relatively open or antagonistic membership where disruption risks may be high. Under realistic conditions, we find that Verdict can support groups of 100 users while maintain-

ing 1-second messaging latencies, or 1000-user groups with 10-second latencies. In a trace-driven evaluation of full-system performance for a microblogging application, Verdict is able to keep up with symmetric-key DC-nets in groups of up to 250 active users.

In contrast with the above “purist” approach, which uses expensive public-key cryptography to construct *all* DC-net ciphertexts, Verdict also implements and evaluates a *hybrid* approach that uses symmetric-key DC-nets for data communication when not under disruption attack, but leverages verifiable DC-nets to enable the system to respond much more quickly and inexpensively to disruption attacks. Dissent uses a *verifiable shuffle* [36] to broadcast an *accusation* anonymously; this shuffle dominates the cost of identifying disruptors. By replacing this verifiable shuffle with a verifiable DC-nets round, Verdict preserves the disruption-free performance of symmetric-key DC-nets, but reduces the time to identify a disruptor in a 1000-node group by two orders of magnitude, from 20 minutes to 26 seconds.

This paper’s primary contributions are:

- the first working implementation and experimental evaluation of verifiable DC-nets in a practical anonymous communication system,
- two novel verifiable DC-nets constructions using standard modular integer and elliptic curve groups, offering an order of magnitude lower computational cost than the original pairing approach [25],
- a hybrid system design that preserves performance of symmetric-key DC-nets, while reducing disruption resolution costs by two orders of magnitude, and
- experimental evidence suggesting that verifiable DC-nets may be practical for realistic applications, such as anonymous microblogging.

Section 2 introduces DC-nets and the disruption problem. Section 3 outlines Verdict’s architecture and adversary model, and Sections 4 and 5 describe its messaging protocol and cryptographic schemes. Section 6 presents application scenarios and evaluation results, Section 7 describes related work, and Section 8 concludes.

2 Background and Motivation

This section first introduces the basic DC-nets concept and known generalizations, then motivates the need for proactive accountability.

2.1 Anonymity with Strong Adversaries

To make the need for traffic-analysis-resistant anonymity systems more concrete, consider a political journalist who obtains some important secret government documents (e.g., the *Pentagon Papers*) from a confidential source. If the journalist publishes these documents under her own name, the journalist might risk prosecution

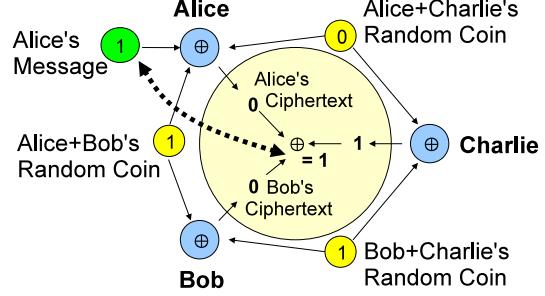


Figure 1: The basic DC-nets algorithm

or interrogation, and she might be pressured to reveal the source of the documents.

To reduce such risks, a number of political journalists could form a Verdict communication group. Any participating journalist may then *anonymously broadcast* the documents to the entire group of journalists, such that no member of the group can determine which journalist sent the documents. With Verdict, even if a government agency plants agents within the group of journalists and observes *all network traffic* during a protocol run, the agency remains unable to learn the source of the leak.

Existing systems such as Tor, which are practical and scalable but vulnerable to known traffic analysis attacks [16, 18, 32], cannot guarantee security in this context. For example, if a US journalist posts a leak to a US website, via a Tor connection whose entry and exit relays are in Europe, then an eavesdropper capable of monitoring transatlantic links [31] can de-anonymize the user via traffic analysis [18, 35]. Prior anonymity systems attempting to offer resistance to traffic analysis, discussed in Section 7, suffer from poor performance or vulnerability to active denial-of-service attacks.

2.2 DC-nets Overview

DC-nets [13] provide anonymous broadcast within a *group* of participants, who communicate lock-step in a series of *rounds*. In a given round, each group member contributes an equal length ciphertext that, when combined with all other members’ ciphertexts, reveals one or more cleartext messages. All group members know that each message was sent by *some* group member—but do not know *which* member sent each message.

In its simplest form, illustrated in Figure 1, we assume one group member wishes to broadcast a 1-bit message anonymously. To do so, every pair of members flips a coin, secretly agreeing on the random outcome of that coin flip. An N -member group thus flips $N(N - 1)/2$ coins in total, of which each member observes the outcome of $N - 1$ coins. Each member then XORs together the values of the $N - 1$ coins she observes, additionally the member who wishes to broadcast the 1-bit

message XORs in the value of that message, to produce that member’s DC-nets *ciphertext*. Each group member then broadcasts her 1-bit ciphertext to the other members. Finally, each member collects and XORs all N members’ ciphertexts together. Since the value of each shared coin is XORed into exactly two members’ ciphertexts, all the coins cancel out, leaving only the anonymous message, while provably revealing no information about which group member sent the message.

2.3 Practical Generalizations

As a standard generalization of DC-nets to communicate L -bit messages, all members in principle simply run L instances of the protocol in parallel. Each pair of members flips and agrees upon L shared coins, and each member XORs together the L -bit strings she observes with her optional L -bit anonymous message to produce L -bit ciphertexts, which XOR together to reveal the L -bit message. For efficiency, in practice each pair of group members forms a cryptographic shared secret—via Diffie-Hellman key agreement, for example—then group members use a cryptographic pseudo-random number generator (PRNG) to produce the L -bit strings.

As a complementary generalization, we can use any finite alphabet or group in place of coins or bits, as long as we have: (a) a suitable combining operator analogous to XOR, (b) a way to encode messages in the chosen alphabet, and (c) a way to generate complementary pairs of one-time pads in the alphabet that cancel under the chosen combining operator. For example, the alphabet might be 8-bit bytes, the combining operator might be addition modulo 256, and from each pairwise shared secret, one member of the pair generates bytes B_1, \dots, B_k from a PRNG, while the other member generates corresponding two’s complement bytes $-B_1, \dots, -B_k$.

2.4 Disruption and Verifiable DC-nets

A key weakness of DC-nets is that a single malicious insider can easily block all communication. An attacker who transmits arbitrary bits—instead of the XORed ciphertext that the protocol prescribes—unilaterally and *anonymously* jams all DC-net communication.

In many online venues such as blogs, chat rooms, and social networks, some users may have legitimate needs for strong anonymity—protest organizers residing in an authoritarian state, for example—while other antagonistic users (e.g., secret police infiltrators) may attempt to block communication if they cannot de-anonymize “unapproved” senders. Even in a system like Dissent that can *eventually* trace and exclude disruptors, an adversary with multiple colluding dishonest group members may still be able to slow or halt communication for long enough to ruin the service’s usability for honest participants. Further, if the group’s membership is open enough

to allow new disruptive members to join more quickly than the tracing process operates, then these infiltrators may be able to shut down communication permanently.

Verifiable DC-nets [25] leverage algebraic groups, such as elliptic curve groups, as the DC-nets alphabet. Using such groups allows for disruption resistance, by enabling members to *prove* the correctness of their ciphertexts’ construction without compromising the secrecy of the shared pseudo-random seeds. Using a hybrid approach that combines a traditional DC-net with a verifiable DC-net, Verdict can achieve the messaging latency of a traditional XOR-based DC-net while providing the strong disruption-resistance of verifiable DC-nets.

3 Verdict Architecture Overview

In this section, we describe the individual components of Verdict and how they combine to form the overall anonymous communication system.

3.1 Deployment and Adversary Model

Verdict builds on Dissent [51, 52] and uses the *multi-provider cloud* model illustrated in Figure 2 (a) to achieve scalability and resilience to ordinary node and link failures. In this model, a communication group consists of mostly unreliable *clients*, and a few *servers* we assume to be highly available and well-provisioned. Servers in a group should be administered independently—each furnished by a different *anonymity provider*, for example—to limit risk of all servers being compromised and colluding against the clients. The servers may be geographically or topologically close, however—possibly even hosted in the same data center, in locked cages physically and administratively accessible only to separate, independent authorities.

Clients directly communicate, at a minimum, with a single upstream server, while each server communicates with all other servers. This topology, shown in Figure 2 (b), reduces the communication and computation burden on the clients, and enables the system to make progress regardless of client churn. In particular, clients need not know which other clients are online at the time they submit their DC-net ciphertexts to their upstream server; clients only assume that all *servers* are online.

To ensure anonymity, clients need *not* assume that any particular server is trustworthy—a client need not even trust its immediately upstream server. Instead, **clients trust only that there exists at least one one honest server**, an assumption previously dubbed *anytrust* [51, 52], as a trust analog to anycast communication.

Verdict, like Dissent, achieves security under the anytrust assumption through the DC-nets key-sharing model shown in Figure 2 (c). Each client shares a secret with *every* server, rendering client ciphertexts indecipherable without the cooperation of *all* servers, and

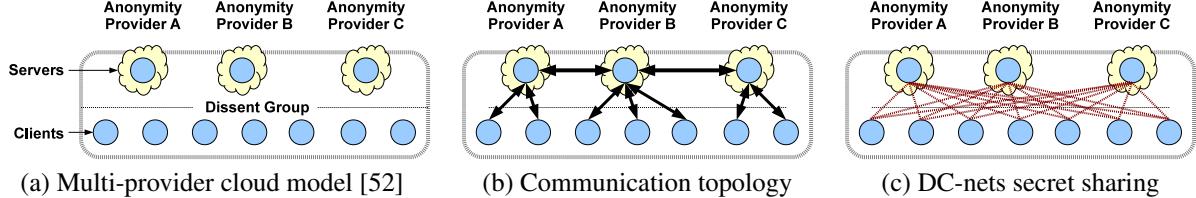


Figure 2: Verdict/Dissent deployment model, physical communication topology, and DC-nets secret sharing

hence protecting a client’s anonymity even if its immediately upstream server is malicious. Each client ultimately obtains an anonymity set consisting of the set of all honest clients, provided that the anytrust assumption holds, and provided the message contents themselves do not in some way reveal the sender’s identity.

A malicious server might refuse to service honest clients, but such refusal does not compromise clients’ anonymity—victims can simply switch to a different server. Although not yet supported in our Verdict prototype, Section 4.6 discusses how one might use threshold secret sharing to tolerate server failures, at the cost of requiring that we assume multiple servers are honest.

3.2 Security Goals

Verdict’s goal is to offer anonymity and disruption resistance in the face of a strong adversary who can potentially monitor all network links, modify packets as they traverse the network, and compromise a potentially large fraction of a group’s participating members. We say that a participant is *honest* if it follows the protocol exactly and does not collude with or leak secret information to other nodes. A participant is *dishonest* otherwise. Dishonest nodes can exhibit *Byzantine behavior*—they can be arbitrarily incorrect and can even just “go silent.”

The system is designed to provide anonymity among the set of *honest* participants, who remain online and uncompromised throughout an interaction period, and who do not compromise their identity via the content of the messages they send. We define this set of honest and online participants as the *anonymity set* for a protocol run. If a group contains many colluding dishonest participants, Verdict can anonymize the honest participants only among the remaining subset of *honest* members: in the worst case of a group containing only one honest member, for example, Verdict operates but can offer that member no meaningful anonymity.

Similarly, Verdict does not prevent long-term intersection attacks [28] against otherwise-honest participants who repeatedly come and go during an interaction period, leaking information to an adversary who can correlate online status with linkable anonymous posts. Reasoning about anonymity sets generally requires making inherently untestable assumptions about *how many* group

members may be dishonest or unreliable, but Verdict at least does not assume that the honest participants know *which* other participants are honest and reliable.

Finally, Verdict’s disruption-resistant design addresses *internal* disruption attacks by misbehaving anonymous participants, a problem specific to anonymous communication tools and particularly DC-nets. Like any distributed system, Verdict may be vulnerable to more general network-level Denial-of-Service (DoS) attacks as well, particularly against the servers that are critical to the system’s availability and performance. Such attacks are important in practice, but not specific to anonymous communication systems. This paper thus does not address general DoS attacks since well-known defenses apply, such as server provisioning, selective traffic blocking, and proof-of-life or proof-of-work challenges.

4 Protocol Design

Verdict consists of two major components: the messaging protocol, and the cryptographic primitive clients and servers use to construct their ciphertexts. This section describes the Verdict messaging protocols, and the following section describes the cryptographic constructions.

4.1 Core Verdict Protocol

Figure 3 summarizes the steps comprising a normal-case run of the Verdict protocol. This protocol represents a direct adaptation of the DC-nets scheme (Section 2.2) to the two-level communication topology illustrated in Figure 2 (b), and to the client/server secret-sharing graph in Figure 2 (c). As in Dissent, Verdict’s anonymity guarantee relies on Chaum’s original security analysis [13], in which an honest node’s anonymity set consists of the set of honest nodes that remain connected in the secret-sharing graph after removing links to dishonest nodes. Since each client shares a secret with every server, and we assume that there exists at least one honest server, this honest server forms a “hub” connecting all honest nodes. This anonymity property holds regardless of physical communication topology, which is why the clients need not trust their immediately upstream server.

The ciphertext- and proof-generation processes assume that communication in the DC-net is broken up into *time slots* (akin to TDMA), such that only one client—

1. **Client Ciphertext Generation.** Each client i generates a client ciphertext, and submits this ciphertext to client i 's upstream server. If client i is the anonymous owner of the current slot, the client computes and submits a slot owner ciphertext using her pseudonym secret key and her plaintext message m .
 2. **Client Set Sharing.** After receiving valid client ciphertexts from its currently connected downstream clients, each server j broadcasts to all servers its set \mathbf{C}_j of collected client ciphertexts.
 3. **Server Ciphertext Generation.** After receiving client ciphertext sets from all servers, each server j computes $\mathbf{C} = \bigcup_k \mathbf{C}_k$, the set of client ciphertexts collected by *all* servers. Server j then uses \mathbf{C} to compute a server ciphertext corresponding to the set of submitted client ciphertexts. Server j broadcasts this server ciphertext to all other servers.
 4. **Plaintext Reveal.** After receiving a server ciphertext from every other server, each server j combines the $|\mathbf{C}|$ client ciphertexts and M server ciphertexts to reveal the plaintext message m . Server j signs m and broadcasts its signature σ_j to all servers.
 5. **Plaintext Sharing.** After receiving valid signatures from all servers, server j sends the plaintext message m and the M signatures $\sigma_1, \dots, \sigma_M$ (one from each server) to its downstream clients.
 6. **Client Verification.** Upon receiving the plaintext m and M valid signatures from its upstream server, client i accepts the plaintext message and considers the messaging round to have completed successfully.
- All messages sent over the network include a session nonce and are signed with the sender's long-term well-known (non-anonymous) signing key.*

Figure 3: Core Verdict messaging protocol

the slot's *owner*—is allowed to send an anonymous message in each time slot. The owner of a slot is the client who holds the private key corresponding to a *pseudonym public key* assigned to the slot. To maintain the slot owner's anonymity, *no one must know* which client owns which transmission slot. Section 4.3 below describes the assignment of pseudonym keys to transmission slots.

Figure 4 shows an example DC-net transmission schedule with three slots, owned by pseudonyms A, B, and C. Each slot owner can transmit one message per *messaging round*, and the slot ordering in the schedule remains the same for the duration of the Verdict *session*.

4.2 Verifiable Ciphertexts in Verdict

While Verdict's anonymity derives from the same principles as Dissent's, the key difference is in the “alpha-

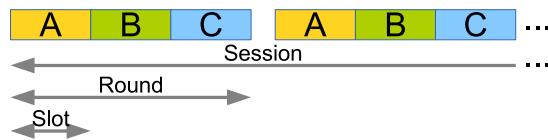


Figure 4: Example DC-net transmission schedule, where anonymous authors A, B, and C transmit in each round.

bet” with which Verdict generates DC-net ciphertexts, and in the way Verdict combines and checks those ciphertexts. Dissent uses a symmetric-key cryptographic pseudo-random number generators (PRNG) to generate shared secrets, and uses bitwise XOR to combine them and later to reveal the plaintext message. While efficient, the symmetric-key approach offers no way to check that any node's ciphertext was generated correctly until the final cleartext messages are revealed. If any node corrupts a protocol round by sending an incorrect ciphertext, Dissent can eventually identify that node only via a complex retroactive *blame* procedure.

Verdict, in contrast, divides messages into chunks small enough to be encoded into elements of algebraic groups, such as Schnorr [42] or elliptic curve groups, to which known proof-of-knowledge techniques apply. Section 5 later outlines three particular ciphertext generation schemes that Verdict implements, although Verdict's architecture and protocol design is agnostic to the specific scheme. These schemes may be considered analogous to “pluggable” ciphersuites in SSL/TLS.

Thus, any Verdict ciphertext is generated *on behalf* of the holder of some particular pseudonym keypair. While the details of a ciphertext correctness proof depend on the particular scheme, all such proofs have the general form of an “either/or” knowledge proof, of the type systematized by Camenisch and Stadler [11]. In particular, a ciphertext correctness proof attests that either:

- the ciphertext is an encryption of *any* message, and the producer of the ciphertext holds the *private* part of the pseudonym key for this slot, OR
- the ciphertext is an encryption of a *null* cover message, which, when combined with other cover ciphertexts and exactly one actual encrypted message ciphertext, will combine to reveal the encrypted message.

Only the pseudonym key owner can produce a correctness proof for an arbitrary message following the first alternative above, while any node can generate an “honest” cover ciphertext—and the proof by construction reveals no information about *which* alternative the proof generator followed. We leave discussion of further details of this process to Section 5, but merely note that such “either/or” proofs are pervasive and well-understood in the cryptographic theory community.

In Verdict, each client computes and attaches a cryptographic correctness proof to each ciphertext it sends to its upstream server, and each server in turn attaches a correctness proof to the server-side ciphertext it generates in Phase 3 of each round (Figure 3). In principle, therefore, each server can immediately verify the correctness of any client’s or other server’s ciphertext it receives, *before* “accepting” it and combining it with the other ciphertexts for that protocol round. As in Dissent, Verdict achieves resilience to client churn by (optionally) requiring clients to submit their ciphertexts before a certain “deadline” in each messaging round. We describe this technique in Section 4.5.

While Verdict nodes can *in principle* verify the correctness of any received ciphertext immediately, actually doing so has performance costs. These costs lead to design tradeoffs between “eager” and “lazy” verification, both of which we implement and evaluate later in Section 6. Lazy verification enables the critical servers to avoid significant computation costs during rounds that are not disrupted, at the expense of making a round’s output unusable if it *is* disrupted. Even if a lazily-verified round is disrupted, however, the fact that Verdict nodes always generate and transmit signed ciphertext correctness proofs greatly simplifies and shortens the retroactive blame process with respect to Dissent.

Verdict currently treats *server-side* failures of all types, including invalid server ciphertexts, as “major events” requiring administrative action, and simply halts the protocol with an alert until such action is taken. Section 4.6 later discusses approaches to make Verdict resilient against server-side failures, but we leave implementing and evaluating such mechanisms to future work. Such server-side failures affect only availability, however; anonymity remains protected as long as at least one (not necessarily online) server remains uncompromised.

4.3 Scheduling Pseudonym Keys

To assign ownership of transmission slots to clients in such a way that *no one* knows which client owns which slot, Verdict applies an architectural idea from Dissent [52]. At the start of a Verdict session, each of the N clients secretly submits a slot owner pseudonym key to a verifiable shuffle protocol [36] run by the servers. The public output of the shuffle is the N pseudonym keys in permuted order—such that *no one* knows which node submitted which pseudonym key other than their own. Verdict participants then use each of these N pseudonym keys to initialize N concurrent instances of the core Verdict DC-net with each instance becoming a slot in a verifiable DC-net *transmission schedule*.

Scheduling Policy Not every client will necessarily want to transmit an anonymous message in every messaging round. In addition, clients may want to transmit

messages of different lengths. To make Verdict more efficient in these cases, Verdict allows clients to request a change in the length of their messaging slot (e.g., so that a client can send a long message in a single messaging round) and to temporarily “close” their transmission slot (if a client does not expect to send a message for several rounds). Clients issue these requests by prepending a few bits of control data to the anonymous message they send in their transmission slot.

4.4 Hybrid XOR/Verifiable DC-Nets

While the verifiable DC-net design above may be needed in scenarios in which disruptions are frequent, the public-key cryptography involved imposes a much higher computational cost than traditional XOR-based DC-nets. To offer better performance in groups with fewer or less frequent disruptions, Verdict has a “hybrid” mode of operation that uses the fast XOR-based DC-net when there are no active disruptors in the group, and reverts to a verifiable DC-net in the presence of an active disruptor. This hybrid Verdict DC-net marries the relatively low computational cost of the XOR-based DC-net with the low accountability cost of the verifiable DC-net.

To set up a hybrid DC-net session, all protocol participants first participate in a pseudonym signing key shuffle, as described above in Section 4.3. At the conclusion of the shuffle, all nodes initialize *two* DC-net slots for *each* of the N clients: one traditional Dissent-style DC-net, and one verifiable Verdict DC-net.

When the group is not being disrupted, clients transmit in their Dissent DC-net slot, allowing nodes to take advantage of the speed of Dissent’s XOR-based DC-net. When nodes detect the corruption of a message in the Dissent DC-net, the client whose message was corrupted reverts to transmitting in its *verifiable* DC-net slot. This client can use the verifiable slot to transmit anonymously the “accusation” necessary to identify the disruptor in the Dissent accusation process [52, Section 3.9]. The Verdict DC-net replaces the expensive verifiable shuffle necessary for nodes to exchange the accusation information in Dissent. In this way, Verdict offers the normal-case efficiency of XOR-based DC-nets while greatly reducing the cost of tracing and excluding disruptors.

4.5 Client Churn

In realistic deployments clients may go offline at any time, and this problem becomes severe in large groups of unreliable clients exhibiting constant churn. To prevent slow or unresponsive clients from disrupting communication, the servers need not wait in Phase 2 for all downstream clients to submit ciphertexts. Instead, servers can wait for a fixed threshold of $t \leq N$ clients to submit ciphertexts, or for some fixed time interval τ . Servers might also use some more complicated *window*

closure policy, as in Dissent [52]: e.g., wait for a threshold of clients and then an additional time period before proceeding. The participants must agree on a window closure policy before the protocol run begins.

There is an inherent tradeoff between anonymity and the system’s ability to cope with unresponsive clients. If the servers close the ciphertext submission window too aggressively, honest but slow clients might be unable to submit their ciphertexts in time, reducing the anonymity of clients who do manage to submit messages. In contrast, if the servers wait until every client has submitted a ciphertext, a single faulty client could prevent protocol progress indefinitely. Optimal policy choices depend on the security requirements of the application at hand.

4.6 Limitations and Future Enhancements

This section outlines some of Verdict’s current limitations, deployment issues, and areas for future work.

Group Evolution Verdict’s architecture assumes that, at the start of the protocol, group members agree to a “roster” of protocol participants—essentially a list of public keys defining the group’s membership. The current prototype simplistically assumes that this group roster is a static list, and that the session nonce is a hash of a file containing this roster and other group policy information. This design trivially ensures that all nodes participating in a given group (uniquely identified by its session nonce) agree upon the same roster and policy. Changing the group roster or policy in the current prototype requires forming a new group, but we are exploring approaches to group management which would allow for on-the-fly membership changes. For now, we simply note that Verdict’s security properties critically depend on group membership policy decisions, which affect how quickly adversarial participants can infiltrate a group. We consider group management policy to be orthogonal to this paper’s communication mechanisms.

Sybil Attacks If it is too easy to join a group, dishonest participants might flood the group with Sybil identities [19], giving an anonymous slot owner the impression that she has more anonymity than she actually does. The current “static group” design shifts the Sybil attack prevention problem to whomever formulates the group roster. Dynamic group management schemes could leverage existing Sybil prevention techniques [47, 53, 54], but we do not consider such approaches herein.

Membership Concealment Verdict considers the group roster, containing the public keys of all participants, to be public information: concealing participation in the protocol is an orthogonal security goal that Verdict currently does not address. We are exploring the use of anonymous authentication techniques [22, 29, 41] to enable Verdict clients to “sign on” and prove member-

ship in the group without revealing to the Verdict servers (or to the adversary) *which* specific group members are online at any given time. Further, we expect that Verdict’s design could be composed with other techniques to achieve membership concealment [33, 49], but we leave such enhancements to future work.

Unresponsive Servers Verdict currently assumes that the servers supporting a group are well-provisioned and highly reliable, and the system simply ceases communication progress in the face of any server’s failure. Any fault-masking mechanism would be problematic, in fact, in the face of Verdict’s assumption that only one server might be honest: if that one honest server goes offline and the protocol continues without it, then the remaining dishonest servers could collude against all honest users.

If we assume that there are $h > 1$ honest servers, however, a currently unimplemented variation of Verdict could allow progress if as many as $h - 1$ servers are unresponsive. Before the protocol run, every server uses a *publicly verifiable secret sharing scheme* [43], to distribute shares of its per-session secret key. The scheme is configured such that any quorum of $M - h + 1$ shares is sufficient to reconstruct the secret. Thus, at least one honest server must remain online and contribute a share for a secret to be reconstructed. (Golle and Juels [25] also use a secret-sharing scheme, but they rely on a trusted third-party to generate and distribute the shares.)

If a server becomes unresponsive, the remaining online servers can broadcast their shares of the unresponsive server’s secret key. Once a quorum of servers broadcasts these shares, the remaining online servers will be able to reconstruct the unresponsive server’s private key. Thereafter, each server can simulate the unresponsive server’s messages for the rest of the protocol session.

Blame Recovery The current Verdict prototype can detect server misbehavior, but it does not yet have a mechanism by which the remaining servers can collectively form a new group “roster” with the misbehaving nodes removed. We expect off-the-shelf Byzantine Fault Tolerance algorithms [12] to be applicable to this *group evolution* problem. Using BFT to achieve agreement, however, requires a stronger security assumption: in a group with f dishonest servers, there must be at least $3f + 1$ total servers. We sketch a possible BFT-based group evolution approach here.

The BFT cluster’s shared state in this case is the group “roster,” containing the session nonce and a list of all active Verdict clients and servers, identified by their public keys. The two operations in this BFT system are:

- **EVOLVE_GROUP**(nonce, node_index, proof), a request to remove a dishonest node (identified by node_index) from the group roster. BFT servers

- remove the dishonest node from the group if the proof is valid, yielding the new group roster.
- `GET_GROUP()`, which returns current the group roster.
- If, at some point during the Verdict session, a Verdict node concludes that the protocol has failed due to the dishonesty of node d , this honest node makes an `EVOLVE_GROUP` request to the BFT cluster and waits for a response. The honest BFT servers will agree on a new group roster with the dishonest node d removed and the Verdict servers will begin a new instance of the Verdict protocol with the new group roster. Clients use `GET_GROUP` to learn the new group roster.

5 Verifiable DC-net Constructions

The Verdict architecture relies on a verifiable DC-net primitive that has many possible implementations. In this section, we first describe the general interface that each of the cryptographic constructions must implement—which could be described as a “Verdict ciphersuite API”—then we describe the three specific experimental schemes that Verdict currently implements.

All three schemes are founded on standard, well-understood cryptographic techniques that have been formally developed and rigorously analyzed in prior work. As with most practical, complex distributed systems with many components, however, we cannot realistically offer a rigorous proof that these cryptographic tools “fit together” correctly to form a perfectly secure overall system. (This is true even of SSL/TLS and its ciphersuites, which have received far more cryptographic scrutiny than Verdict but in which flaws are still found regularly.) We also make no claim that any of the current schemes are “the right” ones or achieve any particular ideal; we merely offer them as contrasting points in a large design space. To lend some informal credibility to their security, we note that our pairing-based scheme is closely modeled on the verifiable DC-nets scheme that Golle and Juels already developed formally [25], and the extended version of this paper [15] sketches a security argument for the third and most computationally efficient scheme.

5.1 Verifiable DC-net Primitive API

The core cryptographic primitive consists of a set of six methods. Each of these six methods takes a list of protocol session parameters (e.g., group roster, session nonce, slot owner’s public key) as an implicit argument:

- *Cover Create*: Given a client session secret key, return a valid client ciphertext representing “cover traffic,” which do not contain actual messages.
- *Owner Create*: Given a client session secret key, the slot owner’s pseudonym secret key, and a plaintext message m to be transmitted anonymously, return a valid *owner* ciphertext that encodes message m .

- *Client Verify*: Given a client public key and a client ciphertext, return a boolean flag indicating whether the client ciphertext is valid.

- *Server Create*: Given a server private key and a set of client ciphertexts, return a valid server ciphertext.
- *Server Verify*: Given a server public key, a set of valid client ciphertexts, and a server ciphertext, return a flag indicating whether the server ciphertext is valid.
- *Reveal*: Combine N client ciphertexts and M server ciphertexts, returning the plaintext message m .

However these methods are implemented, they must obey the following security properties, stated informally:

- **Completeness**: An honest verifier always accepts a ciphertext generated by an honest client or server.
- **Soundness**: With overwhelming probability an honest verifier rejects an incorrect ciphertext, such as an owner ciphertext formed without knowledge of the owner’s pseudonym secret key.
- **Zero-knowledge**: A verifier learns nothing about a ciphertext besides the fact that it is correctly formed.
- **Integrity**: Combining N valid client ciphertexts, including one ciphertext from the anonymous slot owner, and M valid server ciphertexts, reveals the slot owner’s plaintext message.
- **Anonymity**: A verifier cannot distinguish a client ciphertext from the anonymous slot owner’s ciphertext. The extended version of this paper [15] offers a game-based definition of anonymity.

In practice, each of our current implementations of this verifiable DC-nets primitive achieve these security properties in the random-oracle model [5] using non-interactive zero-knowledge proofs [26].

5.2 ElGamal-Style Construction

The first scheme builds on the ElGamal public-key cryptosystem [20]. In ElGamal, a public/private keypair has the form $\langle B, b \rangle = \langle g^b, b \rangle$,¹ and plaintexts and ciphertexts are elements of an algebraic group G .² We refer to this as the “ElGamal-style” construction because its use of an ephemeral public key and encryption by multiplication structurally resembles the ElGamal cryptosystem. Our construction does *not* exhibit the malleability of textbook ElGamal encryption, however, because a proof of knowledge of the secret ephemeral public key is attached to every ciphertext element.

Client Ciphertext Construction Given a list of server public keys $\langle B_1, \dots, B_M \rangle$, a client constructs a ciphertext

¹ We do not require that a trusted third party generate participants’ keypairs, but we *do* require participants to prove knowledge of their secret key at the start of a protocol session, for reasons described in the extended version of this paper [15].

² Throughout, unless otherwise noted, group elements are members of a finite cyclic group G in which the Decision Diffie-Hellman (DDH) problem [6] is assumed computationally infeasible, and g is a public generator of G .

by selecting an ephemeral public key $R_i = g^{r_i}$ at random and computing the ciphertext element:

$$C_i = m (\prod_{j=1}^M B_j)^{r_i}$$

If the client is the slot owner, the client sets m to its secret message, otherwise the client sets $m = 1$.

To satisfy the security properties described in Section 5.1, the client must somehow *prove* that the ciphertext tuple $\langle R_i, C_i \rangle$ was generated correctly. We adopt the technique of Golle and Juels [25] and use a non-interactive proof-of-knowledge of discrete logarithms [11] to prove that the ciphertext has the correct form. If the slot owner’s pseudonym public key is Y , the client’s ephemeral public key is R_i , and the client’s ciphertext element is C_i , the client generates a proof:

$$\text{PoK}\{r_i, y : (R_i = g^{r_i} \wedge C_i = (\prod_{j=1}^M B_j)^{r_i}) \vee Y = g^y\}$$

In words: the sender demonstrates that *either* it knows the discrete logarithm r_i of the ephemeral public key R_i , and the ciphertext is the product of all server public keys raised to the exponent r_i ; *or* the sender knows the slot owner’s secret pseudonym key y , in which case the slot owner can set C_i to a value of her choosing. The extended version of this paper [15] details how to construct and verify this type of non-interactive zero-knowledge proof.

Note that a dishonest slot owner can set C_i to a maliciously constructed value (e.g., $C_i = 1$). The only effect of such an “attack” is that the slot owner compromises *her own* anonymity. Since a dishonest slot owner can always compromise her own anonymity (e.g., by publishing her secret keys), a dishonest slot owner achieves nothing by setting C_i maliciously.

The tuple $\langle R_i, C_i, \text{PoK} \rangle$ serves as the client’s ciphertext. As explained in Section 4.1, all participants sign the messages they exchange for accountability.

Server Ciphertext Construction Given a server public key $B_j = g^{b_j}$ and a list of ephemeral client public keys $\langle R_1, \dots, R_N \rangle$, server j generates its server ciphertext as:

$$S_j = (\prod_{i=1}^N R_i)^{-b_j}$$

The server proves the validity of its ciphertext by creating a non-interactive proof of knowledge that it knows its secret private key b_j and that its ciphertext element S_j is the product of the ephemeral client keys raised to the exponent $-b_j$:

$$\text{PoK}\{b_j : B_j = g^{b_j} \wedge S_j = (\prod_{i=1}^N R_i)^{-b_j}\}$$

Message Reveal To reveal the plaintext message, a participant computes the product of N client ciphertext elements and M server ciphertext elements:

$$m = (\prod_{i=1}^N C_i) (\prod_{j=1}^M S_j)$$

Each factor $g^{r_i b_j}$, where r_i is client i ’s ephemeral secret key and b_j is server j ’s secret key, is included exactly

twice in the above product—once with a positive sign in the client ciphertexts and once with a negative sign in the server ciphertexts. These values therefore cancel, leaving only the plaintext m .

Drawbacks Since the clients must use a new ephemeral public key for *each* ciphertext element, sending a plaintext message that is L group elements in length requires each client to generate and transmit L ephemeral public keys. The proof of knowledge for this construction is $L + O(1)$ group elements long, so a message of L group elements expands to $3L + O(1)$ elements.

5.3 Pairing-Based Construction

A major drawback of the ElGamal construction is that, due to the need for ephemeral keys, every ciphertext is three times as long as the plaintext it encodes. Golle and Juels [25] use bilinear maps to eliminate the need for ephemeral keys. Our pairing-based construction adopts elements of their technique, while avoiding their reliance on a trusted third party, a secret-sharing scheme, and a probabilistic transmission scheduling algorithm.

A symmetric bilinear map \hat{e} maps two elements of a group G_1 into a target group G_2 — $\hat{e} : G_1 \times G_1 \rightarrow G_2$. A bilinear map has the property that: $\hat{e}(aP, bQ) = \hat{e}(P, Q)^{ab}$.³ To be useful, the map must also be non-degenerate (if P is a generator of G_1 , $\hat{e}(P, P)$ is a generator of G_2) and efficiently computable [8]. We assume that the decision bilinear Diffie-Hellman assumption [7] holds in G_1 .⁴

Since pairing allows a *single* pair of public keys to generate a *sequence* of shared secrets, clients need not generate ephemeral public keys for each ciphertext element they send. This optimization leads to shorter ciphertexts and shorter correctness proofs.

Client Ciphertext Construction For a set of server public keys $\langle B_1, \dots, B_M \rangle$, a public nonce $\tau \in G_1$ computed using a hash function, and a client public key $A_i = g^{a_i}$, a pairing-based client ciphertext has the form:

$$C_i = m \hat{e}(\prod_{j=1}^M B_j, \tau)^{a_i}$$

As before, if the client is not the slot owner, the client sets $m = 1$. Each client can produce a proof of the correctness of its ciphertext by executing a proof of knowledge similar to one used in the ElGamal-style construction above:

$$\text{PoK}\{a_i, y : (A_i = g^{a_i} \wedge C_i = \hat{e}(\prod_{j=1}^M B_j, \tau)^{a_i}) \vee Y = g^y\}$$

While the ElGamal-style scheme requires $3L + O(1)$ group elements to encode L elements of plaintext, a

³ Since G_1 is usually an elliptic curve group, the generator of G_1 is written as P (an elliptic curve point) and the repeated group operation is written as aP instead of g^a . We will use the latter notation for consistency with the rest of this section.

⁴ Note that the decision Diffie-Hellman problem is easy in G_1 , since given $g, g^a, g^b, g^c \in G_1$, a DDH tuple will always satisfy $\hat{e}(g^a, g^b) = \hat{e}(g, g^c)$ if $c = ab \bmod q$.

pairing-based ciphertext requires only $L + O(1)$ group elements to encode an L -element plaintext.

Server Ciphertext Construction Using a server public key $B_j = g^{b_j}$, a public round nonce τ , and client public keys $\langle A_1, \dots, A_N \rangle$, a server ciphertext has the form:

$$S_j = \hat{e}(\Pi_{i=1}^N A_i, \tau)^{-b_j}$$

The server proof of correctness is then:

$$\text{PoK}\{b_j : B_j = g^{b_j} \wedge S_j = \hat{e}(\Pi_{i=1}^N A_i, \tau)^{-b_j}\}$$

Message Reveal To reveal the plaintext, the servers take the product of all client and server ciphertexts:

$$m = (\Pi_{i=1}^N C_i)(\Pi_{j=1}^M S_j)$$

Drawbacks The main downside of this construction is the relatively high computational cost of the pairing operation. Computing the pairing operation on two elements of G_1 can take an order of magnitude longer than a normal elliptic curve point addition in a group of similar security level, as Section 6.2 below will show.

5.4 Hashing-Generator Construction

Our hashing-generator construction pursues a “best of both worlds” combination of the ElGamal-style and pairing-based constructions. This construction has short ciphertexts, like the pairing-based construction, but avoids the computational cost of the pairing-based scheme by using conventional integer or elliptic curve groups. To achieve both of these desired properties, the hashing-generator construction adds some protocol complexity, in the form of a session set-up phase.

Set-up Phase In the set-up phase, each client i establishes a Diffie-Hellman shared secret r_{ij} with every server j using their respective public keys g^{a_i} and g^{b_j} by computing $r_{ij} = \text{KDF}(g^{a_i b_j})$ using a key derivation function KDF. Clients publish commitments to these shared secrets $R_{ij} = \hat{g}^{r_{ij}}$ using another public generator \hat{g} .

The hashing-generator construction requires a process by which participants compute a sequence of generators g_1, \dots, g_L of the group G , such that no participant knows the discrete logarithm of any of these generators with respect to any other generator. In other words, *no one* knows an x such that $g_i^x = g_j$, for any i, j pair. In practice, participants compute this sequence of generators by hashing a series of strings, (e.g., the round nonce concatenated with “1”, “2”, “3”, …), to choose the set of generating group elements.

At the end of the set-up phase, every client i can produce a *sequence* of shared secrets with each server j using their shared secret r_{ij} and the L generators: $g_1^{r_{ij}}, \dots, g_L^{r_{ij}}$. In the ℓ th message exchange round, all participants use generator g_ℓ as their common generator.

Client Ciphertext Construction To use the hashing-generator scheme to create a ciphertext, the client uses its shared secrets r_{i1}, \dots, r_{iM} with the servers, and generator g_ℓ for the given protocol round to produce a ciphertext:

$$C_i = mg_\ell^{(\sum_{j=1}^M r_{ij})}$$

As before, $m = 1$ if the sender is not the slot owner.

To prove the validity of a ciphertext element, the client executes the following proof of knowledge, where Y is the slot owner’s pseudonym public key, $r_i = \sum_{j=1}^M r_{ij}$, and R_{ij} is the commitment to the secret shared between client i and server j :

$$\text{PoK}\{r_i, y : ((\Pi_{j=1}^M R_{ij}) = \hat{g}^{r_i} \wedge C_i = g_\ell^{r_i}) \vee Y = g^y\}$$

Server Ciphertext Construction Server j ’s ciphertext for the ℓ th message exchange round is similar to the client ciphertext, except with negated exponents:

$$S_j = g_\ell^{(-\sum_{i=1}^N r_{ij})}$$

The server proves correctness of a ciphertext by executing a proof of knowledge, where $r_j = \sum_{i=1}^N r_{ij}$:

$$\text{PoK}\{r_j : (\Pi_{i=1}^N R_{ij}) = \hat{g}^{r_j} \wedge S_j = g_\ell^{-r_j}\}$$

Message Reveal The product of the client and server ciphertexts reveals the slot owner’s plaintext message m :

$$m = (\Pi_{i=1}^N C_i)(\Pi_{j=1}^M S_j)$$

Failed Session Set-up A dishonest client i might try to disrupt the protocol by publishing a corrupted commitment R'_{ij} that disagrees with server j ’s commitment R_{ij} to the shared secret $r_{ij} = \text{KDF}(g^{a_i b_j})$. If the commitments disagree, the honest server can prove its innocence by broadcasting the Diffie-Hellman secret $\rho_{ij} = g^{a_i b_j}$ along with a proof that it correctly computed the Diffie-Hellman secret using its public key B_j and the client’s public key A_i .

$$\text{PoK}\{b_j : \rho_{ij} = A_i^{b_j} \wedge B_j = g^{b_j}\}$$

If the server is dishonest, the client can produce a similar proof of innocence. Any user can verify this proof, and then use $g^{a_i b_j}$ to recreate the correct commitment R_{ij} . Once the verifier has the correct commitment R_{ij} , the verifier can confirm either that the client in question published an invalid commitment or that the server in question dishonestly accused the client.

Since the session set-up between client i and server j will only fail if either i or j is dishonest, there is no security risk to publishing the shared secret $g^{a_i b_j}$ after a failed set-up—the dishonest client (or server) could have shared this secret with the adversary anyway.

Long Messages The client and server ciphertext constructions described above allow the slot owner to transmit a plaintext message m that is at most one group element in length in each run of the protocol. To encode

longer plaintexts efficiently, participants use a modified proof-of-knowledge construction that proves the validity of L ciphertext elements ($C_{i,1}$ through $C_{i,L}$) at once:

$$\text{PoK}\{r_i, y : ((\prod_{j=1}^M R_{ij} = \hat{g}^{r_i}) \wedge (\wedge_{\ell=1}^L C_{i,\ell} = g_\ell^{r_i})) \vee Y = g^y\}$$

Servers can use a similarly modified proof of knowledge. This modified knowledge proof is surprisingly compact: the length of the proof is *constant* in L , since the length of the proof is linear in the number of proof variables (here, the only variables are r_i and y). The total length of the tuple (\vec{C}_i, PoK) using this proof is $L + O(1)$.

Lazy Proof Verification In the basic protocol, every server verifies the validity proof on every client ciphertext in every protocol round. To avoid these expensive verification operations, servers can use *lazy proof verification*: servers check the validity of the client proofs only if they detect, at the end of a protocol run, that the anonymous slot owner’s message was corrupted. For reasons discussed in the extended version of this paper [15], lazy proof verification is possible only using the pairing-based or hashing-generator ciphertext constructions.

Security Analysis Since the hashing-generator scheme is the most performant variant, we sketch an informal security proof for the hashing-generator proof construction in the extended version of this paper [15].

6 Evaluation

This section describes our Verdict prototype implementation and summarizes the results of our evaluations.

6.1 Implementation

We implemented the Verdict protocol in C++ using the Qt framework as an extension to the existing Dissent prototype [52]. Our implementation uses OpenSSL 1.0.1 for standard elliptic curve groups, Crypto++ 5.6.1 for big integer groups, and the Stanford Pairing-Based Cryptography (PBC) 0.5.12 library for pairings [48]. Unless otherwise noted, the evaluations use 1024-bit integer groups, the 256-bit NIST P-256 elliptic curve group [37], and a pairing group in which G_1 is an elliptic curve over a 512-bit field (using PBC’s “Type A” parameters) [30]. We collected the macrobenchmark and end-to-end evaluation results on the DeterLab [17] testbed.

The source code for our implementation is available at <https://github.com/DeDiS/Dissent>.

6.2 Microbenchmarks

To compare the pure computational costs of the different DC-net schemes, Figure 5 shows ciphertext generation and verification throughput measured at a variety of block sizes, running on a workstation with a 3.2 GHz Intel Xeon W3565 processor. These experiments involve no network activity, and are single-threaded, thus they do not reflect any speedup that parallelization might offer.

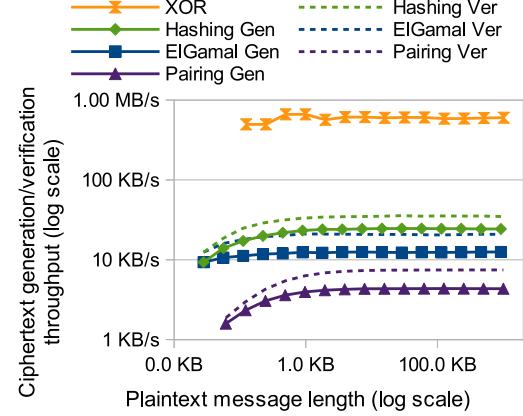


Figure 5: Ciphertext generation and verification throughput for the three verifiable DC-net variants and the XOR-based scheme.

The hashing-generator construction, which is the fastest scheme tested, encrypts 20 KB of client plaintext per second. The slowest, paring-based construction encrypts around 3 KB per second. The fastest verifiable scheme is still over an order of magnitude slower than the traditional (unverifiable) XOR-based scheme, which encrypts 600 KB of plaintext per second. The hashing-generator scheme performs best because it needs no pairing operations and requires fewer group exponentiations than the ElGamal construction.

Figure 5 shows that ciphertext verification is slightly faster than ciphertext generation. This is because generating the ciphertext and zero-knowledge proof requires more group exponentiations than proof verification does.

The three constructions also vary in the size of ciphertexts they generate (Figure 6). While the pairing-based scheme and the hashing-generator schemes encrypt length L plaintexts as ciphertexts of length $L + O(1)$, the ElGamal-style scheme encrypts length L plaintexts as length $3L + O(1)$ ciphertexts. As discussed in Section 5.2, for every plaintext message element encrypted, ElGamal-style ciphertexts must include an ephemeral public key and an additional proof-of-knowledge group element. Since the hashing-generator scheme is the fastest and avoids the ElGamal scheme’s ciphertext expansion, subsequent experiments use the hashing-generator scheme unless otherwise noted.

6.3 Accountability Cost

Figure 7 presents three graphs: (a) the time it takes to set up a transmission schedule via a verifiable shuffle, prior to DC-net communication, (b) the time required to execute a single DC-net protocol round in each scheme, and (c) the time required to identify a disruptor. The graphs compare four protocol variants: Dissent, Verdict, Verdict

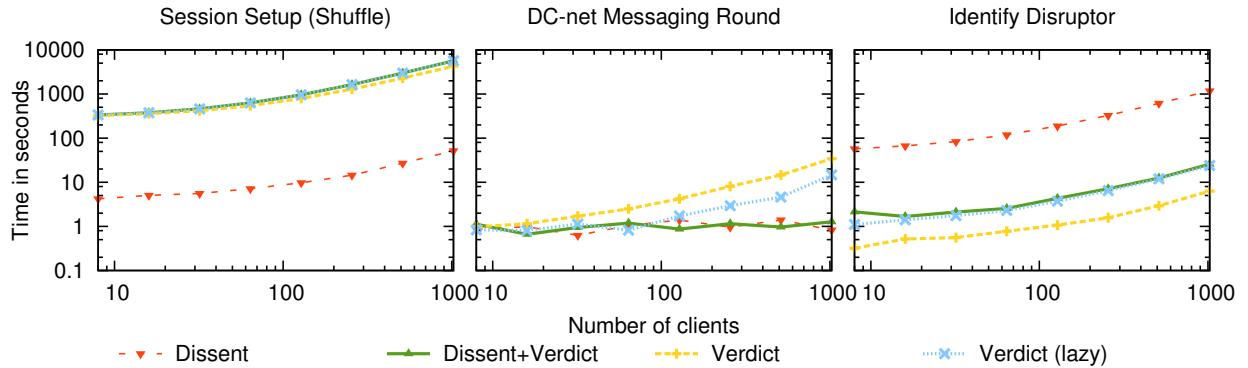


Figure 7: Time required to initialize a session, perform one messaging round, and to identify a disruptor.

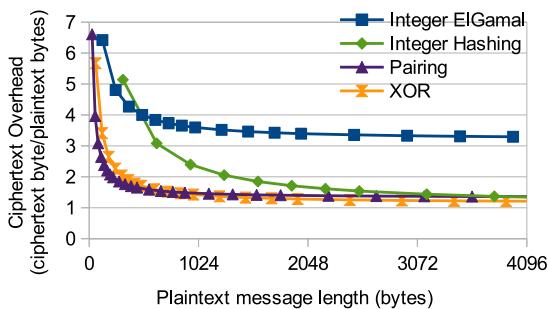


Figure 6: Ciphertext expansion factor (overhead) using the integer ElGamal-style, pairing-based, and hashing-generator protocol variants.

with lazy proof verification, and the Dissent+Verdict hybrid DC-net. We ran this experiment on DeterLab using 8 servers and 128 clients. To scale beyond 128 clients, we ran multiple client processes on each client machine. Session setup time measures the time from session start to just before the first DC-net messaging round.

The one-time session setup time for Verdict is longer than for Dissent because the verifiable shuffle implementation Dissent uses is heavily optimized for shuffling DSA signing keys. Shuffling Verdict public keys, which are drawn from different group types, requires using a less-optimized version of the verifiable shuffle. We do not believe this cost is fundamental to the Verdict approach, and in any case these setup costs are typically amortized over many DC-net rounds.

The Dissent+Verdict hybrid DC-net is just as fast as Dissent in the normal case, since Dissent and the hybrid DC-net run *exactly the same code* if there is no active disruptor in the group. Network latency comprises the majority of the time for a messaging round when using the Dissent and the hybrid Dissent+Verdict DC-nets—messaging rounds take between 0.6 and 1.4 seconds to complete in network sizes of 8 to 1,024 clients.

In contrast, Verdict becomes computationally limited at 64 clients, taking approximately 2.5 seconds per round. Verdict (lazy) improves upon this by becoming computationally limited at 256 clients, requiring approximately 3 seconds per messaging round.

Verdict incurs the lowest accountability (blame) cost of the four schemes. Verdict’s verifiable DC-net checks the validity of each client ciphertext before processing it further, so the time-to-blame in Verdict is equal to the cost of verifying the validity proofs on N client ciphertexts. “Verdict (lazy)” uses the lazy proof verification technique described in Section 5.4—servers verify the client proofs of correctness only if they detect a disruption. Lazy proof verification delays the verification operation to the end of a messaging phase, so the time-to-blame is slightly higher than in pure Verdict.

Dissent, which has the highest time-to-blame, has an accountability process that requires the anonymous client whose message was corrupted to submit an “accusation” message to a lengthy verifiable shuffle protocol, in which all members participate. This verifiable shuffle is the reason that Dissent takes the longest to identify a disruptor. The hybrid Dissent+Verdict DC-net (Section 4.4) avoids Dissent’s extra verifiable shuffle by falling back instead to a verifiable DC-net to resolve disruptions.

As Figure 7 shows, the messaging round time in the hybrid Dissent+Verdict DC-net is as fast as in Dissent, but the hybrid scheme reduces Dissent’s time to detect misbehavior by roughly two orders of magnitude.

6.4 Anonymous Microblogging

Verdict’s ability to tolerate many dishonest nodes makes it potentially attractive for anonymous microblogging in groups of hundreds of nodes. In Twitter, messages have a maximum length of 140 bytes, which means that a single tweet can fit into a few 256-bit elliptic curve group elements. Twitter users can also tolerate messaging latency

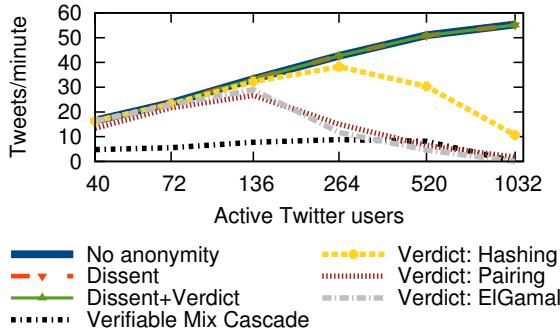


Figure 8: Rate at which various anonymity schemes process tweets, for varying numbers of active users.

of tens of seconds or even a few minutes, which would be unacceptable for interactive web browsing.

This experiment evaluates the suitability of Verdict for *small-scale* anonymous microblogging applications, giving users anonymity among hundreds of nodes, e.g., for students microblogging on a university campus. To test Verdict in this scenario, we recorded 5,000 Twitter users’ activity for one-hour and then took subsets of this trace: the smallest subset contained only the Tweets of the 40 most active users, and the largest subset contained the Tweets of the 1,032 most active users. We replayed each of these traces through Dissent and through Verdict, using each of the three ciphertext constructions.

We ran our experiment on DeterLab [17], on a test topology consisting of eight servers connected to a 100 Mbps LAN with 10 ms of server-to-server latency, and with each set of clients connecting to their upstream server over a *shared* 100 Mbps link with 50 ms of latency. Scarcity of testbed resources limited the number of available delay links, but our experiment attempts to approximate a wide-area deployment model in which clients are geographically dispersed and bandwidth-limited.

Figure 8 shows the Tweet-rate latency induced by the different anonymity systems relative to the baseline (no anonymity) as the number of active users (and hence, the anonymity set size) in the trace increases. Both Dissent and the Dissent+Verdict hybrid systems can keep pace with the baseline in a 1,000-node network—the largest network size feasible on our testbed. The pure Verdict variants could not keep pace with the baseline in a 1,000-node network, while hashing-generator variant of Verdict runs almost as quickly as the baseline in an anonymity set size of 264. These results suggest that Verdict might realistically support proactively accountable anonymity for microblogging groups of up to hundreds of nodes.

Figure 8 also compares Verdict to a mix-net cascade (a set of mix servers) in which each mix server uses a Neff proof-of-knowledge [36] to demonstrate that it has

performed the mixing operation properly. Like Verdict, this sort of mix cascade forms a traffic-analysis-resistant anonymity system, so it might be used as an alternative to Verdict for anonymous messaging. Our evaluation results demonstrate that the hashing-generator variant of Verdict outperforms the mix cascade at all network sizes and that the Tweet throughput of the Dissent+Verdict hybrid is more than 6× greater than the throughput of the mix cascade at a network size of 564 participants.

6.5 Anonymous Web Browsing

Dissent demonstrated that accountable DC-nets are fast enough to support anonymous interactive Web browsing in local-area network deployments [52]. We now evaluate whether Verdict is similarly usable in a web browsing scenario. Our experiment simulates a group of nodes connected to a single WLAN network. This configuration emulates, for example, a group of users in an Internet café browsing the Internet anonymously.

In our simulation on DeterLab [17], 8 servers and 24 clients communicate over a network of 24 Mbps links with 20 ms node-to-node latency. To simulate a Web browsing session, we recorded the sequence of requests and responses that a browser makes to download home page content (HTML, CSS files, images, etc.) from the Alexa “Top 100” Web pages [2]. We then replayed this trace with the client using one of four anonymity overlays: no anonymity, the Dissent DC-net, the Verdict-only DC-net, and the Dissent+Verdict hybrid DC-net. The simulated client sends the upstream (request) traffic through the anonymity network and servers broadcast the downstream (response) traffic to all nodes.

Figure 9 charts the time required to download all home page content using the four different network configurations. The median Web page took one second to load with no anonymity, fewer than 10 seconds over Dissent, and around 30 seconds using Verdict only (Figure 10). Notably, the hybrid Dissent+Verdict scheme exhibits performance nearly identical to that of Dissent alone, since it falls back to the slower verifiable Verdict DC-net only when there is active disruption. The Verdict-only DC-net is much slower than Dissent because every node must generate a computationally expensive zero-knowledge proof in every messaging round.

These experiments show that Verdict adds no overhead to Dissent’s XOR-based DC-net in the absence of disruption. In addition, these experiments illustrate the flexibility of verifiable DC-nets, which can be used either as a “workhorse” for anonymous communication or more selectively in combination with traditional XOR-based DC-nets; we suspect that other interesting applications will be discovered in the future.

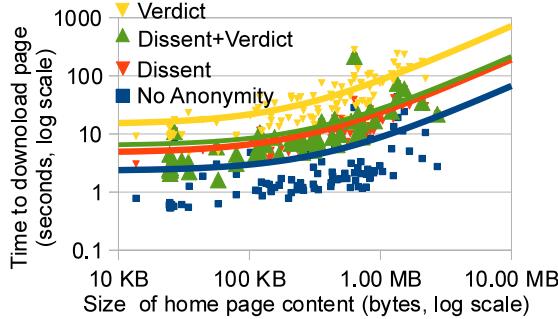


Figure 9: Time required to download home page context for Alexa “Top 100” Web sites (with linear trend lines).

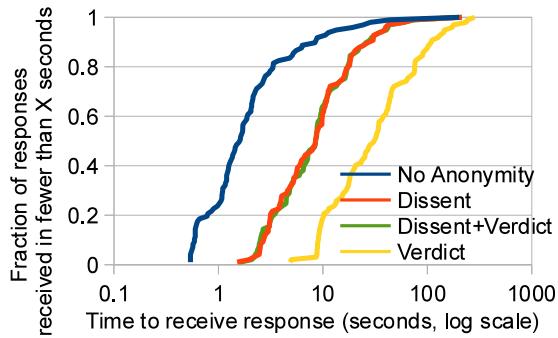


Figure 10: CDF of time required to download home page context for Alexa “Top 100” Web sites.

7 Related Work

Chaum recognized the risk of anonymous disruption attacks in his original formulation of DC-nets [13], and proposed a probabilistic tracing approach based on *traps*, upon which Waidner and Pfitzmann expanded [50].

Herbivore [24, 44] sidestepped the disruption issue by forming groups dynamically, enabling nodes to leave disrupted groups and form new groups until they find a disruption-free group. Unfortunately, the likelihood that a group contains some malicious node likely increases rapidly with group size, and hence anonymity set, limiting this and related partitioning approaches [1] to systems supporting small anonymity sets. Further, in an analog to a known attack against Tor [9], an adversary might selectively disrupt only groups he has only *partially* but not *completely* compromised. With a powerful adversary controlling many nodes, after some threshold a victim becomes *more* likely to “settle into” a group that works precisely because it is *completely* compromised, than to find a working uncompromised group.

k -anonymous message transmission [1] also achieves disruption resistance by partitioning participants into small disruption-free groups. A crucial limitation of the k -anonymity system is that an honest client is anonymous *only* among a small constant (k) number of nodes. In contrast, Verdict clients in principle obtain anonymity among the set of *all* honest clients using the system.

Dissent [14, 52] uses verifiable shuffles [10, 36] to establish a *transmission schedule* for DC-nets, enabling groups to guarantee a one-to-one correspondence of group members to anonymous transmission slots. The original Dissent protocol [14] offered accountability but limited performance. A more recent version [52] improves performance and scalability, but uses a retrospective “blame” protocol which requires an expensive shuffle when disruption is detected.

Golle and Juels [25] introduced the verifiable DC-net concept and formally developed a scheme based on bilinear maps, forming Verdict’s starting point. To our knowledge this scheme was never implemented in a working anonymous communication system, however, and we find that its expensive pairing operations limit its practical performance.

Crowds [40], LAP [27], Mixminion [16], Tarzan [21], and Tor [18], provide anonymity in large networks, but these systems cannot protect against adversaries that observe traffic [4, 35] or perform active attacks [9] on a large fraction of network links. Verdict maintains its security properties in the presence of this type of strong adversary. A cascade of cryptographically verifiable shuffles [23, 36] can offer the same security guarantees that Verdict does, but these shuffles generally require more expensive proofs-of-knowledge.

8 Conclusion

Verdict is a new anonymous group messaging system that combines the traffic analysis resistance of DC-nets with disruption resistance based on public-key cryptography and knowledge proofs. Our experiments show that Verdict may be suitable for messaging in groups of hundreds to thousands of users, and can be combined with traditional XOR-based DC-nets to offer good normal-case performance while reducing the system’s vulnerability to disruption events by two orders of magnitude.

Acknowledgments

We wish to thank Aaron Johnson, Ewa Syta, Michael J. Fischer, Michael Z. Lee, Michael “Fitz” Nowlan, and Ramki Gummadi for their helpful comments. We also thank our shepherd, Micah Sherr, and the anonymous USENIX reviewers, for their valuable feedback. Finally, we thank the DeterLab staff for their flexibility, patience, and support during the evaluation process. This material is based upon work supported by the Defense Advanced Research Agency (DARPA) and SPAWAR Systems Center Pacific, Contract No. N66001-11-C-4018.

References

- [1] Luis von Ahn, Andrew Bortz, and Nicholas J. Hopper. k -anonymous message transmission. In *ACM conference on Computer and Communications Security (CCS)*, pages 122–130, 2003.
- [2] Alexa top 500 global sites, April 2012. <http://www.alexa.com/topsites>.
- [3] Jack M. Balkin. Digital speech and democratic culture: A theory of freedom of expression for the information society. *Faculty Scholarship Series*, 2004. Paper 240.
- [4] Kevin Bauer, Damon McCoy, Dirk Grunwald, Tadayoshi Kohno, and Douglas Sicker. Low-resource routing attacks against Tor. In *Workshop on Privacy in the Electronic Society (WPES)*, pages 11–20, October 2007.
- [5] Mihir Bellare and Phillip Rogaway. Random oracles are practical: A paradigm for designing efficient protocols. In *ACM conference on Computer and communications security (CCS)*, pages 62–73, 1993.
- [6] Dan Boneh. The decision Diffie-Hellman problem. In Joe Buhler, editor, *Algorithmic Number Theory*, volume 1423 of *Lecture Notes in Computer Science*, pages 48–63. Springer Berlin / Heidelberg, 1998. 10.1007/BFb0054851.
- [7] Dan Boneh and Xavier Boyen. Efficient selective-ID secure identity-based encryption without random oracles. In *International Conference on the Theory and Applications of Cryptographic Techniques (Eurocrypt)*, pages 223–238, 2004.
- [8] Dan Boneh and Matt Franklin. Identity-based encryption from the Weil pairing. In *IACR International Cryptology Conference (CRYPTO)*, pages 213–229. 2001.
- [9] Nikita Borisov, George Danezis, Prateek Mittal, and Parisa Tabriz. Denial of service or denial of security? How attacks on reliability can compromise anonymity. In *ACM Conference on Computer and Communications Security (CCS)*, pages 92–102, October 2007.
- [10] Justin Brickell and Vitaly Shmatikov. Efficient anonymity-preserving data collection. In *ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*, pages 76–85, August 2006.
- [11] Jan Camenisch and Markus Stadler. Proof systems for general statements about discrete logarithms. Technical Report 260, Dept. of Computer Science, ETH Zurich, March 1997.
- [12] Miguel Castro and Barbara Liskov. Practical Byzantine fault tolerance. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 173–186, February 1999.
- [13] David Chaum. The dining cryptographers problem: Unconditional sender and recipient untraceability. *Journal of Cryptology*, pages 65–75, January 1988.
- [14] Henry Corrigan-Gibbs and Bryan Ford. Dissent: accountable anonymous group messaging. In *ACM conference on Computer and communications security (CCS)*, pages 340–350, October 2010.
- [15] Henry Corrigan-Gibbs, David Isaac Wolinsky, and Bryan Ford. Proactively accountable anonymous messaging in Verdict. Technical Report YALEU/DCS/TR1478, Department of Computer Science, Yale University, 2013.
- [16] George Danezis, Roger Dingledine, and Nick Mathewson. Mixminion: Design of a Type III anonymous remailer protocol. In *IEEE Security and Privacy (SP)*, pages 2–15, May 2003.
- [17] DeterLab network security testbed, September 2012. <http://isi.deterlab.net/>.
- [18] Roger Dingledine, Nick Mathewson, and Paul Syverson. Tor: the second-generation onion router. In *USENIX Security Symposium*, pages 303–320, 2004.
- [19] John R. Douceur. The Sybil attack. In *1st International Workshop on Peer-to-Peer Systems*, pages 251–260, March 2002.
- [20] Taher ElGamal. A public key cryptosystem and a signature scheme based on discrete logarithms. In George Blakley and David Chaum, editors, *Advances in Cryptology*, volume 196 of *Lecture Notes in Computer Science*, pages 10–18. Springer Berlin / Heidelberg, 1985.
- [21] Michael J. Freedman and Robert Morris. Tarzan: A peer-to-peer anonymizing network layer. In *ACM conference on Computer and communications security (CCS)*, pages 193–206, 2002.
- [22] Eiichiro Fujisaki and Koutarou Suzuki. Traceable ring signature. In *International Conference on Theory and Practice of Public Key Cryptography (PKC)*, pages 181–200, April 2007.
- [23] Jun Furukawa and Kazue Sako. An efficient scheme for proving a shuffle. In *CRYPTO*, pages 368–387, August 2001.
- [24] Sharad Goel, Mark Robson, Milo Polte, and Emin Gun Sirer. Herbivore: A scalable and efficient protocol for anonymous communication. Technical Report 2003-1890, Cornell University, February 2003.
- [25] Philippe Golle and Ari Juels. Dining cryptographers revisited. *Eurocrypt*, pages 456–473, May 2004.
- [26] Jens Groth. *Honest verifier zero-knowledge arguments applied*. PhD thesis, University of Aarhus, October 2004.
- [27] Hsu-Chun Hsiao, Tiffany Hyun-Jin Kim, Adrian Perrig, Akira Yamada, Samuel C. Nelson, Marco Gruteser, and Wei Meng. LAP: Lightweight anonymity and privacy. In *IEEE Security and Privacy*, pages 506–520, 2012.
- [28] Dogan Kedogan, Dakshi Agrawal, and Stefan Penz. Limits of anonymity in open environments. In *5th International Workshop on Information Hiding*, pages 53–69,

October 2002.

- [29] Joseph K. Liu, Victor K. Wei, and Duncan S. Wong. Linkable spontaneous anonymous group signature for ad hoc groups. In *Australian Conference on Information Security and Privacy*, pages 614–623, July 2004.
- [30] Ben Lynn. *On the implementation of pairing-based cryptosystems*. PhD thesis, Stanford University, Stanford, CA, USA, 2007.
- [31] Ewen MacAskill, Julian Borger, Nick Hopkins, Nick Davies, and James Ball. GCHQ taps fibre-optic cables for secret access to world's communications. *The Guardian*, June 2013. <http://www.guardian.co.uk/uk/2013/jun/21/gchq-cables-secret-world-communications-nsa>.
- [32] Ulf Moeller and Lance Cottrell. Mixmaster protocol: Version 2, January 2000. <http://www.eskimo.com/~rowdenw/crypt/Mix/draft-moeller-mixmaster2-protocol-00.txt>.
- [33] Hooman Mohajeri Moghaddam, Baiyu Li, Mohammad Derakhshani, and Ian Goldberg. SkypeMorph: Protocol obfuscation for Tor bridges. In *ACM Conference on Computer and Communications Security (CCS)*, pages 97–108, 2012.
- [34] Steven J. Murdoch and George Danezis. Low-cost traffic analysis of Tor. In *IEEE Security and Privacy*, pages 183–195, May 2005.
- [35] Steven J. Murdoch and Piotr Zieliński. Sampled traffic analysis by Internet-exchange-level adversaries. In *Proceedings of the 7th international conference on Privacy enhancing technologies, PETs'07*, pages 167–183, Berlin, Heidelberg, 2007. Springer-Verlag.
- [36] C. Andrew Neff. A verifiable secret shuffle and its application to e-voting. In *ACM Conference on Computer and Communications Security*, pages 116–125, November 2001.
- [37] National Institute of Standards and Technology. FIPS PUB 186-3: Digital Signature Standard (DSS), 2009.
- [38] Andriy Panchenko, Lukas Niessen, Andreas Zinnen, and Thomas Engel. Website fingerprinting in onion routing based anonymization networks. In *Workshop on Privacy in the Electronic Society (WPES)*, pages 103–114, October 2011.
- [39] Jennifer Preston. Facebook officials keep quiet in its role in revolts. *New York Times*, February 2011. <http://www.nytimes.com/2011/02/15/business/media/15facebook.html>.
- [40] Michael K. Reiter and Aviel D. Rubin. Anonymous Web transactions with Crowds. *Communications of the ACM*, 42(2):32–48, 1999.
- [41] Ronald Rivest, Adi Shamir, and Yael Tauman. How to leak a secret. In *ASIACRYPT*, pages 552–565, December 2001.
- [42] Claus P. Schnorr. Efficient signature generation by smart cards. *Journal of Cryptology*, 4(3):161–174, 1991.
- [43] Berry Schoenmakers. A simple publicly verifiable secret sharing scheme and its application to electronic voting. In *IACR International Cryptology Conference (CRYPTO)*, pages 784–784, 1999.
- [44] Emin Gün Sirer, Sharad Goel, Mark Robson, and Doğan Engin. Eluding carnivores: File sharing with strong anonymity. In *ACM SIGOPS European Workshop (SIGOPS EW)*, September 2004.
- [45] Edward Stein. Queers anonymous: Lesbians, gay men, free speech, and cyberspace. *Harvard Civil Rights-Civil Liberties Law Review*, 2003.
- [46] Al Teich, Mark S. Frankel, Rob Kling, and Ya-ching Lee. Anonymous communication policies for the Internet: Results and recommendations of the AAAS conference. *Information Society*, May 1999.
- [47] Nguyen Tran, Bonan Min, Jinyang Li, and Lakshminarayanan Subramanian. Sybil-resilient online content voting. In *Symposium on Networked System Design and Implementation (NSDI)*, pages 15–28, April 2009.
- [48] Stanford University. The pairing-based cryptography library. <http://crypto.stanford.edu/pbc/>.
- [49] Eugene Vasserman, Rob Jansen, James Tyra, Nicholas Hopper, and Yongdae Kim. Membership-concealing overlay networks. In *ACM conference on Computer and Communications Security (CCS)*, pages 390–399, November 2009.
- [50] Michael Waidner and Birgit Pfitzmann. The dining cryptographers in the disco: Unconditional sender and recipient untraceability with computationally secure serviceability. In *Eurocrypt*, pages 302–319, April 1989.
- [51] David Isaac Wolinsky, Henry Corrigan-Gibbs, Bryan Ford, and Aaron Johnson. Scalable anonymous group communication in the anytrust model. In *European Workshop on System Security (EuroSec)*, April 2012.
- [52] David Isaac Wolinsky, Henry Corrigan-Gibbs, Aaron Johnson, and Bryan Ford. Dissent in numbers: Making strong anonymity scale. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 179–192, October 2012.
- [53] Haifeng Yu, Phillip B. Gibbons, Michael Kaminsky, and Feng Xiao. SybilLimit: A near-optimal social network defense against sybil attacks. In *IEEE Symposium on Security and Privacy*, pages 3–17, May 2008.
- [54] Haifeng Yu, Michael Kaminsky, Phillip B. Gibbons, and Abraham Flaxman. SybilGuard: Defending against sybil attacks via social networks. In *ACM SIGCOMM*, pages 267–278, September 2006.

ZQL: A Compiler for Privacy-Preserving Data Processing

Cédric Fournet
Microsoft Research

Markulf Kohlweiss
Microsoft Research

George Danezis
Microsoft Research

Zhengqin Luo
MSR-INRIA Joint Centre

Abstract

ZQL is a query language for expressing simple computations on private data. Its compiler produces code to certify data, perform client-side computations, and verify the correctness of their results. Under the hood, it synthesizes zero-knowledge protocols that guarantee both integrity of the query results and privacy for all other data.

We present the ZQL language, its compilation scheme down to concrete cryptography, and the security guarantees it provides. We report on a prototype compiler that produces F# and C++. We evaluate its performance on queries for smart-meter billing, for pay-as-you-drive insurance policies, and for location-based services.

1 Introduction

A variety of private user data is used to tailor modern services, and some go as far as billing based on fine grained customer readings. For example, smart meters are used to charge a different tariff depending on the time of electricity usage; pay-as-you-drive insurance premiums depend on detailed driving pattern of drivers. Such schemes are currently implemented by collecting fine-grained information, and processing it on the service side—an architecture that has led to serious privacy concerns.

This paper supports an alternative approach: clients could perform sensitive computations on their own data certified by meters or car on-board units [55, 60], and upload only the results, together with a proof of correctness to ensure their integrity. We propose ZQL, a simple query language to express at a high level such computations, without any cryptographic details. Queries are compiled to code for the data sources, the clients, and the verifiers by synthesizing zero-knowledge protocols.

The most popular language for querying and performing computations on user data is SQL [29] based on relational algebra. The ZQL feature set was chosen to support a subset of SQL. Data is organized into tables of rows, with private and public columns. Queries accept

tables as inputs, and can iterate over them to produce other tables, or aggregate values. Simple arithmetic operations on rows are supported natively, and so is a limited form of SQL joins through table lookups.

ZQL offers advantages over hand-crafted protocols, in that computations are flexible and can be expressed at a high level by application programmers. The computations can also be modified and recompiled, without the need to involve cryptography experts.

The ZQL compiler is free to synthesize custom zero-knowledge protocols behind the scene, and we currently support two main branches, for RSA and Elliptic Curve primitives. We also support a symbolic execution backend to derive estimates of the cost of evaluating and verifying queries. Synthesized protocols themselves are internally represented and optimized as fragments of an extended ZQL language until the final code is emitted. Intermediate ZQL is strongly typed, and precise refinement types can be used to verify security properties on the final compiled code, using F7 [16] or F* [58].

Informally, for a given source query, the desired security properties on the resulting ZQL-compiled code are:

- *Correctness.* For any given source inputs, the sequential composition of the cryptographic queries for the data sources, the user, and the verifier yields the same result as the source query.
- *Integrity.* An adversary given the capabilities of the user cannot get the verifier to accept any other result—except with a negligible probability.
- *Privacy.* An adversary given the capabilities of the verifier, able to choose any two collections of inputs such that the source query yields the same result, and given the result of the user’s cryptographic query, cannot tell which of the two inputs was used.

This corresponds to the source query being executed by a fictional trusted third party sitting between the data sources, the user, and the verifier.

Contents The rest of the paper is organized as follows. §2 introduces our query language using a series of privacy-preserving data processing examples. §3 specifies our target privacy and integrity goals. §4 reviews the main cryptographic mechanisms used by our compiler. §5 describes the compilation process. §6 gives our main security theorems. §7 discusses applications and §8 provides experimental results and discusses future work.

This short version of the paper omits many details and discussions; an extended version with auxiliary definitions, proofs, and examples is available at <http://research.microsoft.com/zql>.

Related work The ZQL language provides private data processing. The zero-knowledge protocols synthesized are standard Σ -protocols [32, 30, 33], but in ZQL they are used for proving the correctness of general computations rather than for cryptographic protocol design.

Arguably, previous works on zero-knowledge compilers focused on the latter as the primary use-case [19, 51, 2, 1]. The use of zero-knowledge for authentication and authorization as in credential and e-cash technologies [23, 51, 4] received particular attention, but, to our knowledge, no-one considered the use of Σ -protocols to prove the execution of general programs.

More specifically, a long line of work [19, 7, 2] culminating in the *CACE* compiler tackles the problem of automatically translating proof goals specified in the Camenisch-Stadler notation [22] into efficient Σ -protocols. Intermediate translations steps of ZQL (the shared translation) are at a similar level of abstraction to the Camenisch-Stadler notation but ZQL also synthesizes those representations from source code, and then proceeds to compile them to low level operations. *ZKPDL* [51], an alternative compiler for Σ -protocols, uses a natural language inspired specification of zero-knowledge proof goals. This specification language may be even closer in spirit to our intermediary notation, as it allows for the possibility to specify the generation of the protocol inputs. The authors of the CACE compiler discuss the difference and similarity between these two approaches in a Usenix poster [9]. The cryptographic prototyping language *Charm* [1] also includes a zero-knowledge proof compiler for Camenisch-Stadler statements which is currently primarily a proof of concept and thus less sophisticated than CACE and ZKPDL. We are also aware of an embedding of a zero-knowledge language in C++ [45].

ZQL differs from standard multi-party computation compilers [49], in that it assumes the client knows all private data. This assumption allows for single round protocols, and the efficient non-interactive implementation of non-linear operations including joins.

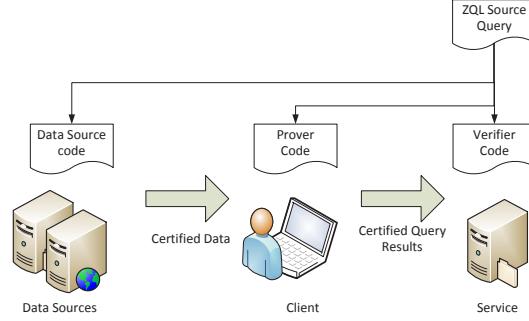


Figure 1: ZQL in a privacy friendly computation system.

2 A Language for Private Data Processing

Why ZQL? We design ZQL to support privacy protocols that rely on client-side computation while requiring high integrity [35]. In this setting, a number of (possibly independent) data sources provide signed personal data items to a user. The signed data is then used as an input to some computation performed on a user device on which the operation of a service relies (for example billing for a utility, determining the proximity of to a specific path, or profiling the shopping habits of a user). The results of the computation are then sent to the relying service, while private input data is kept secret. The ZQL compiler takes a high level description of the computation and is responsible for producing the code executed by the data sources to sign personal data, the computation prover and the computation verifier, as illustrated in Figure 1.

It is assumed that communications take place over private authenticated channels; the data sources are trusted by all to maintain the privacy of the raw personal data they produce, and to securely sign them. Given this, our protocols guarantee integrity through cryptographic proofs that establish the authenticity of the personal data inputs and the correctness of a particular computation. Thus, a malicious client cannot manipulate the result of the computation. On the other hand, the private inputs to the computations are kept secret by the user, and the proofs do not leak anything about them. Thus, privacy is preserved, and only the result of the computations (and any inferences that can be drawn from them) become known to the relying service. Compiling allows us to statically verify the security of the resulting protocols using refinement types (see the full paper). Hence, both the prover and the verifier, or anyone they trust, can separately review the source query, compile the protocol, and verify its security by typing.

There are advantages in de-coupling data sources from specific computations. It allows for meters, or services providing personal data, to remain simple, cheap, and generic. In turn, the computations, such as billing, can be

$e ::=$	Expressions
x	variable
$op \tilde{e}$	application
let $\rho = e$ in e	let binding
$\downarrow e$	declassification
$op ::=$	Operators
$(_, _)$ $(_, _, _)$...	tuples
$+ - *$	arithmetic
$= \wedge$	boolean
map ($\rho \rightarrow e$)	map iterator
fold ($\rho \rightarrow e$)	fold iterator
lookup ρ	table lookup
$\tau ::=$	Types
$int [pub]$	security type
ρ <i>table</i>	table
ρ <i>lookupable</i>	lookup table
$\rho, \theta, \Gamma ::= \varepsilon x : \tau, \rho$	Tuple types

Figure 2: ZQL Syntax

updated without changing the devices that certify readings. Finally, private computations can aggregate disparate data sources that are not aware of one another, or may not trust one another with the privacy or integrity of the computations.

We first provide a brief description of our source language and then illustrate its primitives through simple examples. §7 provides larger examples of protocols that have been proposed in the literature.

The ZQL language At its core, ZQL is a pure expression language, with built-in operators that act on integers and tables. Figure 2 gives its abstract syntax. A query $\theta \rightarrow e$ consists of the declaration of input variables (θ) that can be either public or private, and of an expression body (e).

Expressions consist of variables, operators applied to sub-expressions \tilde{e} (including constants as a special case when \tilde{e} is empty), and let bindings for sequential composition. Expressions evaluate to tuples of values: for example, the expression **let** $x : int, y : int = e$ **in** e_0 first evaluates the sub-expression e to a pair of integers v_x, v_y , then evaluates e_0 after substituting v_x and v_y for x and y .

A variety of operators support arithmetic ($0, 1, +, *$), booleans ($=, \wedge$), and operations on tables (**map**, **fold**, **lookup**). The iterators **map** and **fold** are parametrized by a ZQL expression, conceptually acting as the body of the corresponding loop. (We write these expressions as functions, but they can only specialize the iterator; they cannot be assigned to variables.)

Query inputs and expression results are specified using tuples of typed variables (θ for query inputs and ρ for sub-expressions). Each base type can be marked as

public, and is otherwise treated as private. Types also include tables, where ρ indicates the type of each row in the table. Tables can contain mixtures of public and private columns; for example, $(time:int pub, reading:int)$ table is the type of tables of private readings indexed by public times. On the other hand, the current ZQL compiler does not attempt to hide the query definition itself, or the number of rows in tables.

Intermediate expressions are automatically classified as public or private, depending on the types of their variables, following a standard information flow discipline: public inputs can flow to private results, but not the converse. Alternatively, a ZQL expression can be *explicitly declassified*, using the special operator $\downarrow e$ which specifies that the result of e can be released to the verifier, and marks it as public.

A ZQL query itself defines the privacy goals of the synthesized zero-knowledge protocols. For example, a query $\theta \rightarrow \downarrow e$, where e does not contain any declassification, states that only the final result of the query is released, and that the protocol should not leak any side information on inputs marked as private in θ . A key feature of the language is that the underlying cryptographic mechanisms are totally hidden in the definition of the ZQL query. Since the ZQL query defines what results are declassified, it is important that users, or their proxies, review it to ensure no more than the necessary information leaks from it. Additional privacy mechanisms, such as differential privacy [37], could be used to measure or minimize any leakage resulting from the query declassification.

The ZQL language is strongly typed, with a type system simple enough to allow for automated type checking and type inference, which means that the programmer only has to write the input types of the query. We write $\Gamma \vdash e : \rho$ to state that expression e has type ρ in environment Γ . The type system ensures both *runtime safety*: e returns only to values of types ρ , and *non-interference*: in the absence of declassification, e does not leak inputs typed as private in Γ to results typed as public in ρ . The type system can also be used to track the maximal length of private variables to statically prevent arithmetic overflows. We omit the formal definition of the language semantics and type system, which are standard. Internally, ZQL relies on a richer type system with refinements types [14, 42] to keep track of various properties and to structure our security proofs—see the full paper.

ZQL by example We present the ZQL language and semantics through simple concrete examples, building to fuller queries that address problems in the literature in §7. The first example query computes the discriminant of the polynomial $xk^2 + zk + y$, for public x and private y and z .

let discriminant ($x:int$ pub) ($y:int$) ($z:int$) = $\downarrow (z*z - 4*x*y)$

Anticipating on its compilation, the part of the expression that is linear in the secrets, namely $-4 * x * y$, can be proved efficiently through homomorphisms of Pedersen commitments, while the non-linear $z * z$ requires a Σ -protocol to prove the correctness of the private multiplication. The ZQL compiler will choose to synthesize the right proof mechanisms for each case.

The query declassifies its result, which leaks some information about y and z . For instance, given $x = 30$ and $discriminant\;x\;y\;z = 1000$, if the verifier knows a priori that $0 \leq y < 200$ and $0 \leq z < 200$, then it can infer that (y, z) is one of the pairs $(5, 40), (45, 80), (75, 100)$ or $(155, 140)$, but our privacy theorem ensures that its does not learn which pair was actually used.

Our next examples illustrate the use of tables and iterators **map** and **fold**. The first query computes the sum of all integers in table X , while the second returns the sum of their squares. The third query takes a table with a public column and two secret columns and returns a table with the same public column, and the element-wise sum of the secret columns. By design, the size of the tables is not hidden by ZQL. (Hiding table sizes naively would involve padding the computation to the maximum size of allowed tables, which would be very expensive.)

```
let sum_of_x ( $X$  : int table) =
   $\downarrow$  (fold (( $s, x$ )  $\rightarrow s + x$ ) 0  $X$ )
let sum_of_square ( $X$  : int table) =
   $\downarrow$  (fold (( $s, x$ )  $\rightarrow s + x*x$ ) 0  $X$ )
let linear ( $T$ : (int pub * int * int) table) =
   $\downarrow$  (map (( $a,x,y$ )  $\rightarrow a, x+y$ )  $T$ )
```

In these queries, the iterators are parametrized by a subquery, which is applied to every row of the table, accumulating the sums in s , or building another table of results. The equivalent SQL statements would be written select $\text{SUM}(x)$ from X , select $\text{SUM}(x*x)$ from X , and select $a, x+y$ from T . The first and third queries compute linear combinations of secrets; we compile them without the use of any expensive Σ -protocols.

We found sum queries to be frequent enough to justify some derived syntax: we write **sum** ($\rho \rightarrow e$) T as syntactic sugar for **fold** ($s, \rho \rightarrow s + e$) 0 T .

A key feature of the ZQL language is the ability to perform lookups on input tables. This provides a limited form of *join* and enable the computation of arbitrary functions with small domains. The expression **lookup** $x T$ finds a row x, v_1, \dots, v_n in T that matches x , and returns v_1, \dots, v_n . From an information-flow viewpoint, the result of a lookup on a private variable is also private (even if the lookup table is public); in that case, ZQL leaks no information about which row is returned. If multiple rows match x , the verifier is only able to assert that any matching row was used. If no row matches x , a runtime exception is raised on the prover side, and the proof fails. This semantics allow the implementation of func-

tions, set membership tests, and half-joins.

To enable lookups, each row of the input table currently needs to be signed using a re-randomizable signature by a trusted source, so these tables are given a special type (ρ *lookuptable*) and lookups on intermediate, computed tables are not supported.

The example *blur*, listed below, repeatedly uses a lookup to map private city identifiers to their respective countries; the resulting table is then declassified.

```
let blur ( $X$ : int table) ( $F$ : (int * int) lookuptable) =
   $\downarrow$  (map (city  $\rightarrow$  lookup city  $F$ )  $X$ )
```

The equivalent SQL statement would be `select F.country from X, F where F.city = X.city`. The query implementation relies on a data source that issues a signed table from cities to countries.

3 Security

The next two sections provide rigorous security definitions for what the ZQL compiler achieves and the cryptographic building blocks it uses, necessary for formulating our security theorems in §6. The mere fact that we can give formal cryptographic definitions for a large class of cryptographic protocols relies on our simple expression language having a formal semantic for both source and compiled programs. Readers interested in compiler architecture can jump straight to §5, or those curious about applications can find them in §7.

Notations Consider a well-typed ZQL source query $Q \stackrel{\triangle}{=} \theta \rightarrow e$, with ℓ input variables $\theta = (x_i : \tau_i)_{i=0.. \ell-1}$, that declassifies only its result. As explained in §2, the typed variables θ specify the data sources and privacy policy. Let \vec{T} range over values of type θ , and $R = Q(\vec{T})$ be the corresponding query result. Given Q , our compiler produces queries $(S, (K_i, D_i)_{i=0.. \ell-1}, P, V)$ with formal parameters indicated in parentheses as follows. (We use primed variables for compiled values.)

- S , the setup generator, generates global parameters χ used by the commitment scheme;
- $(K_i)_{i=0.. \ell-1}$, the data sources key generation, generate key pairs $sk_i, vk_i := K_i(\chi)$ used to sign data and verify their signatures;
- $(D_i)_{i=0.. \ell-1}$, the data sources, extend and sign each input: $T'_i := D_i(\chi, sk_i, T_i)$;
- P , the prover, produces an extended result from extended inputs: $R' := P(\chi, \vec{vk}, \vec{T}')$;
- V , the verifier, returns either some source result $R := V(\chi, \vec{vk}, R')$ or a verification error.

Main Properties We first define functional correctness when all participants comply with the protocol.

Definition 1 $(S, (K_i, D_i)_{i=0..l-1}, P, V)$ correctly implements the source query Q when, for any source inputs $\vec{T} : \theta$ and $\chi := S, (sk_i, vk_i := K_i(\chi))_{i=0..l-1}$, we have

$$V(\chi, \vec{vk}, P(\chi, \vec{vk}, D_i(\chi, sk_i, T_i)_{i=0..l-1})) = Q(\vec{T}).$$

We define privacy as indistinguishability between two series of chosen inputs that yield the same query result.

Definition 2 Given a source query Q and an adversary \mathcal{A} , let $\text{Adv}_{\mathcal{A}}^{\text{Priv}} = |2\Pr[\mathcal{A} \text{ wins}] - 1|$ where the event ‘ \mathcal{A} wins’ is defined by the following game:

- (1) The challenger runs S and K_i to generate setup χ and keys \vec{sk}, \vec{vk} ; it provides χ and \vec{vk} to \mathcal{A} .
- (2) The adversary \mathcal{A} provides two vectors of input data $\vec{T}^0 : \theta$ and $\vec{T}^1 : \theta$ such that (a) they coincide on public data and (b) $Q(\vec{T}^0) = Q(\vec{T}^1)$.
- (3) The challenger picks a random bit b , encodes the corresponding inputs $\vec{T}' := (D_i(\chi, sk_i, T_i^b))_{i=0..l-1}$, and generates $R' := P(\chi, \vec{vk}, \vec{T}')$.
- (4) Given R' , \mathcal{A} returns his guess b' , and wins iff $b = b'$.

$(S, (K_i)_{i=0..l-1}, (D_i)_{i=0..l-1}, P, V)$ is (t, ϵ) -private when, for all \mathcal{A} running at most for time t , we have $\text{Adv}_{\mathcal{A}}^{\text{Priv}} \leq \epsilon$.

Note that we do *not* formally provide privacy protection against corrupted data sources. To strengthen our scheme against data source attacks, we would have to rerandomize all cryptographic material flowing from data sources to verifiers, which precludes our efficient use of homomorphic commitments.

We define integrity as a game in which an adversary has to produce an invalid but accepted response.

Definition 3 Given a source query Q and an adversary \mathcal{A} , let $\text{Adv}_{\mathcal{A}}^{\text{Snd}} = \Pr[\mathcal{A} \text{ wins}]$ where the event ‘ \mathcal{A} wins’ is defined by the following game:

- (1) The challenger runs S and K_i to generate setup χ and keys \vec{sk}, \vec{vk} ; it provides χ and \vec{vk} to \mathcal{A} .
- (2) The adversary \mathcal{A} can adaptively corrupt data sources D_i to get their signing keys sk_i and, at the same time, it can obtain signed inputs $T'_i := D_i(\chi, sk_i, T_i)$ for source inputs $T_i : \tau_i$ of its choice.
- (3) Valid results are source values $R = Q(\vec{T})$ such that, for each i , either i was corrupted or T_i was signed. The adversary wins if he outputs R' such that $V(\chi, \vec{vk}, R')$ returns any invalid result R^* .

$(S, (K_i)_{i=0..l-1}, (D_i)_{i=0..l-1}, P, V)$ is (t, ϵ) -sound when, for all \mathcal{A} running at most for time t , we have $\text{Adv}_{\mathcal{A}}^{\text{Snd}} \leq \epsilon$.

Depending on the adversary, there can be zero, one, or numerous valid responses. In fact, depending on the query and the input tables, whether a response is valid may not even be efficiently checkable. The definition is, however, still meaningful.

4 Main Cryptographic Tools (Review)

Signatures A digital signature scheme allows everyone in possession of the verification key vk to verify the authenticity of data signed by the owner of the corresponding signing key sk . We use signatures to let verifiers authenticate data sources. Instead of signing private data in the clear, data sources sign public commitments; thus, the resulting signature tags are also public.

Cryptographic groups Besides conventional digital signatures, for which we use standardized schemes, our remaining cryptographic tools can either be specified for composite order groups, obtained by computing modulo an RSA modulus, or for prime order groups with a bilinear pairing. We use the latter for our presentation and formal analysis as it offers both performance and conceptual advantages.

Let G, \hat{G} , and G_T be groups of prime order q . Let $g \in G$ and $\hat{g} \in \hat{G}$ be generators of G and \hat{G} respectively. A bilinear pairing is an efficiently computable function $e : G * \hat{G} \rightarrow G_T$ that is bilinear, i.e. $\forall a, b \in \mathbb{F}_q : e(g^a, \hat{g}^b) = e(g, \hat{g})^{ab}$ and non-degenerate, i.e. $e(g, \hat{g}) \neq 1$. Whenever possible we perform all operations in the base group G with the shortest representation.

Commitments A commitment scheme allows a user to commit to a hidden value such that he can reveal the committed value at a later stage. The properties of a commitment scheme are *hiding*: the committed value must remain hidden until the reveal stage, and *binding*: the only value which may be revealed is the one that was chosen in the commit stage. We use the perfectly hiding commitment scheme proposed by Pedersen [54]: given a group G of prime order q with generators g and h , generate a commitment C_x to $x \in \mathbb{F}_q$ by sampling a random opening $ox \leftarrow \mathbb{F}_q$ and computing $C_x = g^{x} h^{ox}$. The commitment is opened by revealing both x and ox .

Two useful properties of Pedersen commitments are (i) their *homomorphic property* that allows to derive a commitment to the linear combination of input values; and (ii) their *algebraic structure* that allows for efficient zero-knowledge proofs. For RSA groups, we use commitments with similar properties [43, 34].

Zero-knowledge proofs [59, 39, 11] provide a verifying algorithm with an efficient means for checking the truth of a statement by guaranteeing that given access to a successful proof generation algorithm one can extract a secret witness for said truth. At the same time, zero-knowledge proofs [47, 46], and the related concepts of witness indistinguishable proofs [38, 32], allow the prover to keep this witness secret. We make use of a long line of work on efficient proofs of conjunctions of discrete logarithm (DL) representations [57, 28, 52, 32, 30, 18, 26, 33, 50]. For non-linear computations such as

multiplication, we use the approach of Brands [18], Camenisch [26], and Cramer and Damgård [31].

DL representation proofs are interactive protocols of three or more messages. To ease deployment and minimize communications, we use the Fiat-Shamir Heuristic [40] and replace random messages sent by the verifier with hash function computations. The resulting protocols can still be formally analyzed in the random oracle model [12, 62].

Proof compatible signatures The combination of zero-knowledge proofs and digital signatures allows us to prove authentication properties on private data, such as, for instance, the existence and properties of a matching row when performing a private lookup.

We use CL signatures [20], as they are compatible with DL representation proofs. The original scheme was proven secure under the Strong RSA assumption and requires groups with hidden order [6, 24]. Other CL signature proposals rely on a variety of assumptions based on bilinear pairings [21, 17, 3, 25] and require more standard prime order DL-representation proofs. We also use the scheme of [25], a good trade-off between security and performance. An additional benefit is that it is syntactically very close to RSA-based CL signatures.

To certify our lookup tables, data sources extend each row of the table with a CL signature. For instance, tables of triples of private integers (x_0, x_1, x_2) are extended to tables with rows of the form (x_0, x_1, x_2, e, v, A) . The verification equations for RSA and bilinear pairing based CL signatures are of the form $Z = A^e R_0^{x_0} R_1^{x_1} R_2^{x_2} S^v$ and $\hat{e}(Z, \hat{g}) = \hat{e}(A, pk * g^e) \hat{e}(R_0^{x_0} R_1^{x_1} R_2^{x_2} S^v, \hat{g})$ respectively, where $(Z, R_0, R_1, R_2, S, pk)$ are group elements that form the components of the verification key vk . Both verification equations can be proven using efficient DL representations. The security of these two schemes is based on the *strong RSA* assumption and the *strong Diffie-Hellman* (SDH) assumption respectively.

5 Compiler Architecture

Protocol Overview The ZQL compiler takes a source query, which contains no cryptographic computations, and automatically produces programs for each data source, for the prover, and for the verifier.

First, the compiler augments the source query with various cryptographic commitments to secrets and representation equations to generate a *shared translation* that will lead to both prover and verifier code. Some commitments are computed and signed by the data sources that certify the computation inputs, and simply passed to the prover and verifier programs. Others, representing intermediate secrets in the query, are interleaved with the source computation: for any such secret x , the prover may sample a secret opening ox , compute a Pedersen

commitment $C_x =_G g^x h^{\alpha x}$, and send it to the verifier; and the verifier may check it using a zero-knowledge proof.

Linear relations between secrets do not require complex zero knowledge proofs, as they can be checked by the verifier simply by using the homomorphisms of Pedersen commitments. For example, a private sum $z = x + y$ will have commitment $C_z =_G C_x * C_y$. Such commitments need not be transmitted, as they can be recomputed by the verifier. On the other hand, non-linear relations between secrets, including multiplication and table lookup, require Σ -protocol proofs to be synthesized. For instance, to prove that z is the product of a secret x committed in C_x and a secret y , one proves the conjunction of the representation equations $C_x =_G g^x h^{\alpha x}$ and $1 =_G (C_x)^{-y} g^z h^{\alpha z}$. Note that the second equation uses a variable commitment C_x as a base.

All Σ -protocols used in the compiler come down to proving knowledge of the secret values underlying the discrete logarithm representations of public group elements, and equality relations between the secret values. Assume the ZQL query reduces to proving in zero-knowledge the representations $\vec{C} =_G \vec{e}[\tilde{x}]$ of a number of commitments \vec{C} , represented by public group elements, using a number of secrets \tilde{x} (including secret openings). For the multiplication example above, we have two equations on five secrets: $\vec{C} \equiv (C_x, 1)$, $\tilde{x} \equiv (x, ox, y, oz)$ and $\vec{e}[(\alpha, \beta, \gamma, \delta, \epsilon)] \equiv (g^\alpha * h^\beta, C_x^{-\gamma} g^\delta h^\epsilon)$. The zero-knowledge protocol synthesized works as follows. The prover

- (1) samples a vector of random values \tilde{t} , one for each secret in \tilde{x} ; We call \tilde{t} values the proof randomness;
- (2) computes the challenge $c = H(\vec{e}[\tilde{t}])$;
- (3) computes the responses $\tilde{r} = \tilde{t} - c * \tilde{x}$, for all secrets.

The proof sent to the verifier consists of the public parameters and values, the commitments \vec{C} , the global challenge c , and the responses \tilde{r} . The verifier checks that $H(\vec{C}^c *_G \vec{e}[\tilde{r}]) = c$, which ensures that the prover knows the secret values in the commitments [40, 12]. As detailed below, our compiled prover and verifier programs introduce secrets and process equations on the fly, depending on the query and its inputs.

Once the shared translation is decided, its specialization into prover and verifier code is relatively straightforward. It involves mainly ensuring the right data flows within the query processing to compute all commitments and responses, and to correctly verify them in the same order. The inputs of the shared translation also determine the data source programs that generate keys, compute commitments, and sign extended data.

Embedding cryptography within ZQL Our compiler mostly operates within ZQL, with F# and C++ back-ends to turn the compiled queries into executable code. This

$e ::= \dots$	Expressions
assert $\varphi; e$	static assertion
$op ::= \dots$	Operators
$-1, 0, 1, \dots$	constants
$sample random div$	exponents (mod q)
$*_G =_G exp_G$	group operations
$\hat{e} : G * \hat{G} \rightarrow G_T$	EC bilinear form
$extend finalize$	cryptographic hash
$keygen sign verify$	plain signatures
mapP $_ $ mapV $_ $	translated iterators
foldP $_ $ foldV $_ $	Types
$\tau ::= \dots$	Types
$num xopening xrand$	exponents (mod q)
$elt_G xo commitment$	group elements
$hash$	cryptographic hash
$tag_i sk_i vk_i$	plain signatures

Figure 3: ZQL internal constructs

enables us to reason about code in a simple, domain-specific language. To this end, Figure 3 supplements the source language of Figure 2 with the types and operators for expressing cryptographic operations. Expressions are extended with **assert**, used in the shared translation to embed proof obligations. As an invariant, all asserted equations φ must hold at runtime. We have types and operations for integers modulo q (\mathbb{F}_q , written *num*), for group elements (elt_G), and for bitstrings, and more specific sub-types to keep track of their usage. For instance, *hash* is the sub-type of bitstrings representing cryptographic hashes, and *xopening* is a sub-type of *num* tracking openings generated for the secret value x . In our presentation, we use standard abbreviated forms for their operations; for instance we often omit group parameters, writing g^x for $exp_G g x$.

Setup and Key Generation The abstract setup S produces global parameters χ supplied by our cryptographic runtimes, including q , the prime order of G , \hat{G} , and G_T ; and independent, random generators g , h , $(R_i)_{i=0..n}$, S , Z in G ; and \hat{g} in \hat{G} . Its fixed code is provided by our cryptographic libraries.

We use $D_{LT} \subseteq 0..l - 1$ to denote the subset of data sources that sign lookup table. The key generation K_i is defined as *keygen* χ when τ_i is a scalar or a table ($i \notin D_{LT}$), and as the CL-key generation **let** $sk = sample() \mathbf{in} sk, (\hat{g})^{sk}$ when τ_i is a lookup table ($i \in D_{LT}$). The data source code D_i is explained below, as we discuss these two representations.

Shared Translation We extend the source query with openings and commitments, but not yet with the corresponding proof randomness and responses.

The main difficulty of the translation is to select cryptographic mechanisms, and notably intermediate com-

mitments, to run the private computation: for every private sub-expression, our compiled protocol may rely on zero, one, or more Pedersen openings and commitments, and it may allocate some proof randomness or not.

In this presentation, for simplicity, we give a formal translation that assumes that *all* source private integer variables are handled uniformly, with a commitment in the same group, sharing the same bases, and (later) with a proof randomness for the secret and for its opening. Figure 4 and 5 show how we translate types and expressions, respectively, in this special case. We discuss our general, more efficient compilation scheme below.

A source expression is *public* in a typing environment when all its free variables have public types. The translation leaves public types (1) and expressions (3) unchanged. The translation of a private integer expression is a triple of an integer for the source value, its opening, and its commitment, with the types given on line (2).

Fresh commitments Our compilation rules may require openings and commitments on their arguments, and may not produce openings and commitments on their results. Our compiler attempts to minimize those cases. Nonetheless, assuming for instance that we need a commitment for z , we produce it on demand, using the expression abbreviation **Commit** z below

Commit $z \triangleq$
let $oz : z opening = sample() \mathbf{in}$
let $C_z : oz commitment = g^z *_G h^{\varphi} \mathbf{in}$
assert $C_z = g^z *_G h^{\varphi};$
 z, oz, C_z

The translation is compositional, as can be seen on lines (4,5,6) in the figure. For instance, we translate **let** expressions by translating their two sub-expressions, and we translate source maps to maps that operate on their translated arguments.

The translation assumes prior rewriting of the source query into simpler sub-expressions. For instance, to compile the discriminant query of §2, we first introduce intermediate variables for the private product and the declassification, rewriting expression $\downarrow (z * z - 4 * x * y)$ into

$e_d \triangleq \mathbf{let} p = z * z \mathbf{in} \mathbf{let} d = p - 4 * x * y \mathbf{in} \downarrow d.$

As a sanity check, our translation preserves typing, in an environment extended with the constants used in our cryptographic libraries; variants of this lemma with more precise refinement types for the prover and verifier translation can be used to verify their privacy and integrity.

Lemma 1 (Typing the shared translation) Let $\Gamma_0 \triangleq g, h, Z, R_0, \dots, R_n, S : elt_G, \hat{g}, (pk_i)_{i \in D_{LT}} : elt_{\hat{G}}$. If $\Gamma \vdash e : \rho$, then $\Gamma_0, [\Gamma] \vdash [\Gamma \vdash e] : [\rho]$.

Next, we explain and illustrate the base cases of the shared translation on private expressions.

$$\begin{aligned}
\llbracket x : \tau\{\varphi\} \rrbracket &= x : \tau\{\varphi\} \text{ when } \tau \text{ is public;} & (1) \\
&\quad \text{otherwise:} \\
\llbracket x : \text{int}\{\varphi\} \rrbracket &= x : \text{int}\{\varphi\}, & (2) \\
&\quad \text{ox : } x \text{ opening,} \\
&\quad C_x : x \text{ ox commitment} \\
\llbracket \rho \text{ table} \rrbracket &= \llbracket \rho \rrbracket \text{ table}, s : \text{tag} \\
\llbracket \rho \text{ lookuptable} \rrbracket &= (\rho, \sigma) \text{ table} \\
\sigma &= e : \text{num}, v : \text{num}, A : \text{elt}_G \\
\llbracket \varepsilon \rrbracket &= \varepsilon \\
\llbracket x : \tau\{\varphi\}, \rho \rrbracket &= \llbracket x : \tau\{\varphi\} \rrbracket, \llbracket \rho \rrbracket
\end{aligned}$$

Figure 4: Shared translation of types and environments

Expressions affine in private variables are translated by supplementing the expression with a linear expression on openings and an homomorphic product of commitments (7); we easily check that the resulting triple (z, oz, C_z) is such that $C_z = g^z *_G h^{oz}$. Note that the public constant a_0 is not included in the opening computation.

Expressions polynomial in private variables are translated using an auxiliary representation equation for every product of private expressions, depending on the availability of openings and commitments—see translation rule (8). To illustrate affine and quadratic expressions, let us translate the discriminant query $\theta \rightarrow \downarrow(e_d)$ where the source environment $\theta = x : \text{int pub}, y : \text{int}, z : \text{int}$ specifies that x is public, whereas y and z are private. By definition, the translated environment $\llbracket \theta \rrbracket$ is

$$\begin{aligned}
x &: \text{int pub}, \\
y &: \text{int}, oy : y \text{ opening}, C_y : y \text{ oy commitment}, \\
z &: \text{int}, oz : z \text{ opening}, C_z : z \text{ oz commitment}
\end{aligned}$$

and, from the translation invariant, we already know that $C_y =_G g^y h^{oy}$ and $C_z =_G g^z h^{oz}$. Applying rules (4), (8), (7), and (10) and inlining the definition of **Commit** we arrive at the shared translation

$$\begin{aligned}
\text{let } p, op, C_p &= \\
\text{let } p &= z * z \text{ in} \\
\text{let } o' &= oz * z \text{ in} \\
\text{assert } 1 &= (C_z)^z *_G g^{-p} *_G h^{-o'}; & (E_1) \\
\text{let } op &= \text{sample}() \text{ in} \\
\text{let } C_p &= g^p *_G h^{op} \text{ in} \\
\text{assert } C_p &= g^p *_G h^{op}; & (E_2) \\
&(p, op, C_p) \\
\text{let } d, od, C_d &= \\
&(p - 4 * x * y), (op - 4 * x * oy), (C_p * C_y^{-4*x}) \\
&\downarrow d
\end{aligned}$$

and we easily check that C_d is a commitment to $z^2 - 4xy$ with opening $op - 4x * oy$. The code of the shared translation makes explicit the two representation equations for the private multiplication, presented more abstractly at the beginning of §5. Anticipating on the next stages of

$$\begin{aligned}
\llbracket \Gamma \vdash e \rrbracket &= e \quad \text{when } e \text{ is public} & (3) \\
\llbracket \Gamma \vdash x \rrbracket &= \llbracket \Gamma(x) \rrbracket \quad \text{otherwise} \\
\llbracket \Gamma \vdash \text{let } \rho = e \text{ in } e_0 \rrbracket &= & (4) \\
\text{let } \llbracket \rho \rrbracket &= \llbracket \Gamma \vdash e \rrbracket \text{ in } \llbracket \Gamma, \rho \vdash e_0 \rrbracket \\
\llbracket \Gamma \vdash \text{map } (\rho \rightarrow e) T \rrbracket &= & (5) \\
\text{map } (\llbracket \rho \rrbracket \rightarrow \llbracket \Gamma, \rho \vdash e \rrbracket) \llbracket \Gamma \vdash T \rrbracket & \\
\text{where } \Gamma(T) = \rho \text{ table and } \Gamma, \rho \vdash e : \rho' \\
\llbracket \Gamma \vdash \text{fold } (a : \tau, \rho \rightarrow e) A T \rrbracket &= & (6) \\
\text{fold } (\llbracket a : \tau, \rho \rrbracket \rightarrow \llbracket \Gamma, a : \tau, \rho \vdash e \rrbracket) & \\
\llbracket \Gamma \vdash a \rrbracket \llbracket \Gamma \vdash T \rrbracket & \\
\text{where } \Gamma(T) = \rho \text{ table and } \Gamma, a : \tau, \rho \vdash e : \tau \\
\llbracket \Gamma \vdash a_0 + \sum_{i=1}^n a_i * x_i \rrbracket &= & (7) \\
a_0 + \sum_{i=1}^n a_i * x_i, & \\
\sum_{i=1}^n a_i * ox_i, & \\
g^{a_0} *_G \prod_{i=1}^n (C_{X_i})^{a_i} & \\
\text{when the } a_i \text{ are public and the } x_i \text{ private} \\
\llbracket \Gamma \vdash x * y \rrbracket &= & (8) \\
\text{let } p : \text{int} = x * y \text{ in} \\
\text{let } o' : \text{num} = ox * y \text{ in} \\
\text{assert } 1 &= (C_x)^y *_G g^{-p} *_G h^{-o'}; \\
\text{Commit } p & \\
\text{when } x \text{ and } y \text{ private} \\
\llbracket \Gamma \vdash \text{lookup } x_0 T_i \rrbracket &= & (9) \\
\text{let } x_1, \dots, x_n, e, v, A = \text{lookup } x_0 T_i \text{ in} \\
\text{let } d, od, C_d = \text{Commit } (\text{random}()) \text{ in} \\
\text{let } p = d * e \text{ in} \\
\text{let } o' = od * e \text{ in} \\
\text{assert } 1 &= G C_d^e g^{-p} h^{-o'} \\
\text{let } A' = A * h^{-d} \text{ in} \\
\text{assert } \hat{e}(Z, \hat{g}) \hat{e}(1/A', pk_i) &= G_T \\
&(\prod_{i=0}^n \hat{e}(R_i, \hat{g})^{x_i}) \hat{e}(A', \hat{g})^e \\
&\hat{e}(S, \hat{g})^v \hat{e}(h, \hat{g})^p \hat{e}(h, pk_i)^d \\
\text{Commit } x_1, \dots, \text{Commit } x_n & \\
\text{where } \Gamma(T_i) = (x_i : \text{int})_{i \in 0..n-1} \text{ lookuptable} \\
\llbracket \Gamma \vdash \downarrow x \rrbracket &= \downarrow x \text{ when } x \text{ private} & (10)
\end{aligned}$$

Figure 5: Shared translation of typed source expressions

the translation, the prover will compute C_p , pass it to the verifier, and extend its challenge computation with equation E_2 , whereas the verifier will receive some C_p and use it to check this equation. Note that the cryptographic overhead depends on the target level of privacy: given instead a source environment θ declaring that x is also private, the same discriminant expression would involve representation proofs for two private products.

Private lookups are translated using proofs of knowledge of signatures. To enable this, data sources extend input tables $T : \rho \text{ lookuptable}$, where ρ is of the form $x_0 : \text{int}, \dots, x_n : \text{int}$, into tables $T' : (\rho, \sigma) \text{table}$ with a CL signature at the end of each row, as follows:

$$\begin{aligned} D_i \triangleq \chi, sk, T \rightarrow & \mathbf{map}(x_0 \dots x_n \rightarrow \\ & \mathbf{let } e = \mathbf{random}() \mathbf{in} \\ & \mathbf{let } v = \mathbf{random}() \mathbf{in} \\ & \mathbf{let } A = (\prod_{G,i=0}^n R_i^{x_i} S^v Z^{-1})^{\frac{1}{sk+e}} \mathbf{in} \\ & x_0, \dots, x_n, e, v, A) \\ & T \end{aligned}$$

Although this pre-processing may be expensive for large tables, it can be amortized over many queries.

A lookup within a source query, such as the one from the *blur* query of §2, is translated to a proof of possession of a CL signature. For instance, let us translate the expression **lookup** c F in environment

$$\rho = F : (city : int, country : int) lookuptable, c : int.$$

The environment is first translated to

$$[\![\rho]\!] = (city : int, country : int, \sigma)table, c : int, ox : c opening, C_c : c ox commitment$$

The lookup itself is translated (using rule 9) to

$$\begin{aligned} [\![\Gamma \vdash \mathbf{lookup} c F]\!] = & \mathbf{let } country, e, v, A = \mathbf{lookup} c F \mathbf{in} \\ & \mathbf{let } d, od, C_d = \mathbf{Commit}(\mathbf{random}()) \mathbf{in} \\ & \mathbf{let } p, o' = d * e, od * e \mathbf{in} \\ & \mathbf{assert } 1 =_G C_d^e g^{-p} h^{-o'}; \\ & \mathbf{let } A' = A * h^{-d} \mathbf{in} \\ & \mathbf{assert } \hat{e}(Z, \hat{g}) \cdot \hat{e}(1/A', pk_i) =_{G_T} \\ & \hat{e}(R_0, \hat{g})^c \cdot \hat{e}(R_1, \hat{g})^{country}. \\ & \hat{e}(A', \hat{g})^e \cdot \hat{e}(S, \hat{g})^v \cdot \hat{e}(h, \hat{g})^p \cdot \hat{e}(h, pk_i)^d; \\ & \mathbf{Commit } country \end{aligned}$$

This code first looks for a signed tuple $(city, country, e, v, A)$ in F such that $c = city$ and retrieves the remaining elements; it then proves knowledge of this tuple, without revealing which tuple is used in the proof, by blinding the element A of the signature. (Note that this proof internally relies on a proof of multiplication.)

Iterators and Committed Tables ZQL supports tables with mixed public and private columns, as well as iterators **map** and **fold**. To enable processing on their private contents, data sources extend tables with commitments and sign them. For instance, here is the code for the provider of the table of cities for the *blur* query.

$$\begin{aligned} D_i \triangleq \chi, sk, X \rightarrow & \mathbf{let } X' = \mathbf{map}(x : int \rightarrow \mathbf{Commit} x) X \mathbf{in} \\ & \mathbf{let } H = \mathbf{fold}(H, x, ox, C_x \rightarrow \mathbf{extend} H C_x) H_0 X' \mathbf{in} \\ & X', \mathbf{sign } sk H \end{aligned}$$

This code first uses **map** to extend each source integer with a fresh opening and commitment, using the **Commit** abbreviation; this yield the extended table X' passed to the prover. It then uses **fold** to compute the joint hash of these commitments, and finally signs the result. (In the hash computation, H_0 is some fixed tag, and we omit a conversion from elt_G to $hash$). As outlined at the end of this section, both the prover and the verifier perform some initial processing for these extended

tables: the prover must show his knowledge of the representation for these commitments, and the verifier must verify the signature and the representation proofs for these commitments.

We illustrate the translation of the **map** iterator (5) on the *blur* query from §2. The translation of **fold** (6) is similar. The map expression of *blur* is translated to another map expression, in a translated environment that provides the extended input $X : [\![x : int]\!] table$:

$$\begin{aligned} [\![\Gamma \vdash \mathbf{map}(c \rightarrow \mathbf{lookup} c F) X]\!] = & \mathbf{map}(c, ox, C_c \rightarrow [\![\Gamma, c : int \vdash \mathbf{lookup} c F]\!]) X \end{aligned}$$

and the translation continues with the **lookup** expression, as explained above.

Prover Translation Continuing from the result of the shared translation, the prover translation uniformly turns its assertions into a custom non-interactive Σ -protocol, in two passes, written $[\![-]\!]_1$ and $[\![-]\!]_2$, that produce code first for the message randomness, then for the responses.

Figure 6 defines these two passes, as well as the top-level query translation $[\![-]\!]_{PROVER}$ that combines $[\![-]\!]_1$ and $[\![-]\!]_2$ with additional glue. Overall, the prover for a source query $\theta \rightarrow e$ is thus defined using this translation after the shared translation: $P \triangleq [\![\![\theta \vdash e]\!]\!]_{PROVER}$.

First-message translation In the first pass, H is the public hash incrementally computing the global challenge; a is the accumulated cryptographic evidence that will be sent to the verifier; and every private variable x is replaced with a pair x, t_x where t_x is the proof randomness for x . (Openings are treated as any other secrets.) In combination with the shared translation, every private source expression becomes a tuple of the form $[\![\![e]\!]\!]_1 : (x, t_x, ox, t_{ox}, C_x)$ where x is the value of e , t_x is the randomness for x , ox is an opening for x , t_{ox} is the randomness for ox , and C_x is a commitment to x . For efficiency, all these additional values are optional in our compiler.

Compositionally, the type translation $[\![\rho]\!]_1$ maps shared environments to environments extended with an entry for each proof randomness, and leaves the other entries unchanged; the expression translation $[\![-]\!]_1$ takes H and a as free variables and returns their updated values of the form $\mathbf{extend} \dots (extend H E_1) \dots E_n$, with one exponential expression E_i for each assertion in e , and a, a_1, \dots, a_m for each additional evidence a_j produced by e .

We explain the main cases of the first-pass translation. Public expressions are (still) unaffected. Note that they may include public expressions generated by the shared translation, such as products of commitments. Affine private expressions are translated homomorphically, adding a corresponding linear expression on the proof randomness. Private exponential computations yields evidence that must be communicated to the verifier; we add their results to a . More complex private expression are supplemented with the sampling of a fresh message randomness

for their result—we rely on the assertions introduced by the shared translation to prove those expressions.

Assertions of equations of the form $e_p = e_x$ are transformed into extensions of the global-challenge computation. The left-hand-side must be a public expression, and is discarded. The right-hand-side must be an expression on private variables. Let e_t be the expressions obtained by replacing each of these variables x with t_x . The translation computes it, and extends H with the result. Declassifications are similarly translated: the declassified value x is added to a , and the hash is extended with g^{t_x} to link it to its proof randomness (as if we were translating `assert gx=gx`). Continuing with our example, we give below the expression e_1 , obtained by translating the shared-translation of the discriminant query, after removing the unnecessary commitment C_d . (This code has been rearranged for simplicity; the full code produced by the translation rules appears in the full paper.)

```

let p = z*z in
let tp = random() in
let o' = oz * z in
let to' = random() in
let H = extend H((Cz)tz*Gg-tp*Gh-to') in
let op = sample() in
let top = random() in
let Cp = gp*Ghop in
let H = extend (extend H Cp) (gtp*Ghtop) in
let d = p - 4*x*y in
let td = tp - 4*x*ty in
let H = extend (extend H gd) gtd in
let a = a, (p,tp), (o',to'), (op,top), Cp, d in
(H, a, d)

```

Response Translation In the second pass, after completing the computation of the global challenge c , we revisit the collected evidence a , and we replace every pair of a private value x and associated proof randomness t_x with the response $r_x = t_x - c*x$. This pass is defined by induction on the *type* of a , produced by the first-message translation, which indicates where those pairs are. (Technically, this pass also needs to re-balance nested tuples, as the prover produces $(\dots(a_0, a_1), a_2, \dots, a_n)$ whereas the verifier consumes $(a_0, (a_1, (\dots a_n) \dots))$; we omit those details.) Continuing with the discriminant prover, the resulting evidence $a : \delta$ binds the series of variables

$(z, t_z), (oz, t_{oz}), (p, t_p), (o', t_{o'}), (op, t_{op}), C_p, d$

and thus $\llbracket \delta \rrbracket_2$ simply computes the responses for the five pairs of secret and associated proof randomness:

$$\begin{aligned} \llbracket \delta \rrbracket_2 &\stackrel{\Delta}{=} \text{let } (z, t_z), (oz, t_{oz}), (p, t_p), (o', t_{o'}), (op, t_{op}), C_p, d = a \\ &\quad \text{let } r_z = t_z - c * z \\ &\quad \text{let } r_{oz} = t_{oz} - c * oz \\ &\quad \text{let } r_p = t_p - c * p \\ &\quad \text{let } r_{o'} = t_{o'} - c * o' \\ &\quad \text{let } r_{op} = t_{op} - c * op \\ &\quad (r_z, r_{oz}, r_p, r_{o'}, r_{op}, C_p, d) \end{aligned}$$

Top-Level Prover Translation (P) We arrive at the following code for the prover, given here for the discriminant query. (See Figure 6 for the general case.) This prover relies on data sources extending both private source inputs y and z with an opening, a commitment, and a signature on that commitment

```

x, (y, oy, Cy, σy), (z, oz, Cz, σz) →
let H = extend (extend H0 Cy) Cz
let tz = random()
let toz = random()
let a = (z, tz), (oz, toz)
let H = extend H (gtz*Ghtoz)
let H, a: δ, d = [[θ]] ⊢ [e]]1 // phase 1 detailed above
let c = finalize H in
let a = [[δ]]2 // phase 2 detailed above
(x, (Cy, σy), (Cz, σz), a, c)

```

In this code, H_0 is the hash of all public values used as bases in the Σ -protocol, $[[θ]]_D$ is the tuple type of the (extended) provided data, and $[[θ]]_{pub}$ is an expression that extracts their public parts (including the plain signatures, excluding lookup tables). The type δ of the additional evidence depends on the first-pass of the translation, and is used to drive the second part. In-between, the final value $H : hash$ is finalized into the global challenge $c : num$. The last line assembles the message passed from the prover to the verifier, which consists of (1) the public parts of the input data and of the result; (2) the additional evidence for proving this result; and (3) the global challenge for verifying this proof.

Verifier Translation Also following the shared translation, the prover translation leaves the public parts of the query unchanged, and it incrementally re-computes the challenge using the responses and additional evidence prepared by the prover for the private parts of the query. Figure 7 gives the compositional translation applied to the result of the shared translation, and the top-level translation $\llbracket _\rrbracket_{VERIFIER}$. In combination, the verifier is defined as $V \stackrel{\Delta}{=} \llbracket [[θ \vdash e]] \rrbracket_{VERIFIER}$.

Compositional translation $\llbracket _\rrbracket_V$ In the verification pass, H is the public hash incrementally re-computing the global challenge, a is the received evidence consumed by the verifier, and every private variable x is replaced with a (public) response variable r_x —the type translation $\llbracket \rho \rrbracket_V$ performs this replacement. In combination with the shared translation, every private source expression now yields a tuple of the form r_x, r_{ox}, C_x where r_x and r_{ox} are (presumably) responses associated with the exponents committed to C_x . (Again, all these values are actually optional in the compiler.)

The verifier expression $\llbracket e \rrbracket_V$ takes free variables H and a , and additionally returns the updated H and the rest of a . Public expressions are unchanged. Private expressions are discarded, and replaced with response expressions, either computed (for affine expressions) or read

off the evidence a (for more complex expressions). Note that the translation of affine expressions includes a term $-c * a_0$ for the constant, to ensure that, given correct responses for its free variables, the translation of an expression also produces a correct response.

Assertions of equations of the form $e_P = e_x$ are translated to hash computations, by computing the expression $(e_P)^c * e_r$, where e_r is obtained from e_x by replacing every variable x with r_x , and by extending H with the result. Declassifications $\downarrow x$ are similarly translated by reading x off the evidence a and extending the hash with g^{x+c*r_x} .

For instance, continuing with the discriminant query, the (simplified) verifier translation $\llbracket \llbracket \rho \rrbracket \vdash \llbracket e_d \rrbracket \rrbracket_v$ is

```
let rp, ro', rop, Cp, d, a = a in
let H = extend H ((Cz)rz *G g-rp *G h-ro') in
let H = extend (extend H Cp) ((Cp)c *G grp *G hrop) in
let rd = rp - 4 * x * ry in
let H = extend (extend H gd) ((gd)c *G grd) in
(H, a, d)
```

Top-Level Verifier We finally give below the top-level verifier translation, also for our sample discriminant query; see Figure 7 for additional details.

```
x, Cy, σy, Cz, σz, a, c →
  verify vky Cy σy;
  verify vkz Cz σz;
let H = extend (extend H0 Cy) Cz
let rz, ro', a = a in
let H = extend H Czc *G gzr *G hro' in
let H, a, d =  $\llbracket \llbracket \theta \rrbracket \vdash \llbracket e \rrbracket \rrbracket_v$  in // translation detailed above
check c = finalize H;
d
```

The prover first verifies the signatures on the two received commitments for y and z ; it starts the challenge re-computation on the representation equation for input z (since we need a response for z an o'_z to check the proof of the square z^2), then proceeds with the verification for the query expression; it checks that the received and re-computed challenges match; it finally returns the public result d (unless of course *verify* or *check* raised an error.)

6 Security Theorems

Consider a well-typed ZQL source query $Q \stackrel{\Delta}{=} \theta \rightarrow \downarrow e$, with ℓ input variables $\theta = (x_i : \tau_i)_{i=0..l-1}$, that declassifies only its result and its translation $(S, (K_i, D_i)_{i=0..l-1}, P, V)$. We give our main results based on the definitions of §3. We refer to the full paper for the proof outlines, and for a discussion of automated, type-based verification for the compiled protocols. For functional correctness and soundness, we also suppose that there is no source-program overflow—formally, integers and their operations are computed modulo q .

Theorem 1 (Functional Correctness)

$(S, (K_i, D_i)_{i=0..l-1}, P, V)$ is correct.

Theorem 2 (Perfect Privacy)

$(S, (K_i, D_i)_{i=0..l-1}, P, V)$ is $(t, 0)$ -private.

Our soundness theorem below is in the random-oracle model, requiring that *extend* and *finalize* are independent random oracles. It assumes that the Discrete Logarithm (DL) and Strong Diffie Hellman (SDH) assumptions hold—to guarantee the security of commitments and CL signatures, respectively—and assuming that the ℓ_{CMA} conventional signatures primitives of data-sources are chosen message attack secure (CMA).

Theorem 3 (Computational Soundness)

$(S, (K_i, D_i)_{i=0..l-1}, P, V)$ is (t, ε) -sound, where the execution time t and success probability ε are respectively lower- and upper-bounded by the corresponding parameters of the assumptions.

Concretely, let t_{DL} , t_{SDH} , t_{CMA} and ε_{DL} , ε_{SDH} , ε_{CMA} be those parameters, for large enough bounds on the number of calls to their primitives. If $t < t_{CMA} - t_{red1}$, $t < (t_{DL} - t_{red2})/2$, and $t < (t_{SDH} - t_{red3})/2$, where the t_{redi} are small constants, then $\varepsilon < \ell_{CMA} \cdot \varepsilon_{CMA} + Q \cdot \sqrt{\varepsilon_{DL} + (\ell - \ell_{CMA}) \cdot \varepsilon_{SDH}} + Q^2/q$, where Q is the number of random oracle queries made by \mathcal{A} and q is the order of G and thus also the size of the challenge.

In contrast with our privacy theorem, which is information-theoretic, our concrete-security soundness theorem is somewhat more cumbersome than the asymptotic security theorems often found in theoretical cryptography, but it remains closer to reality, in which cryptographic primitives come with concrete security bounds, and thus provides guidance for configuring these primitives to achieve adequate security.

7 ZQL applications

The expressivity of ZQL stems from the ease with which the primitive operators can be composed to build larger queries. We illustrate this by providing queries for applications in prior literature.

In the setting of smart metering, a meter issues signed private readings, and a household needs to compute their bill on the basis of a public tariff policy that maps each reading to a fee over time. A number of custom privacy protocols have been proposed to do this [55, 48]. One such billing policy takes a table of public times and private readings, as well as a lookup table from readings to prices to be summed:

```
let smart_meter_bill
  (R: (int pub * int) table) // time, reading
  (T: (int * int) lookuptable) = // reading, fee
  ↓ (sum ((time, reading) → lookup reading T) R)
```

The query looks up the non-linear price of each reading in the table T using **lookup** and sums the results.

Another popular application in the literature involves pay-as-you-drive insurance schemes. Such schemes require drivers to fit a black box in their car that records their driving habits, and allow the insurer to compute a premium based on the safety of the driving, as well as distance or time. The use of zero-knowledge protocols to support such automotive settings, including road usage billing and tolling has been well established in the literature [5, 61, 44].

An example policy used by a UK auto insurance pilot scheme involves recording the segment of road travelled, the distance and the speed and use those to subtract “points” from a virtual driving license. Points are linked to the magnitude of speed violations on the road segments travelled. The insurance rate per mile is then computed as a function of the points subtracted, up to a threshold where the insurance becomes invalid. We can express such a policy in ZQL using a table for the recorded road segments used, and lookup tables to encode the speed limit of road segments, the penalty points per magnitude of violation, and finally the insurance premium for a certain number of points:

```
let pay_as_you_go
  (Segments : (int * int * int * int) table)
  (Limits : (int * int) lookuptable )
  (Penalties : (int * int) lookuptable )
  (Rates : (int * int) lookuptable ) =
let points =
  sum ((time, road, speed, miles) →
    let limit = lookup road Limits
    lookup (speed - limit) Penalties) Segments
let rate = lookup points Rates
let miles =
  sum ((time, road, speed, miles) →miles) Segments
  ↓ (miles * rate)
```

The *pay as you go* application makes extensive use of lookup tables to simulate traditional database half-joins between tables. The values of these tables are largely arbitrary and related to the insurance policy. We note that to fully secure this insurance mechanism, some information about the start and end times of the segments must also be signed by the black box and verified to avoid malicious replays or omissions. We also note that, depending on policies, the query leaks information from individual secret inputs to the computed premium. Securing against source query leakage is beyond the remit of ZQL, but could be achieved by adapting differentially private schemes [36].

The final example illustrates how ZQL lookups can be used to approximate functions on real numbers. A very common problem in privacy preserving protocols for location based services is to prove that the reading from a trusted sensor is at a certain distance from a specific location. For example privacy friendly theft prevention system may need to periodically prove that a trusted reading

is within a certain distance from their (secret) home location [56]. Similar protocols can be of use for offender monitoring, curfew enforcement or tracking of trucks of goods. Previous work has proposed zero-knowledge distance protocols, such as [15].

The *gps distance* protocol takes as secret inputs the longitude and latitude of two points, as well as some precomputed tables, and returns an approximation of the distance between the two points in meters. The approximation used works for small distances under the assumption that the curvature of the earth is negligible. It still requires the computation of the trigonometric function $\cos(x/2)$. To achieve this, we assume the input longitude and latitudes are in the units $rad/10^5$, and that intermediate computations are precise to two decimal points.

```
let gps_distance (lat1: int) (lon1: int) (lat2: int) (lon2: int)
  (hcos: (int * int) lookuptable )
  (red: (int * int) lookuptable)
  (dist: (int * int) lookuptable) =
let latsum = lat1 + lat2
// Table: hcos(x) = round( $\cos((x)/2 \cdot 10^5) \cdot 10^2$ )
let hc = lookup latsum hcos
let dlat = lat2 - lat1
let dlon = lon2 - lon1
let lon_cos = dlon * hc
// Table: red(x) = round( $(x/10^2) \ln(\text{rad}/10^5)^2$ )
let r2 = lookup lon_cos red
let squares = dlat*dlat + r2
// Table: dist(x) = round( $\sqrt{x} \cdot R/10^5$ )
// where R is earth's radius (meters).
↓ (lookup squares dist)
```

In this example, lookups are used to approximate real functions, including trigonometric functions and division which is not yet natively supported. The *hcos* table has a large domain (~ 1 million items) but can be reused across multiple operations. Other tables have a relatively small domain related to the distances of the points compared.

8 Discussion

Prototype implementation & limitations Our compiler uses the language development and testing facilities of F#: we program source queries as (a small subset of) F#, then extract the ZQL abstract syntax tree (AST) through reflection. The compilation pipeline performs ZQL type-checking, applies the shared translation, and finally produces the data-source, prover and verifier code. Each of these steps operates on well-typed ZQL expressions. This enables us to share many optimizations as ZQL-to-ZQL transformations.

Besides standard optimizations, the compiler supports a more general variant of **lookup** primitive, named **find**, that returns any lookup-table row that meets a condition expressed as a boolean expression on the whole content

of the row. This provides more flexibility on the use of lookup tables, but its compilation is more complex.

In addition to cryptographic code, ZQL also synthesizes a custom marshaller and un-marshaller for the cryptographic evidence and results of the query. Following the ZQL approach, this code is specialized and compiled for a specific proof. Hence, the size and location of all fields, parametrized on the input table lengths, is known at compile time and there is no need to rely on a general-purpose parser, a component that is traditionally a source of security flaws.

We support three distinct compiler back-ends:

Concrete F# The main branch of the compiler transforms and compiles the final ZQL data source, prover and verifier into F# code, linked either to the standard .NET big integer libraries, or to proprietary managed libraries that support pairing based cryptography.

Symbolic F# The second branch of the compiler is linked against symbolic execution libraries for all the operators and primitives. Interestingly, since the F# branch makes extensive use of abstract types in the final prover and verifier, there is no need to write a separate symbolic execution environment: the mathematical functions can simply be replaced with equivalents computing on symbolic polynomials. The resulting code jointly computes the execution time and the proof size, as polynomial expressions of the input lengths and the unit costs of each cryptographic operation. We use symbolic execution to predict the performance of the compiler, and hope to use it in the future to chose between alternative optimization strategies at compile time.

Concrete C++ Finally, we support compilation of the verifier to native C++ code, linked with high performance native big integer libraries. This branch involves transforming the functional ZQL verifier and un-marshaller code into an imperative program and optimizing it using standard low-level techniques such as removing dead code, removing spurious copies, and minimizing memory re-allocations. The resulting native program takes a proof as an input, and outputs the verified result. The native branch does not support on-the-fly compilation and execution, and currently works for RSA groups only. Yet the resulting binary can be easily deployed where .NET runtimes are not available.

The process of compiling a query remains fast even on small devices. Thus, a service could simply send ZQL queries to the user, to be reviewed, compiled, then executed locally. To this end, our compiler also has an API that takes source ZQL ASTs, compiles them to F#, then also compiles and dynamically load the resulting F# code. This is likely to be faster, cheaper, safer and more reliable than providing custom binaries every time the query is updated.

The prototype compiler is still subject to limitations. For instance, some optimizations, such as moving de-classifications up in the dataflow to minimize the size of the Σ -protocol, or batching some exponential computations, could be systematically applied.

Performance Evaluation Table 1 illustrates the performance of ZQL code for the three applications presented in Section 7. It provides the execution time for the F# provers and verifiers, as well as the size of the proof, for different security parameters of RSA (1024 bits, 2048 bits) and the pairing based cryptography over a 254 bits Barreto-Naehrig curve (BN254). The *smart_meter_bill* readings table is of size $\ell_{read} = 5$ and the *pay_as-you-go* query road segments table is of size $\ell_{seg} = 25$. This means that for the 1024 bit RSA branch, the prover can process a meter reading every $\sim 120mS$ or a segment of road every $\sim 360mS$. The proof size for the pairing based branch is ~ 755 bytes per reading and ~ 1921 bytes per segment. As expected, the pairing based proofs are more compact than their RSA counterparts for the same or even higher levels of security: a 254 bits curve provides about 128 bits of security which would correspond to a 3072 bits RSA modulus.¹ This is further aggravated by the lack of tightness in RSA-based security reductions [8]. Prover timings take into account the generation of random numbers. We note that these numbers, while slow by the standards of non-privacy friendly computation, are perfectly adequate for computing bills and insurance premiums in real time.

Besides the main F# backend we experimented with a C++ back-end that compiles to a native verifier. Although more performant in absolute terms, the native verifier is not significantly faster than its F# counterpart. The RSA 1024 bit computation of the *pay_as-you-go* verifier took $4,290mS$ as compared with the F# backend using native big integer binding that took $5,111mS$. Profiling the C++ execution indicates that more than 90% of the time is spent inside the modular multiplication function performing exponentiations. Thus, improving the performance of ZQL comes down to either faster exponentiations (through batching, multi-exponentiation or hardware) or reducing the number of operations required through more aggressive simplification of the protocols.

Finally, table 1 illustrates the output of the symbolic execution engine on these three applications, in a configuration that measures the number of exponentiations (E), pairings (\hat{e}), and signature verification operations ($sigv$) in terms of the length of the input tables (ℓ_{read} and ℓ_{seg}), and ignore all other costs.

Where next? The current ZQL language is subject to some intrinsic limitations, and we are actively exploring options to overcome them.

¹http://www.cryptopp.com/wiki/Security_Level

Examples (branch)	prover (mS)	verifier (mS)	proof size (Bytes)
smart meter bill (1024)	586	599	6,106
smart meter bill (2048)	3,498	3,148	10,585
smart meter bill (BN254)	1,374	2,092	3,773
smart meter bill (symbolic)	$E + 16 \cdot E \cdot \ell_{read} + 6 \cdot \ell_{read} \cdot \hat{e}$	$6 \cdot E + 14 \cdot E \cdot \ell_{read} + 8 \cdot \ell_{read} \cdot \hat{e} + sigv$	$67 + h + sig + 2 \cdot \ell_{Ga} + \ell_{Ga} \cdot \ell_{read} + 22 \cdot \ell_{read} + 2 \cdot \ell_{read} \cdot q + num + 7 \cdot q$
pay as you go (1024)	5,314	5,111	57,368
pay as you go (2048)	32,442	30,859	100,099
pay as you go (BN254)	8,305	12,261	28,819
pay as you go (symbolic)	$15 \cdot E + 40 \cdot E \cdot \ell_{seg} + 12 \cdot \ell_{seg} \cdot \hat{e} + 6 \cdot \hat{e}$	$29 \cdot E + 35 \cdot E \cdot \ell_{seg} + 16 \cdot \ell_{seg} \cdot \hat{e} + 8 \cdot \hat{e} + sigv$	$167 + h + sig + 6 \cdot \ell_{Ga} + 4 \cdot \ell_{Ga} \cdot \ell_{seg} + 56 \cdot \ell_{seg} + 8 \cdot \ell_{seg} \cdot q + num + 23 \cdot q$
gps dist (1024)	501	529	5044
gps dist (2048)	3,017	2,889	8629
gps dist (BN254)	841	1,253	2751
gps dist (symbolic)	$60 \cdot E + 18 \cdot \hat{e}$	$71 \cdot E + 24 \cdot \hat{e} + 4 \cdot sigv$	$233 + h + 4 \cdot sig + 10 \cdot \ell_{Ga} + 33 \cdot q$

Table 1: Performance for our three applications: runtime, and communicated proof sizes. The *smart_meter_bill* readings table is of size $\ell_{read} = 5$, the *pay_as_you_go* query road segments table is of size $\ell_{seg} = 25$, the *gps_distance* is between two points.

Many of the limitations are cryptographic and could be overcome by applying more advanced protocols. For example, **lookup** and **find** are currently restricted to externally signed tables. Lookup tables based on accumulators [13] or vector commitments [27] would be more flexible and may reduce cost. At a lower level, table processing leads to many similar cryptographic operations in a data-parallel style. Batch proof and verification techniques and homomorphic signature schemes could speed them up [10]. Well known, zero-knowledge proofs for disjunctions, would allow ZQL branching statements. The shared translation could bundle multiple secrets per commitment. Alternatively one could also employ completely different low-level proof engines, e.g., [53]. We note that choosing automatically the best encoding and technique, as well as compiling them in a compositional manner are challenging open problems. For some preliminary work in this direction see [41].

On the language design side, we illustrated in §7 how functions can be approximated through lookups. ZQL could automate and optimize the process by compiling data sources that calculate and sign function-tables appropriately. Finally, by design, our source language shields programmers from cryptography, and this may hinder power-users that wish to customize our compilation scheme, or experiment with its variants. Similarly, some users may wish to rely on external, unverified procedures, and use ZQL only to validate their results. Advanced APIs exposing the internals of the ZQL compiler without breaking its invariants would help them.

Acknowledgments The authors would like to thank Ian Goldberg for early discussions of languages for zero-knowledge proofs and the advantages of compilation versus interpretation, and Nikhil Swamy for his comments.

References

- [1] J. A. Akinyele, M. D. Green, and A. D. Rubin. Charm: A framework for rapidly prototyping cryptosystems. Cryptology ePrint Archive, Report 2011/617, 2011.
- [2] J. B. Almeida, M. Barbosa, E. Bangerter, G. Barthe, S. Krenn, and S. Z. Béguelin. Full proof cryptography: verifiable compilation of efficient zero-knowledge protocols. In *ACM Conference on Computer and Communications Security*, pages 488–500, 2012.
- [3] M. H. Au, W. Susilo, and Y. Mu. Constant-size dynamic k-TAA. In R. D. Prisco and M. Yung, editors, *SCN 2006*, volume 4116 of *LNCS*, pages 111–125, Maiori, Italy, 2006. Springer.
- [4] M. Backes, M. Maffei, and K. Pecina. Automated synthesis of privacy-preserving distributed applications. *19th Annual Network & Distributed System Security Symposium (NDSS12)*, 2012.
- [5] J. Balasch, A. Rial, C. Troncoso, B. Preneel, I. Verbauwhede, and C. Geuens. PRTP: Privacy-preserving electronic toll pricing. In *USENIX Security Symposium*, pages 63–78, 2010.
- [6] E. Bangerter, J. Camenisch, and U. M. Maurer. Efficient proofs of knowledge of discrete logarithms and representations in groups with hidden order. In *Public Key Cryptography*, pages 154–171, 2005.
- [7] E. Bangerter, T. Briner, W. Henecka, S. Krenn, A.-R. Sadeghi, and T. Schneider. Automatic generation of sigma-protocols. In *EuroPKI*, pages 67–82, 2009.
- [8] E. Bangerter, S. Krenn, A.-R. Sadeghi, T. Schneider, and J.-K. Tsay. On the design and implementation of efficient zero-knowledge proofs of knowledge. *ECRYPT workshop on Software Performance Enhancements for Encryption and Decryption and Cryptographic Compilers (SPEED-CC '09)*, 2009.
- [9] E. Bangerter, S. Krenn, A.-R. Sadeghi, and T. Schneider. Yaczk: Yet another compiler for zero-knowledge. In *USENIX Security Symposium*, 2010.
- [10] S. Bayer and J. Groth. Efficient zero-knowledge argument for correctness of a shuffle. In *EUROCRYPT*, pages 263–280, 2012.
- [11] M. Bellare and O. Goldreich. On defining proofs of knowledge. In *CRYPTO*, pages 390–420, 1992.
- [12] M. Bellare and P. Rogaway. Random oracles are practical: A paradigm for designing efficient protocols. In *ACM Conference on Computer and Communications Security*, pages 62–73, 1993.
- [13] J. C. Benaloh and M. de Mare. One-way accumulators: A decentralized alternative to digital signatures (extended abstract). In

- EUROCRYPT*, pages 274–285, 1993.
- [14] J. Bengtson, K. Bhargavan, C. Fournet, A. D. Gordon, and S. Maffei. Refinement types for secure implementations. In *21st IEEE Computer Security Foundations Symposium (CSF'08)*, pages 17–32, 2008.
- [15] T. S. Benjamin. Zero-knowledge protocols to prove distances. Personal communication, 2008.
- [16] K. Bhargavan, C. Fournet, and A. D. Gordon. F7: refinement types for F#, 2008. Microsoft Research Technical Report.
- [17] D. Boneh, X. Boyen, and H. Shacham. Short group signatures. In *CRYPTO*, pages 41–55, 2004.
- [18] S. Brands. Rapid demonstration of linear relations connected by boolean operators. In *EUROCRYPT*, pages 318–333, 1997.
- [19] T. Briner. Compiler for zero-knowledge proof-of-knowledge protocols. Master thesis, ETH Zurich & IBM Research Lab Zurich, 2004.
- [20] J. Camenisch and A. Lysyanskaya. A signature scheme with efficient protocols. In *SCN*, pages 268–289, 2002.
- [21] J. Camenisch and A. Lysyanskaya. Signature schemes and anonymous credentials from bilinear maps. In *CRYPTO*, pages 56–72, 2004.
- [22] J. Camenisch and M. Stadler. Efficient group signature schemes for large groups. In B. Kaliski, editor, *Advances in Cryptology — CRYPTO '97*, volume 1296 of *LNCS*, pages 410–424. Springer Verlag, 1997.
- [23] J. Camenisch and E. Van Herreweghen. Design and implementation of the *idemix* anonymous credential system. Technical Report Research Report RZ 3419, IBM, May 2002.
- [24] J. Camenisch, A. Kiayias, and M. Yung. On the portability of generalized schnorr proofs. In *EUROCRYPT*, pages 425–442, 2009.
- [25] J. Camenisch, M. Kohlweiss, and C. Soriente. Solving revocation with efficient update of anonymous credentials. In *SCN*, pages 454–471, 2010.
- [26] J. L. Camenisch. *Group Signature Schemes and Payment Systems Based on the Discrete Logarithm Problem*. PhD thesis, ETH Zürich, 1998. Diss. ETH No. 12520, Hartung Gorre Verlag, Konstanz.
- [27] D. Catalano and D. Fiore. Vector commitments and their applications. Cryptology ePrint Archive, Report 2011/495, 2011.
- [28] D. Chaum and T. P. Pedersen. Wallet databases with observers. In *CRYPTO*, pages 89–105, 1992.
- [29] E. F. Codd. A relational model of data for large shared data banks. *Commun. ACM*, 13(6):377–387, 1970.
- [30] R. Cramer. *Modular Design of Secure yet Practical Cryptographic Protocols*. PhD thesis, University of Amsterdam, 1997.
- [31] R. Cramer and I. Damgård. Zero-knowledge proofs for finite field arithmetic; or: Can zero-knowledge be for free? In *CRYPTO*, pages 424–441, 1998.
- [32] R. Cramer, I. Damgård, and B. Schoenmakers. Proofs of partial knowledge and simplified design of witness hiding protocols. In *CRYPTO*, pages 174–187, 1994.
- [33] I. Damgård. On Σ -protocols, 2002. Available at <http://www.daimi.au.dk/~ivan/Sigma.ps>.
- [34] I. Damgård and E. Fujisaki. An integer commitment scheme based on groups with hidden order. *IACR Cryptology ePrint Archive*, 2001:64, 2001.
- [35] G. Danezis and B. Livshits. Towards ensuring client-side computational integrity. In *CCSW*, pages 125–130, 2011.
- [36] G. Danezis, M. Kohlweiss, and A. Rial. Differentially private billing with rebates. In *Information Hiding*, pages 148–162, 2011.
- [37] C. Dwork. Differential privacy: A survey of results. *Theory and Applications of Models of Computation*, pages 1–19, 2008.
- [38] U. Feige and A. Shamir. Witness indistinguishable and witness hiding protocols. In *STOC*, pages 416–426, 1990.
- [39] U. Feige, A. Fiat, and A. Shamir. Zero knowledge proofs of identity. In *STOC*, pages 210–217, 1987.
- [40] A. Fiat and A. Shamir. How to prove yourself: Practical solutions to identification and signature problems. In *CRYPTO*, pages 186–194, 1986.
- [41] M. Fredrikson and B. Livshits. Z0: An optimizing distributing zero-knowledge compiler. 2013. MSR Technical report.
- [42] T. Freeman and F. Pfenning. Refinement types for ML. In *Programming Language Design and Implementation (PLDI'91)*, pages 268–277. ACM, 1991.
- [43] E. Fujisaki and T. Okamoto. Statistical zero knowledge protocols to prove modular polynomial relations. In *CRYPTO*, pages 16–30, 1997.
- [44] F. D. Garcia, E. R. Verheul, and B. Jacobs. Cell-based roadpricing. In *EuroPKI*, pages 106–122, 2011.
- [45] I. Goldberg. Natural zero-knowledge embedding in c++. Personal communication, October 2011.
- [46] O. Goldreich, S. Micali, and A. Wigderson. How to prove all np-statements in zero-knowledge, and a methodology of cryptographic protocol design. In *CRYPTO*, pages 171–185, 1986.
- [47] S. Goldwasser, S. Micali, and C. Rackoff. The knowledge complexity of interactive proof systems. *SIAM J. Comput.*, 18(1):186–208, 1989.
- [48] M. Jawurek, M. Johns, and F. Kerschbaum. Plug-in privacy for smart metering billing. In *PETS*, pages 192–210, 2011.
- [49] D. Malkhi, N. Nisan, B. Pinkas, and Y. Sella. Fairplay - a secure two-party computation system. In *USENIX Security*, pages 287–302, 2004.
- [50] U. M. Maurer. Unifying zero-knowledge proofs of knowledge. In B. Preneel, editor, *AFRICACRYPT*, volume 5580, pages 272–286. Springer, 2009.
- [51] S. Meiklejohn, C. C. Erway, A. Küpcü, T. Hinkle, and A. Lysyanskaya. ZKPDL: A language-based system for efficient zero-knowledge proofs and electronic cash. In *USENIX Security Symposium*, pages 193–206, 2010.
- [52] T. Okamoto. Provably secure and practical identification schemes and corresponding signature schemes. In *CRYPTO*, volume 740, pages 31–53. Springer, 1992.
- [53] B. Parno, C. Gentry, J. Howell, and M. Raykova. Pinocchio: Nearly practical verifiable computation. In *IEEE Symposium on Security and Privacy*, 2013.
- [54] T. P. Pedersen. Non-interactive and information-theoretic secure verifiable secret sharing. In *CRYPTO '92*, volume 576 of *LNCS*, pages 129–140, 1992.
- [55] A. Rial and G. Danezis. Privacy-preserving smart metering. In *WPES*, pages 49–60, 2011.
- [56] T. Ristenpart, G. Maganis, A. Krishnamurthy, and T. Kohno. Privacy-preserving location tracking of lost or stolen devices: Cryptographic techniques and replacing trusted third parties with dhds. In *17th USENIX Security Symposium*, pages 275–290, 2008.
- [57] C.-P. Schnorr. Efficient signature generation by smart cards. *J. Cryptology*, 4(3):161–174, 1991.
- [58] N. Swamy, J. Chen, C. Fournet, P.-Y. Strub, K. Bhargavan, and J. Yang. Secure distributed programming with value-dependent types. In *ICFP*, pages 266–278, 2011.
- [59] M. Tompa and H. Woll. Random self-reducibility and zero knowledge interactive proofs of possession of information. In *FOCS*, pages 472–482, 1987.
- [60] C. Troncoso, G. Danezis, E. Kosta, and B. Preneel. PriPAYD: privacy friendly pay-as-you-drive insurance. In P. Ning and T. Yu, editors, *WPES*, pages 99–107. ACM, 2007.
- [61] C. Troncoso, G. Danezis, E. Kosta, J. Balasch, and B. Preneel. Priipayd: Privacy-friendly pay-as-you-drive insurance. *IEEE Trans. Dependable Sec. Comput.*, 8(5):742–755, 2011.
- [62] H. Wee. Zero knowledge in the random oracle model, revisited. In *ASIACRYPT*, pages 417–434, 2009.

First stage:

- $\llbracket x : \tau, \rho \rrbracket_1 = x : \tau, \llbracket \rho \rrbracket_1$ when x public (including all group elements)
- $\llbracket x : \tau, \rho \rrbracket_1 = x : \tau, t_x : x \text{ witness}, \llbracket \rho \rrbracket_1$ when x private int or num
- $\llbracket \Gamma \vdash e \rrbracket_1 = H, a, e$ when e public expression, that is, whose variables are all public in Γ .
- $\llbracket \Gamma \vdash e \rrbracket_1 = \text{let } C = e \text{ in extend } H C, (a, C), C$ when $\Gamma \vdash e : \text{elt}_G$ and e is not public
- $\llbracket \Gamma \vdash a_0 + \sum_{i=1}^n a_i * x_i \rrbracket_1 = H, a, a_0 + \sum_{i=1}^n a_i * x_i, \sum_{i=1}^n a_i * t_{x_i}$ when the x_i are private and the a_i public:
 $(\Gamma(a_i) = \text{pub num})_{i=0..n}, (\Gamma(x_i) = \text{num})_{i=1..n}$
- $\llbracket \Gamma \vdash e \rrbracket_1 = \text{let } a, \rho = e \text{ in } (\text{let } t_{x_i} = \text{random}() \text{ in })_{x_i} H, (a, [\llbracket \rho \rrbracket_1], [\llbracket \rho \rrbracket_1])$ when $\Gamma \vdash e : \rho$ non-linear private expression (including assoc, random, opening...) and x_i ranges over the private variables bound in ρ
- $\llbracket \Gamma \vdash \text{assert } e_C =_G e_x \rrbracket_1 = \text{extend } H e_t, a, e$ when e_C public and e_x algebraic on private exponents
- $\llbracket \Gamma \vdash \downarrow x \rrbracket_1 = \text{let } a = a, x \text{ in extend } (\text{extend } H g^x) g^{t_x}, a, x$
- $\llbracket \Gamma \vdash \text{let } \rho = e \text{ in } e_0 \rrbracket_1 = \text{let } H, a, [\llbracket \rho \rrbracket_1] = \llbracket \Gamma \vdash e \rrbracket_1 \text{ in } \llbracket \Gamma, \rho \vdash e_0 \rrbracket_1$

Second stage:

- $\llbracket \delta, \rho \rrbracket_2 = \text{let } a, [\llbracket \rho \rrbracket_1] = a \text{ in } [\llbracket \delta \rrbracket_2], [\text{let } r_x = t_x - c * x \text{ in }]_x [\llbracket \rho \rrbracket_v]$ where x ranges over the private variables bound in ρ

- $\llbracket \delta, \delta' \text{ table} \rrbracket_2 = \text{let } a, A = a \text{ in } [\llbracket \delta \rrbracket_2], \text{map } (\delta' \rightarrow [\llbracket \delta' \rrbracket_2]) A$
- $\llbracket \varepsilon \rrbracket_2 = \varepsilon$

```

 $\llbracket \theta \rightarrow e \rrbracket_{\text{PROVER}} = \llbracket \theta \rrbracket_D \rightarrow$ 
   $\text{let } H = H_0 \text{ in let } a = () \text{ in}$ 
  // hash and prove commitments for all private inputs (omitted)
   $\text{let } H : \text{hash}, a : \delta, r = \llbracket \theta \vdash e \rrbracket_1 \text{ in}$ 
   $\text{let } c = \text{finalize } H \text{ in}$ 
   $\llbracket \theta, r \rrbracket_{\text{pub}}, \llbracket \delta \rrbracket_2, c$ 

```

Figure 6: Prover Translation (see full paper for **map** and **fold**)

$\llbracket x : \tau, \rho \rrbracket_v = x : \tau, \llbracket \rho \rrbracket_v$ when x public (including all group elements)

$\llbracket x : \tau, \rho \rrbracket_v = r_x : (c, x) \text{ response}, \llbracket \rho \rrbracket_v$ when x private

- $\llbracket \Gamma \vdash e \rrbracket_v = H, a, e$ when e public expression, that is, whose variables are all public in Γ .
- $\llbracket \Gamma \vdash e \rrbracket_v = \text{let } C, a = a \text{ in extend } H C, a, C$ when $\Gamma \vdash e : \text{elt}_G$ and e is not public
- $\llbracket \Gamma \vdash a_0 + \sum_{i=1}^n a_i * x_i \rrbracket_v = H, a, -c * a_0 + \sum_{i=1}^n a_i * r_{x_i}$ when the x_i are private and the a_i public:
 $(\Gamma(a_i) = \text{pub num})_{i=0..n}, (\Gamma(x_i) = \text{num})_{i=1..n}$
- $\llbracket \Gamma \vdash e \rrbracket_v = \text{let } a, [\llbracket \rho \rrbracket_v] = a \text{ in } H, a, [\llbracket \rho \rrbracket_v]$ when $\Gamma \vdash e : \rho$ non-linear private-exponent expression (including assoc, random, opening...) and ρ binds private exponents and public elements
- $\llbracket \Gamma \vdash \text{assert } e_C =_G e_x \rrbracket_v = \text{extend } H ((e_C)^c *_G \llbracket e_x \rrbracket_v), a, \varepsilon$ when e_x algebraic on private exponents
- $\llbracket \Gamma \vdash \downarrow x \rrbracket_v = \text{let } x, a = a \text{ in }$
 $\text{extend } (\text{extend } H g^x) g^{c*x+r_x}, a, x$
- $\llbracket \Gamma \vdash \text{let } \rho = e \text{ in } e_0 \rrbracket_v = \text{let } H, a, [\llbracket \rho \rrbracket_v] = \llbracket \Gamma \vdash e \rrbracket_1 \text{ in } \llbracket \Gamma, \rho \vdash e_0 \rrbracket_v$

```

 $\llbracket \theta \rightarrow e \rrbracket_{\text{VERIFIER}} = \llbracket \theta, r \rrbracket_{\text{pub}'}, a, c \rightarrow$ 
  // check plain signatures, hash commitments into  $H$ ,
  // and check commitment proofs for all private inputs (omitted)
   $\text{let } H = H_0 \text{ in}$ 
   $\text{let } H, a, r = \llbracket \Gamma \vdash e \rrbracket_v \text{ in}$ 
   $\text{check } c = \text{finalize } H;$ 
   $r$ 

```

Figure 7: Verifier Translation (see full paper for **map** and **fold**)

DupLESS: Server-Aided Encryption for Deduplicated Storage

Mihir Bellare

University of California, San Diego

Sriram Keelvedhi

University of California, San Diego

Thomas Ristenpart

University of Wisconsin–Madison

Abstract

Cloud storage service providers such as Dropbox, Mozy, and others perform deduplication to save space by only storing one copy of each file uploaded. Should clients conventionally encrypt their files, however, savings are lost. Message-locked encryption (the most prominent manifestation of which is convergent encryption) resolves this tension. However it is inherently subject to brute-force attacks that can recover files falling into a known set. We propose an architecture that provides secure deduplicated storage resisting brute-force attacks, and realize it in a system called DupLESS. In DupLESS, clients encrypt under message-based keys obtained from a key-server via an oblivious PRF protocol. It enables clients to store encrypted data with an existing service, have the service perform deduplication on their behalf, and yet achieves strong confidentiality guarantees. We show that encryption for deduplicated storage can achieve performance and space savings close to that of using the storage service with plaintext data.

1 Introduction

Providers of cloud-based storage such as Dropbox [3], Google Drive [7], and Mozy [63] can save on storage costs via deduplication: should two clients upload the same file, the service detects this and stores only a single copy. The savings, which can be passed back directly or indirectly to customers, are significant [50, 61, 74] and central to the economics of the business.

But customers may want their data encrypted, for reasons ranging from personal privacy to corporate policy to legal regulations. A client could encrypt its file, under a user’s key, before storing it. But common encryption modes are randomized, making deduplication impossible since the SS (Storage Service) effectively always sees different ciphertexts regardless of the data. If a client’s encryption is deterministic (so that the same file will al-

ways map to the same ciphertext) deduplication is possible, but only for that user. Cross-user deduplication, which allows more storage savings, is not possible because encryptions of different clients, being under different keys, are usually different. Sharing a single key across a group of users makes the system brittle in the face of client compromise.

One approach aimed at resolving this tension is message-locked encryption (MLE) [18]. Its most prominent instantiation is convergent encryption (CE), introduced earlier by Douceur et al. [38] and others (c.f., [76]). CE is used within a wide variety of commercial and research SS systems [1, 2, 5, 6, 8, 12, 15, 32, 33, 55, 60, 66, 71, 78, 79]. Letting M be a file’s contents, hereafter called the message, the client first computes a key $K \leftarrow H(M)$ by applying a cryptographic hash function H to the message, and then computes the ciphertext $C \leftarrow E(K, M)$ via a deterministic symmetric encryption scheme. The short message-derived key K is stored separately encrypted under a per-client key or password. A second client B encrypting the same file M will produce the same C , enabling deduplication.

However, CE is subject to an inherent security limitation, namely susceptibility to offline brute-force dictionary attacks. Knowing that the target message M underlying a target ciphertext C is drawn from a dictionary $S = \{M_1, \dots, M_n\}$ of size n , the attacker can recover M in the time for $n = |S|$ off-line encryptions: for each $i = 1, \dots, n$, it simply CE-encrypts M_i to get a ciphertext denoted C_i and returns the M_i such that $C = C_i$. (This works because CE is deterministic and keyless.) Security is thus only possible when the target message is drawn from a space too large to exhaust. We say that such a message is *unpredictable*.

Bellare, Keelvedhi, and Ristenpart [18] treat MLE formally, providing a definition (semantic-security for unpredictable messages) to capture the best possible security achievable for MLE schemes in the face of the inherent limitation noted above. The definition is based

on previous ones for deterministic encryption, a primitive subject to analogous inherent limitations [16, 17, 27]. The authors go on to show that CE and other mechanisms achieve their definition in the random-oracle model.

The unpredictability assumption. The above-mentioned work puts security on a firm footing in the case messages are unpredictable. In practice, however, security only for unpredictable data may be a limitation for, and threat to, user privacy. We suggest two main reasons for this. The first is simply that data is often predictable. Parts of a file’s contents may be known, for example because they contain a header of known format, or because the adversary has sufficient contextual information. Some data, such as very short files, are inherently low entropy. This has long been recognized by cryptographers [43], who typically aim to achieve security regardless of the distribution of the data.

The other and perhaps more subtle fear with regard to the unpredictability assumption is the difficulty of validating it or testing the extent to which it holds for “real” data. When we do not know how predictable our data is to an adversary, we do not know what, if any, security we are getting from an encryption mechanism that is safe only for unpredictable data. These concerns are not merely theoretical, for offline dictionary attacks are recognized as a significant threat to CE in real systems [77] and are currently hindering deduplication of outsourced storage for security-critical data.

This work. We design and implement a new system called DupLESS (Duplicateless Encryption for Simple Storage) that provides a more secure, easily-deployed solution for encryption that supports deduplication. In DupLESS, a group of affiliated clients (e.g., company employees) encrypt their data with the aid of a key server (KS) that is separate from the SS. Clients authenticate themselves to the KS, but do not leak any information about their data to it. As long as the KS remains inaccessible to attackers, we ensure high security. (Effectively, semantic security [43], except that ciphertexts leak equality of the underlying plaintexts. The latter is necessary for deduplication.) If both the KS and SS are compromised, we retain the current MLE guarantee of security for unpredictable messages.

Unlike prior works that primarily incorporate CE into new systems, our goal is to make DupLESS work transparently with existing SS systems. DupLESS therefore sits as a layer on top of existing simple storage interfaces, wrapping store, retrieve, and other requests with algorithms for encrypting filenames and data on the fly. This also means that DupLESS was built: to be as feature-compatible as possible with existing API commands, to not assume any knowledge about the systems implementing these APIs, to give performance very close to that of

using the SS without any encryption, and to achieve the same availability level as provided by the SS.

We implement DupLESS as a simple-to-use command-line client that supports both Dropbox [3] and Google Drive [7] as the SS. We design two versions of the KS protocol that clients can use while encrypting files. The first protocol uses a RESTful, HTTPS based, web interface, while the second is a custom protocol built over UDP. The first is simpler, being able to run on top of existing web servers, and the latter is optimized for latency, and capable of servicing requests at close to the (optimal) round-trip time of the network. These protocols and their implementations, which at core implement an oblivious pseudorandom function (OPRF) [64] service, may be of independent interest.

To evaluate end-to-end performance, we deploy our KS on Amazon EC2 [10] and experimentally evaluate its performance. DupLESS incurs only slight overheads compared to using the SS with plaintext data. For a 1 MB file and using Dropbox, the bandwidth overhead is less than 1% and the overhead in the time to store a file is about 17%. We compute storage overheads of as little as 4.5% across a 2 TB dataset consisting of over 2,000 highly deduplicable virtual machine file system images that we gathered from Amazon EC2. All this shows that DupLESS is practical and can be immediately deployed in most SS-using environments. The source code for DupLESS is available from [4].

2 Setting

At a high level, our setting of interest is an enterprise network, consisting of a group of affiliated clients (for example, employees of a company) using a deduplicated cloud storage service (SS). The SS exposes a simple interface consisting of only a handful of operations such as storing a file, retrieving a file, listing a directory, deleting a file, etc.. Such systems are widespread (c.f., [1, 3, 7, 11, 63]), and are often more suitable to user file backup and synchronization applications than richer storage abstractions (e.g., SQL) [37, 69] or block stores (c.f., [9]). An example SS API, abstracted from Dropbox, is detailed in Figure 5 (Section 6). The SS performs deduplication along file boundaries, meaning it checks if the contents of two files are the same and deduplicates them if so, by storing only one of them.

Clients have access to a key server (KS), a semi-trusted third party which will aid in performing deduplicable encryption. We will explain further the role of the KS below. Clients are also provisioned with per-user encryption keys and credentials (e.g., client certificates).

Threat model. Our goal is to protect the confidentiality of client data. Attackers include those that gain access to the SS provider’s systems (including malicious insiders working at the provider) and external attackers with access to communication channels between clients and the KS or SS. Security should hold for all files, not just unpredictable ones. In other words, we seek semantic security, leaking only equality of files to attackers.

We will also be concerned with compromise resilience: the level of security offered by the scheme to legitimate clients should degrade gracefully, instead of vanishing, should other clients or even the KS be compromised by an attacker. Specifically, security should hold at least for unpredictable files (of uncompromised clients) when one or more clients are compromised and when the KS is compromised.

We will match the availability offered by the SS, but explicitly do not seek to ensure availability in the face of a malicious SS: a malicious provider can always choose to delete files. We will, however, provide protection against a malicious SS that may seek to tamper with clients’ data, or mount chosen-ciphertext attacks, by modifying stored ciphertexts.

Malicious clients can take advantage of an SS that performs client-side deduplication to mount a side-channel attack [46]. This arises because one user can tell if another user has already stored a file, which could violate the latter’s privacy.¹ We will not introduce such side-channels. A related issue is that client-side deduplication can be abused to perform illicit file transfers between clients [73]. We will ensure that our systems can work in conjunction with techniques such as proofs-of-ownership [45] that seek to prevent such issues.

We will not explicitly target resistance to traffic analysis attacks that abuse leakage of access patterns [48] or file lengths [24, 31, 40, 47, 59, 65, 72], though our system will be compatible with potential countermeasures.

Our approaches may be used in conjunction with existing mechanisms for availability auditing [13, 41, 51, 70] or file replication across multiple services [26]. (In the latter case, our techniques will enable each service to independently perform deduplication.)

Design goals. In addition to our security goals, the system we build will meet the following functionality properties. The system will be *transparent*, both from the perspective of clients and the SS. This means that the system will be backwards-compatible, work within existing SS APIs, make no assumptions about the implementation details of the SS, and have performance closely matching that of direct use of the SS. In normal operation and for all clients of a particular KS, the space required to store

all encrypted data will match closely the space required when storing plaintext data. The system should never reduce storage availability, even when the KS is unavailable or under heavy load. The system will not require *any* client-side state beyond a user’s credentials. A user will be able to sit down at any system, provide their credentials, and synchronize their files. We will however allow client-side caching of data to improve performance.

Related approaches. Several works have looked at the general problem of enterprise network security, but none provide solutions that meet all requirements from the above threat model. Prior works [42, 53, 54, 58, 75] which build a secure file system on top of a flat outsourced storage server break deduplication mechanisms and are unfit for use in our setting. Convergent encryption (CE) based solutions [8, 71], as we explored in the Introduction, provide security only for unpredictable messages even in the best case, and are vulnerable to brute-force attacks. The simple approach of sharing a secret key across clients with a deterministic encryption scheme [16, 68] fails to achieve compromise resilience. Using CE with an additional secret shared across all clients [76] does not work for the same reason.

3 Overview of DupLESS

DupLESS starts with the observation that brute-force ciphertext recovery in a CE-type scheme can be dealt with by using a key server (KS) to derive keys, instead of setting keys to be hashes of messages. Access to the KS is preceded by authentication, which stops external attackers. The increased cost slows down brute-force attacks from compromised clients, and now the KS can function as a (logically) single point of control for implementing rate-limiting measures. We can expect that by scrupulous choice of rate-limiting policies and parameters, brute-force attacks originating from compromised clients will be rendered less effective, while normal usage will remain unaffected.

We start by looking at secret-parameter MLE, an extension to MLE which endows all clients with a system-wide secret parameter sk (see Section 4). The rationale here is that if sk is unknown to the attacker, a high level of security can be achieved (semantic security, except for equality), but even if sk is leaked, security falls to that of regular MLE. A server-aided MLE scheme then is a transformation where the secret key is restricted to the KS instead of being available to all clients. One simple approach to get server-aided MLE is to use a PRF F , with a secret key K that never leaves the KS. A client would send a hash H of a file to the KS and receive back a *message-derived* key $K' \leftarrow F(K, H)$. The other steps are as in CE. However, this approach proves unsatisfying

¹The reader might be interested to note that our experience with the Dropbox client suggests this side channel still exists.

from a security perspective. The KS here becomes a single point of failure, violating our goal of compromise resilience: an attacker can obtain hashes of files after gaining access to the KS, and can recover files with brute-force attacks. Instead, DupLESS employs an oblivious PRF (OPRF) protocol [64] between the KS and clients, which ensures that the KS learns nothing about the client inputs or the resulting PRF outputs, and that clients learn nothing about the key. In Section 4, we propose a new server-aided MLE scheme DupLESSMLE which combines a CE-type base with the OPRF protocol based on RSA blind-signatures [20, 29, 30].

Thus, a client, to store a file M , will engage in the RSA OPRF protocol with the KS to compute a message-derived key K , then encrypt M with K to produce a ciphertext C_{data} . The client’s secret key will be used to encrypt K to produce a key encapsulation ciphertext C_{key} . Both C_{key} and C_{data} are stored on the SS. Should two clients encrypt the same file, then the message-derived keys and, in turn, C_{data} will be the same (the key encapsulation C_{key} will differ, but this ciphertext is small). The DupLESS client algorithms are described in Section 6 along with how DupLESS handles filenames and paths.

Building a system around DupLESSMLE requires careful design in order to achieve high performance. DupLESS uses at most one or two SS API calls per operation. (As we shall see, SS API calls can be slow.) Because interacting with the KS is on the critical path for storing files, DupLESS incorporates a fast client-to-KS protocol that supports various rate-limiting strategies. When the KS is overloaded or subjected to denial-of-service attacks, DupLESS clients fall back to symmetric encryption, ensuring availability. On the client side, DupLESS introduces *dedup heuristics* (see Section 6) to determine whether the file about to be stored on the SS should be selected for deduplication, or processed with randomized encryption. For example, very small files or files considered particularly sensitive can be prevented from deduplication. We use deterministic authenticated encryption (DAE) [68] to protect, in a structure-preserving way, the path and filename associated to stored files. Here we have several choices along an efficiency/security continuum. Our approach of preserving folder structure leaks some information to the SS, but on the other hand, enables direct use of the SS-provided API for file search and moving folders.

DupLESS is designed for a simple SS API, but can be adapted to settings in which block-oriented deduplication is used, and to complex network storage and backup solutions that use NFS [62], CIFS [56] and the like, but we do not consider these further.

In the following sections we go into greater detail on the various parts of the DupLESS system, starting with the cryptographic primitives in Section 4, then moving

on to describing KS design in Section 5, and then on to the client algorithms in Section 6, followed by performance and security in Sections 7 and 8 respectively.

4 Cryptographic Primitives

A *one-time encryption scheme* SE with key space $\{0,1\}^k$ is a pair of deterministic algorithms (E, D) . Encryption E on input a key $K \in \{0,1\}^k$ and message $M \in \{0,1\}^*$ outputs a ciphertext C . Decryption D takes a key and a ciphertext and outputs a message. CTR mode using AES with a fixed IV is such a scheme. An *authenticated encryption* (AE) scheme is pair of algorithms $AE = (EA, DA)$ [19, 67]. Encryption EA takes as input a key $K \in \{0,1\}^k$, associated data $D \in \{0,1\}^*$, and message $M \in \{0,1\}^*$ and outputs a ciphertext of size $|M| + \tau_d$, where τ_d is the ciphertext stretch (typically, 128 bits). Decryption DA is deterministic; it takes input a key, associated data, and a ciphertext and outputs a message or error symbol \perp . When encryption is deterministic, we call the scheme a deterministic authenticated encryption (DAE) scheme [68]. We use the Encrypt-then-MAC [19] scheme for AE and SIV mode [68] for DAE, both with HMAC[SHA256] and CTR[AES].

Oblivious PRFs. A (verifiable) oblivious PRF (OPRF) scheme [64] consists of five algorithms $OPRF = (Kg, EvC, EvS, Vf, Ev)$, the last two deterministic. Key generation $(pk, sk) \xleftarrow{\$} Kg$ outputs a public key pk which can be distributed freely among several clients, and a secret key sk , which remains with a single entity, the server. The evaluation protocol runs as follows: on the client-side, EvC starts with an input x and ends with output y such that $y = Ev(sk, x)$, while on the server-side, EvS starts with secret key sk and ends without output. Figure 1 gives an example. Verification $Vf(pk, x, y)$ returns a boolean. Security requires that (1) when keys are picked at random, $Ev(sk, \cdot)$ outputs are indistinguishable from random strings to efficient attackers without pk , and (2) no efficient attacker, given (pk, sk) , can provide x, x', y such that $Vf(pk, x, y) = Vf(pk, x', y) = \text{true}$, or $Vf(pk, x, y) = \text{true}$ but $Ev(sk, x) \neq y$, or $Vf(pk, x, y) = \text{false}$ but $Ev(sk, x) = y$, except with negligible probability. Moreover, in the OPRF protocol, the server learns nothing about client inputs or resulting PRF outputs, and the client learns nothing about sk .

Verifiable OPRF schemes can be built from deterministic blind signatures [29]. The RSA-OPRF[G, H] scheme based on RSA blind signatures [20, 30] is described as follows. The public RSA exponent e is fixed as part of the scheme. Key generation Kg runs RSA Kg with input e to get N, d such that $ed \equiv 1 \pmod{\phi(N)}$, modulus N is the product of two distinct primes of roughly equal length and $N < e$. Then, $(N, (N, d))$ is output as

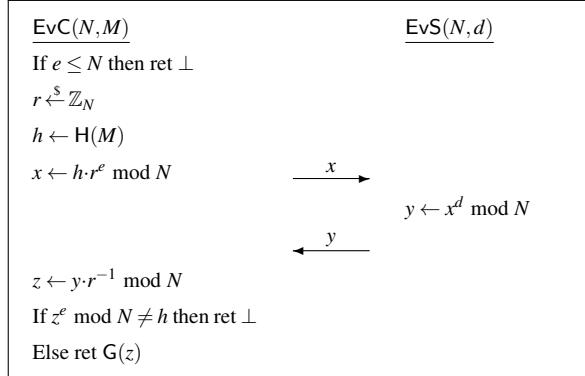


Figure 1: The RSA-OPRF protocol. The key generation Kg outputs PRF key N, d and verification key N . The client uses two hash functions $H: \{0, 1\}^* \rightarrow \mathbb{Z}_N$ and $G: \mathbb{Z}_N \rightarrow \{0, 1\}^k$.

the public key, secret key pair. The evaluation protocol (EvC, EvS) with verification Vf is shown in Figure 1. The client uses a hash function $H: \{0, 1\}^* \rightarrow \mathbb{Z}_N$ to first hash the message to an element of \mathbb{Z}_N , and then blinds the result with a random group element r raised to the e -th power. The resulting blinded hash, denoted x , is sent to the KS. The KS signs it by computing $y \leftarrow x^d \text{ mod } N$, and sends back y . Verification then removes the blinding by computing $z \leftarrow yr^{-1} \text{ mod } N$, and then ensures that $z^e \text{ mod } N$ is indeed equal to $H(M)$. Finally, the output of the PRF is computed as $G(z)$, where $G: \mathbb{Z}_N \rightarrow \{0, 1\}^k$ is another hash function.

This protocol can be shown to be secure as long as the map $f_e: \mathbb{Z}_N^* \rightarrow \mathbb{Z}_N^*$, defined by $f_e(x) = x^e \text{ mod } N$ for all $x \in \mathbb{Z}_N^*$, is a permutation on \mathbb{Z}_N^* , which is assured by $\gcd(\varphi(N), e) = 1$. In particular, this is true if the server creates its keys honestly. However, in our setting, the server can cheat while generating the keys, in an attempt to glean something about $H(M)$. This is avoided by requiring that $N < e$, which will be verified by the client. Given that e is prime, this standard technique ensures that $\gcd(\varphi(N), e) = 1$ even if N is maliciously generated, and thus ensures that f_e is a permutation. Since f_e is a permutation and the client checks the signature, even a malicious server cannot force the output $K = G(z)$ to be a fixed value or force two keys output for distinct messages to collide, as long as G is collision-resistant.

MLE. A deterministic Message-Locked Encryption (MLE) scheme is a tuple $MLE = (P, K, E, D)$ of algorithms, the last three deterministic². Parameter generation outputs a public parameter $P \xleftarrow{\$} P$, common to all users of a system. To encrypt M , one generates the message-derived key $K \leftarrow K(P, M)$ and ciphertext

²We drop the tag generation algorithm which was part of the original MLE formulation [18]. Since we restrict attention to deterministic MLE schemes, we let ciphertexts work as tags.

$C \leftarrow E(P, K, M)$. Decryption works as $M \leftarrow D(P, K, C)$. Security requires that no efficient attacker can distinguish ciphertexts of unpredictable messages from random strings except with negligible probability. Convergent encryption (CE) [38] is the most prominent MLE scheme. We use CE with parameters P set to random 128-bit strings, key generation returning the first 128 bits of $SHA256(P \| M)$ on input M , and encryption and decryption being implemented with CTR[AES].

In a secret-parameter MLE scheme SPMLE, parameter generation outputs a (system-wide) secret parameter sk along with a public parameter P . This secret parameter, which is provided to all legitimate users, is used to generate message-derived keys as $K \leftarrow K(P, sk, M)$. In a server-aided MLE scheme, the secret parameter is provided only to a KS. Clients interact with the KS to obtain message-derived keys. A simple way of doing this of course is that clients can send the messages to the KS which would then reply with message-derived keys. But, as we saw in the previous section, this is undesirable in the DupLESS setting, as the KS now becomes a single point of failure. Instead, we propose a new server-aided MLE scheme DupLESSMLE combining RSA-OPRF[G, H] = (Kg, EvC, EvS, Vf, Ev) and CTR[AES]. Here parameter generation runs Kg to get $(N, (N, d))$, then outputs N as the public parameter and (N, d) as the secret parameter (recall that e is fixed as part of the scheme). From a message M , a key K is generated as $K \leftarrow Ev((N, d), M) = G(H(M)^d \text{ mod } N)$ by interacting with the KS using EvC and EvS . Encryption and decryption work as in CE, with CTR[AES]. We use RSA1024 with full-domain-hash using SHA256 in the standard way [22] to get H and G .

The advantage of server-aided MLE is the prospect of multi-tiered security. In DupLESSMLE in particular, when the adversary does not have access to the KS (but has access to ciphertexts and OPRF inputs and outputs), it has no knowledge of sk , and semantic-security similar to deterministic SE schemes follows, from the security of RSA-OPRF[G, H] and CTR[AES]. When the attacker has access to the KS additionally, attacks are still constrained to be online and consequently slow, and subject to rate-limiting measures that the KS imposes. Security here relies on implementing the OPRF protocol correctly, and ensuring that the rate-limiting measures cannot be circumvented. We will analyze this carefully in Section 5. Even when sk is compromised to the attacker, DupLESSMLE provides the usual MLE-style security, conditioned on messages being unpredictable. Moreover, we are guaranteed that the clients' inputs are hidden from the KS, even if the KS is under attack and deviates from its default behavior, from the security of the RSA-OPRF[G, H] protocol.

5 The DupLESS KS

In this section we describe the KS side of DupLESS. This includes protocols for client-KS interaction which realize RSA-OPRF[G, H], and rate limiting strategies which limit client queries to slow down online brute-force attacks. We seek low-latency protocols to avoid degrading performance, which is important because the critical path during encryption includes interaction with a KS. Additionally, the protocol should be light-weight, letting the KS handle a reasonably high request volume.

We describe two protocols: OPRFv1, and OPRFv2, which rely on a CA providing the KS and clients with verifiable TLS certificates. In the following, we assume that each client has a unique certificate, and that clients can be identified by their certificates. Of course, the protocols can be readily converted to work with other authentication frameworks. We believe our OPRF protocols to be faster than previous implementations [36], and given the support for rate-limiting, we expect that they will be useful in other applications using OPRFs.

HTTPS based. In the first protocol, OPRFv1, all communication with the KS happens over HTTPS. The KS exposes an interface with two procedures: `KSInit` and `KSReq`. The first time a client uses the KS, it makes a `KSInit` request to obtain, and then locally cache, the KS’s OPRF public key. Here the client must perform any necessary checks of the public key, which for our scheme is simply that $e > N$. When the client wants a key, say for a file it is about to upload, the client will make use of the `KSReq` interface, by sending an HTTPS POST of the blinded hash value. Now, the KS checks request validity, and performs rate-limiting measures which we describe below. Then, the KS computes the signature over the blinded hash value, and sends this back over the established HTTPS channel.

OPRFv1 has the benefit of extreme simplicity. With 20 lines of code (excluding rate limiting logic) in the form of a Web-Server Gateway Interface (WSGI) Python module, one can run the KS on top of most web servers. We used Apache 2.0 in our implementation.

Unfortunately, while simple, this is a high latency solution, as it requires four full round trips across the network (1 for TCP handshake, 2 for the TLS handshake, 1 for the HTTP request) to perform `KSReq`. While sub-second latency is not always critical (e.g., because of poor SS performance or because the KS and clients share a LAN), it will be critical in many settings, and so we would like to do better.

UDP based. We therefore turn to OPRFv2, which removes the slow per-request handshakes from the critical path of encryption. Here, the `KSInit` procedure starts with a TLS handshake with mutual authentication, initi-

ated by a client. The KS responds immediately following a valid handshake with the OPRF public key pk , a TLS identifier of a hash function H (by default SHA-256), a random session identifier $S \in \{0, 1\}^{128}$, and a random session key $K_S \in \{0, 1\}^k$ (we set $k = 128$ in our implementations). We shave off one round trip from `KSInit` by responding immediately, instead of waiting for an HTTP message as in OPRFv1. The KS also associates a sequence number with this session, initialized to zero. Internally the KS maintains two tables, one mapping session identifiers with keys, and a second which keeps track of sequence numbers. Each session lasts for a fixed time period (currently 20 minutes in our implementation) and table entries are removed after the session expires. The client caches pk, S and K_S locally and initializes a sequence number $N = 0$.

To make an OPRF request `KSReq` on a blinded value X , the client first increments the sequence number $N \leftarrow N + 1$, then computes a MAC tag using its session key, as $T \leftarrow \text{HMAC}[H](K_S, S \| N \| X)$ and sends the concatenation $S \| N \| X \| T$ to the KS in a single UDP packet. The KS recovers S, N, X and T and looks up K_S and N_S . It ensures that $N > N_S$ and checks correctness of the MAC T . If the packet is malformed or if some check fails, then the KS drops the packet without further action. If all the checks pass, the KS sends the OPRF protocol response in a single UDP packet.

The client waits for time t_R after sending a `KSReq` packet before triggering timeout behavior. In our implementation, this involves retrying the same request twice more with time t_R between the tries, incrementing the sequence number each time. After three attempts, the client will try to initiate a new session, again timing out after t_R units. If this step fails, the client believes the KS to be offline. This timeout behavior is based on DNS, and following common parameters, we set $t_R = 1$ second.

We implemented OPRFv2 in Python. It comes to 165 lines of code as indicated by the cloc utility, the bulk of which is in fact the rate limiting logic discussed below. Our current KS implementation is not yet optimized. For example it spawns and kills a new thread for each connection request (as opposed to keeping a pool of children around, as in Apache). Nevertheless the implementation is fully functional and performs well.

Rate limiting KS requests. We explore approaches for per-client rate limiting. In the first approach, called Bounded, the KS sets a bound q on the total number of requests a client can make during a fixed time interval t_E , called an epoch. Further queries by the client will be ignored by the KS, until the end of the epoch. Towards keeping the KS simple, a single timer controls when epochs start and end, as opposed to separate timers for each client that start when their client performs a ses-

sion handshake. It follows that no client can make more than $2q$ queries within a t_E -unit time period.

Setting q gives rise to a balancing act between online brute-force attack speed and sufficiently low-latency KS requests, since a legitimate client that exceeds its budget will have to wait until the epoch ends to submit further requests. However, when using these OPRF protocols within DupLESS, we also have the choice of exploiting the trade-off between dedupability and online brute-force speed. This is because we can build clients to simply continue with randomized encryption when they exceed their budgets, thereby alleviating KS availability issues for a conservative choice of q .

In any case, the bound q and epoch duration should be set so as to not affect normal KS usage. Enterprise network storage workloads often exhibit temporal self-similarity [44], meaning that they are periodic. In this case, a natural choice for the epoch duration is one period. The bound q can be set to the expected number of client requests plus some buffer (e.g., one or more standard deviations). Administrators will need to tune this for their deployment; DupLESS helps ease this burden by its tolerance of changes to q as discussed above.

We also considered two other mechanisms for rate limiting. The fixed delay mechanism works by introducing an artificial delay t_D before the KS responds to a client’s query. This delay can either be a system-wide constant, or be set per client. Although this method is the simplest to implement, to get good brute-force security, the delay introduced would have to be substantially high and directly impacts latency. The exponential delay mechanism starts with a small delay, and doubles this quantity after every query. The doubling stops at an upper limit t_U . The server maintains synchronized epochs, as in the bounded approach, and checks the status of active clients after each epoch. If a client makes no queries during an entire epoch, its delay is reset to the initial value. In both these approaches, the server maintains an active client list, which consists of all clients with queries awaiting responses. New queries from clients in the active client list are dropped. Client timeout in fixed delay is $\max(t_D, t_R)$ and in exponential delay it is $\max(t_U, t_R)$.

To get a sense of how such rate-limiting mechanisms might work in real settings, we estimate the effects on brute-force attacks by deriving parameters from the characteristics of a workload consisting of about 2,700 computers running on an enterprise network at NetApp, as reported in [57]. The workload is periodic, with similar patterns every week. The clients together make 1.65 million write queries/week, but the distribution is highly skewed, and a single client could potentially be responsible for up to half of these writes. Let us be conservative and say that our goal is to ensure that clients making at most 825,000 queries/week should be unaffected by rate-

Mechanism	Rate formula	NetApp Scenario
Bounded	$2q/t_E$	2.73
Fixed delay	$1/t_D$	1.36
Exp. delay	$2t_E/t_U$	2.73
None	3,200	3,200
Offline	120–12000	120–12000

Figure 2: Comparing brute-force rates in queries per second for different rate limiting approaches, no rate limiting (None), and hashes as computed using SHA-256 (Offline). The first column is the formula used to derive the rate as a function of the request limit q , epoch duration t_E , delay t_D , and upper limit t_U . The second column is the rates as for the NetApp workload. The None row does not include offline computation cost.

limiting. We set the epoch duration t_E as one week and query bound as $q = 825k$. The fixed delay would need to be set to 730 milliseconds (in order to facilitate 825k requests in one week), which is also the upper limit t_U for the exponential technique.

The maximum query rates in queries per second that an attacker who compromised a client can achieve are given in Figure 2, along with the formulas used to calculate them. The “None” row, corresponding to no rate limiting, gives as the rate the highest number of replies per second seen for OPRFv2 in the throughput experiment above. The offline brute force rate was measured by running Intel’s optimized version of SHA256 [49] to get processing speed as 120 MBps on our client system, whose 7200-RPM hard disk has peak read speed of 121MBps (as measured by hdparm). The range then varies from the number of hashes per second for 1 MB files up to the number of hashes per second for 1 KB files, assuming just a single system is used.

Despite being generous to offline brute-force attacks (by just requiring computation of a hash, not considering parallelization, and not including in the online attacks any offline computational costs), the exercise shows the huge benefit of forcing brute-force attackers to query the KS. For example, the bounded rate limiting mechanism slows down brute-force attacks by anywhere from 43x for large files up to 4,395x for small files. If the attacker wants to identify a 1KB file which was picked at random from a set S of 2^{25} files, then the offline brute-force attack requires less than an hour, while the bounded rate limited attack requires more than twenty weeks.

We note that bounded rate-limiting is effective only if the file has enough unpredictability to begin with. If $|S| < q = 825k$, then the online brute-force attack will be slowed down only by the network latency, meaning that it will proceed at one-fourth the offline attack rate. Moreover, parallelization will speed up both online and offline attacks, assuming that this is permitted by the KS.

Operation	Latency (ms)
OPRFv1 KSReq (Low KS load)	374 ± 34
OPRFv2 KSInit	278 ± 56
OPRFv2 KSReq (Low KS load)	83 ± 16
OPRFv2 KSReq (Heavy KS load)	118 ± 37
Ping (1 RTT)	78 ± 01

Figure 3: The median time plus/minus one standard deviation to perform KSInit and KSReq operations over 1000 trials. Low KS load means the KS was otherwise idle, whereas Heavy KS load means it was handling 3000 queries per second.

Performance. For the OPRF, as mentioned in Section 4, we implement RSA1024 with full-domain-hash using SHA256 in the standard way [22]. The PKI setup uses RSA2048 certificates and we fix the ECDHE-RSA-AES128-SHA ciphersuite for the handshake. We set up the two KS implementations (OPRFv1 and OPRFv2) on Amazon EC2 m1.large instances. The client machine, housed on a university LAN, had an x86-64 Intel Core i7-970 processor with a clockspeed fixed at 3201 MHz.

Figure 3 depicts the median times, in milliseconds, of various operations for the two protocols. OPRFv2 significantly outperforms OPRFv1, due to the reduced number of round trip times. On a lightly loaded server, a KS request requires almost the smallest possible time (the RTT to the KS). The time under a heavy KS load was measured while a separate m1.large EC2 instance sent 3000 requests per second. The KS request time for OPRFv2 increases, but is still three times faster than OPRFv1 for a low KS load. Note that the time reported here is only over successful operations; ones that timed out three times were excluded from the median.

To understand the drop rates for the OPRFv2 protocol on a heavily loaded server and, ultimately, the throughput achievable with our (unoptimized) implementation, we performed the following experiment. A client sent $100i$ UDP request packets per second (qps) until a total of 10,000 packets are sent, once for each of $1 \leq i \leq 64$. The number of requests responded to was then recorded. The min/max/mean/standard deviation over 100 trials are shown in Figure 4. At rates up to around 3,000 queries per second, almost no packets are dropped. We expect that with further (standard) performance optimizations this can be improved even further, allowing a single KS to support a large volume of requests with very occasional single packet drops.

Security of the KS protocols. Adversarial clients can attempt to snoop on, as well as tamper with, communications between (uncompromised) clients and the KS. With rate-limiting in play, adversaries can also attempt to launch denial-of-service (DOS) attacks on uncompromised clients, by spoofing packets from such clients. Finally, adversaries might try to circumvent rate-limiting. A secure protocol must defend against all these threats.

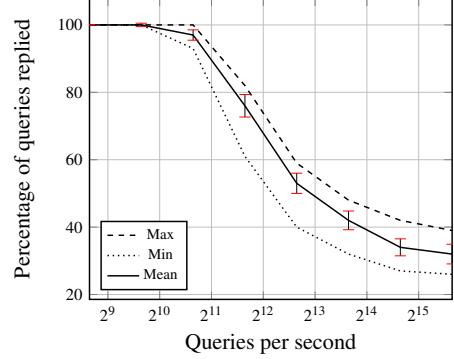


Figure 4: Packet loss in OPRFv2 as a function of query rate. Packet loss is negligible at rates < 3k queries per second.

mised clients, by spoofing packets from such clients. Finally, adversaries might try to circumvent rate-limiting. A secure protocol must defend against all these threats.

Privacy of OPRF inputs and outputs follows from blinding in the OPRF protocol. Clients can check OPRF output correctness and hence detect tampering. In OPRFv1, every KSReq interaction starts with a mutual-authentication TLS handshake, which prevents adversaries from spoofing requests from other clients. In OPRFv2, creating a new session once again involves a mutual-authentication TLS handshake, meaning that an adversary cannot initiate a session pretending to be an uncompromised client. Moreover, an adversary cannot create a fresh KSReq packet belonging to a session which it did not initiate, without a successful MAC forgery (HMAC with SHA256 specifically). Packets cannot be replayed across sessions, due to session identifiers being picked at random and being included in the MAC, and packets cannot be replayed within a session, due to increasing sequence numbers. Overall, both protocols offer protection against request spoofing, and neither of the two protocols introduce new denial-of-service vulnerabilities.

In the Bounded rate-limiting approach, the server keeps track of the total number of the queries made by each client, across all sessions in an epoch, and stops responding after the bound q is reached, meaning that even adversarial clients are restricted to q queries per epoch. In the fixed-delay and exponential-delay approaches, only one query from a client is handled at a time by the KS in a session through the active clients list. If a client makes a second query — even from a different session, while a query is in process, the second query is not processed by the KS, but simply dropped.

Command	Description
$\text{SSput}(P, F, M)$	Stores file contents M as P/F
$\text{SSget}(P, F)$	Gets file P/F
$\text{SSlist}(P)$	Gets metadata of P
$\text{SSdelete}(P, F)$	Delete file F in P
$\text{SSsearch}(P, F)$	Search for file F in P
$\text{SScreate}(P)$	Create directory P
$\text{SSmove}(P_1, F_1, P_2, F_2)$	Move P_1/F_1 to P_2/F_2

Figure 5: API commands exposed by the storage service (SS) used by DupLESS. Here F represents a filename and P is the absolute path in a directory hierarchy.

6 The DupLESS client

The Dupless client works with an SS which implements the interface described in Figure 5 (based on the Dropbox API [39]), and provides an analogous set of commands DLput , DLget , DLlist , etc. Figure 6 gives pseudocode for the DupLESS commands for storing and retrieving a file. We now explain the elements of these commands, and will then discuss how other API commands are handled.

Path and filename encryption. The SS provides a rudimentary file system abstraction. Clients can generate directories, use relative and absolute paths, move files from one directory to another, etc. Following our design goal of supporting as much of the base SS functionality as possible, DupLESS should also support paths, filenames, and related functionalities such as copying files. One option is to treat paths and filenames as non-private, and simply mirror in clear the directory hierarchy and filenames asked for by a user. This has the benefit of simplicity and no path-related overheads, but it relies on users guaranteeing that paths and filenames are, in fact, not confidential. A second option would be to hide the directory structure from the SS by using just a single directory, and storing the client’s directory hierarchy and filenames in completely encrypted form using some kind of digest file. But this would increase complexity and decrease performance as one would (essentially) have to build a file system on top of the SS. For example, this would bar use of the SS API to perform filename searches on behalf of DupLESS.

We design DupLESS to provide some security for directory and filenames while still enabling effective use of the SS APIs. To encrypt file and directory names, we use the SIV DAE scheme [68] $\text{SIV} = (\text{ED}, \text{DD})$ with $\text{HMAC}[\text{SHA256}]$ and $\text{CTR}[\text{AES}]$. The EncPath subroutine takes as input a DAE key K_{dae} , a path P (a sequence of directory names separated by ‘/’), and a filename F , and returns an encrypted path C_{path} and an encrypted filename F . It does so by encrypting each directory D

in P by way of $\text{ED}(K_{\text{dae}}, 0, D)$ and likewise encrypting F by $\text{ED}(K_{\text{dae}}, 0, F)$. (The associated data being set to 0 here will be used to distinguish this use from that of the key encapsulation, see below.) Being deterministic, twice encrypting the same file or directory name results in the same ciphertext. We will then use the ciphertexts, properly encoded into a character set allowed by the SS, as the directory names requested in calls to, e.g., SScreate . We note that the choice of encoding as well as the ciphertext stretch τ_d mean that the maximum filename length supported by DupLESS will be shorter than that of the SS. Should this approach prove limiting, an alternative approach would be to use format-preserving encryption [21] instead to reduce ciphertext expansion.

All this means that we will be able to search for file and directory names and have efficient file copy and move operations. That said, this approach does leak the structure of the plaintext directory hierarchy, the lengths of individual directory and file names, and whether two files have the same name. While length leakage can be addressed with padding mechanisms at a modest cost on storage overhead, hierarchy leakage cannot be addressed without adversely affecting some operations.

Store requests. To store a file with filename F and contents M at path P , the DupLESS client first executes the client portion of the KS protocol (see Section 5). The result is either a message-derived key K or an error message \perp . The client then runs a check canDedup to determine whether to use dedupable encryption or non-dedupable encryption. If $K = \perp$ or canDedup returns false, then a random key is selected and will be used in place of a message-derived key. In this case the resulting ciphertext will not be dedupable. We discuss canDedup more below. The client next encrypts M under K with $\text{CTR}[\text{AES}]$ and a fixed IV to produce ciphertext C_{data} , and then wraps K using SIV to produce ciphertext C_{key} . We include the filename ciphertext C_{name} and C_{data} in order to cryptographically bind together the three ciphertexts. The client uploads to the SS via the SSput command the file “ $C_{\text{name}}.key$ ” with contents C_{key} and C_{data} in file “ $C_{\text{name}}.data$ ”. DupLESS encodes the ciphertexts into character sets allowed by the SS API. Both files are uploaded in parallel to the SS. Usually, the SS might require the client to be authorized, and if this is the case, the authorization can be handled when the client starts.

The “.data” file contains only ciphertext C_{data} , and can be deduplicated by the SS assuming K was not replaced by a random value. The “.key” file cannot be deduplicated, its contents being essentially uniformly distributed, but requires only a fixed, small number of bits equal to $k + \tau_d$. With our instantiation choices, this is 384 bits, and does not lead to significant overheads as we show in Section 7.

$\text{DLput}_{K_{\text{dae}}, K_{\text{ae}}, pk_{\text{ks}}}(P, F, M)$ <hr/> $K \xleftarrow{\$} \text{EvC}^{\text{EvS}}(pk_{\text{ks}}, M)$ $C_{\text{path}}, C_{\text{name}} \leftarrow \text{EncPath}(K_{\text{dae}}, P, F)$ If $\text{canDedup}(P, F, M) = \text{false}$ then $C_{\text{data}} \leftarrow \text{EA}(K_{\text{ae}}, C_{\text{name}}, M)$ $\text{SSput}(C_{\text{path}}, C_{\text{name}} \parallel \text{"data"}, C_{\text{data}})$ Else If $K = \perp$ then $K \xleftarrow{\$} \{0, 1\}^k$ $C_{\text{data}} \leftarrow \text{E}(K, M)$ $C_{\text{key}} \leftarrow \text{ED}(K_{\text{dae}}, 1 \parallel C_{\text{name}} \parallel C_{\text{data}}, K)$ $\text{SSput}(C_{\text{path}}, C_{\text{name}} \parallel \text{"key"}, C_{\text{key}})$ $\text{SSput}(C_{\text{path}}, C_{\text{name}} \parallel \text{"data"}, C_{\text{data}})$	$\text{DLget}_{K_{\text{dae}}, K_{\text{ae}}}(P, F)$ $C_{\text{path}}, C_{\text{name}} \leftarrow \text{EncPath}(K_{\text{dae}}, P, F)$ $C_{\text{data}} \leftarrow \text{SSget}(C_{\text{path}}, C_{\text{name}} \parallel \text{"data"})$ $C_{\text{key}} \leftarrow \text{SSget}(C_{\text{path}}, C_{\text{name}} \parallel \text{"key"})$ If $C_{\text{key}} = \perp$ then Return $\text{DA}(K_{\text{ae}}, C_{\text{name}}, C_{\text{data}})$ Else $K \leftarrow \text{DD}(K_{\text{dae}}, 1 \parallel C_{\text{name}} \parallel C_{\text{data}}, C_{\text{key}})$ If $K = \perp$ then Ret \perp Else Ret $\text{D}(K, C_{\text{data}})$
--	--

Figure 6: DupLESS client procedures for storage and retrieval. They use our server-aided MLE scheme DupLESSMLE = (P, K, E, D) , built with RSA-OPRF[G, H] = $(\text{Kg}, \text{EvC}, \text{EvS}, \text{Vf}, \text{Ev})$ along with the DAE scheme SIV = (ED, DD) , and the AE scheme EtM = (EA, DA) . Instantiations are as described in text. The subroutine `canDedup` runs dedup heuristics while `EncPath` encrypts the path and file name using SIV.

Dedupability control. The `canDedup` subroutine enables fine-grained control over which files end up getting deduplicated, letting clients enforce policies such as not deduplicating anything in a personal folder, and setting a lower threshold on size. Our current implementation uses a simple length heuristic: files less than 1 KB in size are not deduplicated. As our experiments show in Section 7, employing this heuristic does not appear to significantly degrade storage savings.

By default, `DLput` ensures that ciphertexts are of the same format regardless of the output of `canDedup`. However, should `canDedup` mark files non-dedupable based only on public information (such as file length), then we can further optimize performance by producing only a single ciphertext file (i.e. no C_{key}) using an authenticated-encryption scheme with a key K_{ae} derived from the client’s secret key. We use AES in CTR mode with random IVs with HMAC in an Encrypt-then-MAC scheme. This provides a slight improvement in storage savings over non-deduped ciphertexts and requires just a single `SSput` call. We can also query the KS only if needed, which is more efficient.

When `canDedup`’s output depends on private information (e.g., file contents), clients should always interact with the KS. Otherwise there exists a side channel attack in which a network adversary infers from the lack of a KS query the outcome of `canDedup`.

Retrieval and other commands. The pseudocode for retrieval is given in Figure 6. It uses `EncPath` to recompute the encryptions of the paths and filenames, and then issues `SSget` calls to retrieve both C_{key} and C_{data} . It then proceeds by decrypting C_{key} , recovering K , and then using it to decrypt the file contents. If non-dedupable encryption was used and C_{key} was not uploaded, the second

`SSget` call fails and the client decrypts accordingly.

Other commands are implemented in natural ways, and we omit pseudocode for the sake of brevity. DupLESS includes listing the contents of a directory (perform an `SSlist` on the directory and decrypt the paths and filenames); moving the contents of one directory to another (perform an `SSmove` command with encrypted path names); search by relative path and filename (perform an `SSsearch` using the encryptions of the relative path and filename); create a directory (encrypt the directory name and then use `SScreate`); and delete (encrypt the path and filename and perform a delete on that).

The operations are, by design, simple and whenever possible, one-to-one with underlying SS API commands. The security guarantees of SIV mean that an attacker with access to the SS cannot tamper with stored data. An SS-based attacker could, however, delete files or modify the hierarchy structure. While we view these attacks as out of scope, we note that it is easy to add directory hierarchy integrity to DupLESS by having `EncPath` bind ciphertexts for a directory or file to its parent: just include the parent ciphertext in the associated data during encryption. The cost, however, is that filename search can only be performed on full paths.

In DupLESS, only `DLput` requires interaction with the KS, meaning that even if the KS goes down files are *never* lost. Even `DLput` will simply proceed with a random key instead of the message-derived key from the KS. The only penalty in this case is loss of the storage savings due to deduplication.

Other APIs. The interface in Figure 5 is based on the Dropbox API [39]. Google Drive [7] differs by indexing files based on unique IDs instead of names. When a file is uploaded, `SSput` returns a file ID, which should be

provided to `SSget` to retrieve the file. The `SSlist` function returns a mapping between the file names and their IDs. In this case, DupLESS maintains a local map by prefetching and caching file IDs by calling `SSlist` whenever appropriate; this caching reduces `DLget` latency. When a file is uploaded, the encrypted filename and returned ID are added to this map. Whenever a local map lookup fails, the client runs `SSlist` again to check for an update. Hence, the client can start without any local state and dynamically generate the local map.

Supporting keyword search in DupLESS requires additional techniques, such as an encrypted keyword index as in searchable symmetric encryption [34], increasing storage overheads. We leave exploring the addition of keyword search to future work.

7 Implementation and Performance

We implemented a fully functional DupLESS client. The client was written in Python and supports both Dropbox [3] and Google Drive [7]. It will be straightforward to extend the client to work with other services which export an API similar to Figure 5. The client uses two threads during store operations in order to parallelize the two SS API requests. The client takes user credentials as inputs during startup and provides a command line interface for the user to type in commands and arguments. When using Google Drive, a user changing directory prompts the client to fetch the file list ID map asynchronously. We used Python’s SSL and Crypto libraries for the client-side crypto operations and used the OPRFv2 KS protocol.

We now describe the experiments we ran to measure the performance and overheads of DupLESS. We will compare both to direct use of the underlying SS API (no encryption) as well as when using a version of DupLESS modified to implement just MLE, in particular the convergent encryption (CE) scheme, instead of DupLESSMLE. This variant computes the message-derived key K by hashing the file contents, thereby avoiding use of the KS. Otherwise the operations are the same.

Test setting and methodology. We used the same machine as for the KS tests (Section 5). Measurements involving the network were repeated 100 times and other measurements were repeated 1,000 times. We measured running times using the `timeit` Python module. Operations involving files were repeated using files with random contents of size 2^{2i} KB for $i \in \{0, 1, \dots, 8\}$, giving us a file size range of 1 KB to 64 MB.

Dropbox exhibited significant performance variability in the course of our experiments. For example, the median time to upload a 1 KB file was 0.92 seconds, while the maximum observed was 2.64 seconds, with standard

deviation at 0.22 seconds. That is close to 25% of the median. Standard deviation decreases as the file size increases, for example it is only 2% of the median upload time for 32 MB files. We never observed more than 1 Mbps throughput to Dropbox. Google Drive exhibited even slower speeds and more variance.

Storage and retrieval latency. We now compare the time to store and retrieve files using DupLESS, CE, and the plain SS. Figure 7 (top left chart) reports the median time for storage using Dropbox. The latency overhead when storing files with DupLESS starts at about 22% for 1 KB files and reduces to about 11% for 64 MB files.

As we mentioned earlier, Dropbox and Google Drive exhibited significant variation in overall upload and download times. To reduce the effect of these variations on the observed relative performance between DupLESS over the SS, CE over the SS and plain SS, we ran the tests by cycling between the three settings to store the same file, in quick succession, as opposed to, say, running all plain Dropbox tests first. We adopted a similar approach with Google Drive.

We observe that the CE (Convergent Encryption) store times are close to DupLESS store times, since the `KSReq` step, which is the main overhead of DupLESS w.r.t CE, has been optimized for low latency. For example, median CE latency overhead for 1 KB files over Dropbox was 15%. Put differently, the overhead of moving to DupLESS from using CE is quite small, compared to that of using CE over the base system.

Relative retrieval latencies (bottom left, Figure 7) for DupLESS over Dropbox were lower than the store latencies, starting at about 7% for 1 KB files and reducing to about 6% for 64 MB files.

Performance with Google Drive (Figure 7, top middle chart) follows a similar trend, with overhead for DupLESS ranging from 33% to 8% for storage, and 40% to 10% for retrieval, when file sizes go from 1 KB to 64 MB.

These experiments report data only for files larger than 1 KB, as smaller files are not selected for deduplication by `canDedup`. Such files are encrypted with non-dedupable, randomized encryption and latency overheads for storage and retrieval in these cases are negligible in most cases.

Microbenchmarks. We ran microbenchmarks on `DLput` storing 1MB files, to get a breakdown of the overhead. We report median values over 100 trials here. Uploading a 1 MB file with Dropbox takes 2700 milliseconds (ms), while time for the whole `DLput` operation is 3160 ms, with a 17% overhead. The `KSReq` latency, from Section 5, is 82 ms or 3%. We measured the total time for all `DLput` steps except the two `SSput` operations (refer to Figure 6) to be 135 ms, and uploading the content file on top of this took 2837 ms. Then, net overhead

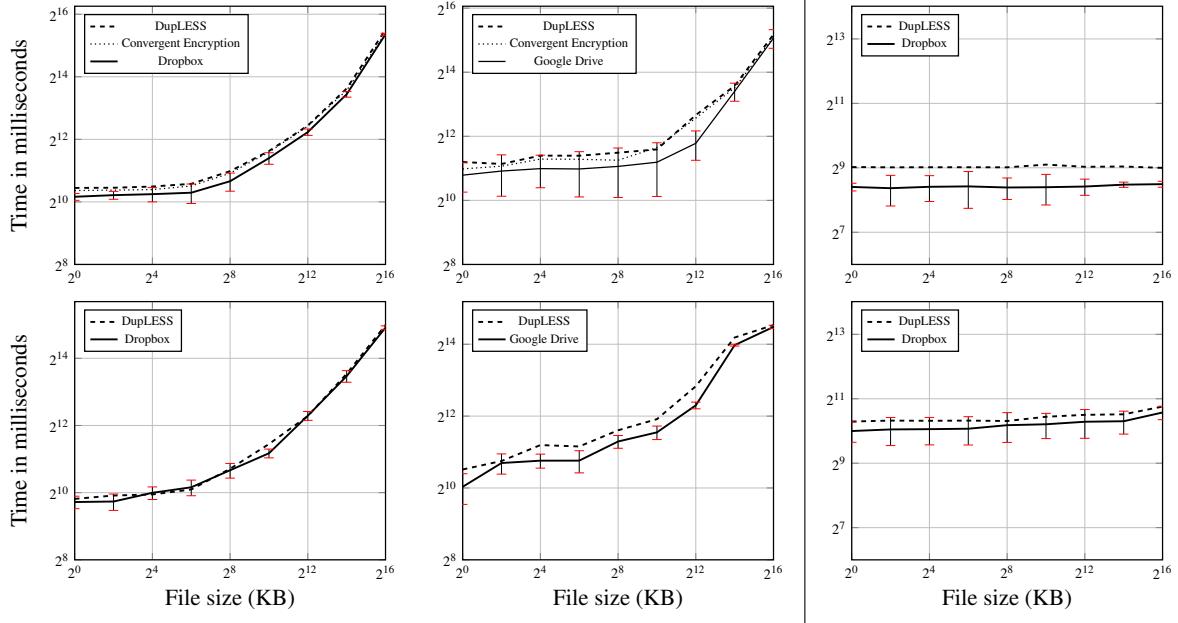


Figure 7: (Left) Median time to store (top two graphs) and retrieve (bottom two graphs) as a function of file size. **(Top Right)** Median time to delete a file as a function of file size. **(Bottom Right)** Median time to copy a file as a function of file size. All axes are log-scale and error bars indicate one standard deviation. Standard deviations are displayed only for base Dropbox/Google Drive times to reduce cluttering.

of KS and cryptographic operations is about 5%, while storing the key file accounts for 12%. Our implementation of `DLput` stores the content and key files simultaneously, by spawning a new thread for storing the key, and waiting for both the stores to complete before finishing. If `DLput` exits before the key store thread completes, i.e., if the key is uploaded asynchronously, then the overhead drops to 14%. On the other hand, uploading the files sequentially by storing the content file first, and then storing the key, incurs a 54% overhead (for 1 MB files).

Bandwidth overhead. We measured the increase in transmission bandwidth due to DupLESS during storage. To do so, we used `tcpdump` and filtered out all traffic unrelated to Dropbox and DupLESS. We took from this the total number of bytes (in either direction). For even very small files, the Dropbox API incurs a cost of about 7 KB per upload. Figure 8 (middle) shows the ratio of bandwidth used by DupLESS to that used by plain Dropbox as file size increases. Given the small constant size of the extra file sent by DupLESS, overhead quickly diminishes as files get larger.

Storage overhead. DupLESS incurs storage overhead, due to the encrypted file name, the MLE key, and the MAC. The sizes of these components are independent of the length of the file. Let n denote the length of the filename in bytes. Then, encrypting the filename with SIV and encoding the result with base64 encoding consumes

$2n + 32$ bytes. Repeating the process for the content and key files, and adding extensions brings the file name overhead to $4n + 72 - n = 3n + 72$ bytes. The contents of the key file include the MLE key, which is 16 bytes long in our case, and the 32 byte HMAC output, and hence 48 bytes together. Thus, the total overhead for a file with an n -byte filename is $3n + 120$ bytes. Recall that if the file size is smaller than 1 KB, then `canDedup` rejects the file for deduplication. In this case, the overhead from encrypting and encoding the file name is $n + 32$ bytes, since only one file is stored. Randomized encryption adds 16 bytes, bringing the total to $n + 48$ bytes.

To assess the overall effect of this in practice, we collected a corpus of around 2,000 public Amazon virtual machine images (AMIs) hosting Linux guests. The AMIs were gathered using techniques similar to those used previously [14, 28], the difference being that we as well downloaded a snapshot of the full file system for each public AMI. There are 101,965,188 unique files across all the AMIs, with total content size of all files being 2,063 GB. We computed cryptographic hashes over the content of all files in the dataset, in order to simulate the storage footprint when using plain deduplication as well as when using DupLESS. This dataset has significant redundancy, as one would expect, given that many AMIs are derivative of other AMIs and so share common files. The plain dedup storage required for the file contents is just 335 GB. DupLESS with the dedupability

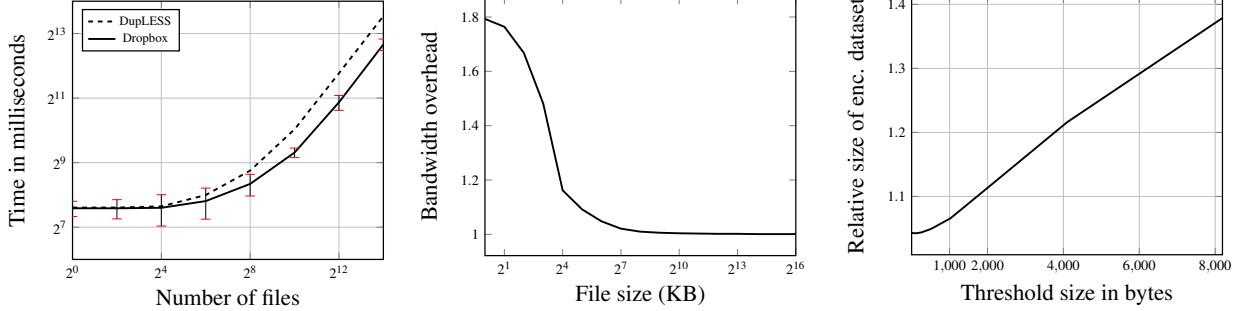


Figure 8: (Left) Median time to list a directory as a function of number of files in the directory. Both axes are logscale and error bars are one standard deviation. (Middle) Network bandwidth overhead of DupLESS as a function of file size (log-scale axis) for store operations. (Right) The ratio of space required when DupLESS is used for the AMI dataset and when plain dedup is used, as a function of the dedupable threshold length.

length threshold used by `canDedup` (see Section 6) set to zero (all files were deduplicable) requires 350 GB, or an overhead of about 4.5%. In this we counted the size of the filename and path ciphertexts for the DupLESS estimate, though we did not count these in the base storage costs. (This can only inflate the reported overhead.)

We also measure the effect of higher threshold values, when using non-dedupable encryption. Setting the threshold to 100 bytes saves a few hundred megabytes in storage. This suggests little benefit from deduping small files, which is in line with previous observations about deduplication on small files [61].

Figure 8 plots the storage used for a wide range of threshold values. Setting a larger threshold leads to improved security (for those files) and faster uploads (due to one less `SSput` request) and appears to have, at least for this dataset, only modest impact on storage overheads for even moderately sized thresholds.

The above results may not extend to settings with significantly different workloads. For example, we caution when there is significantly less deduplication across the corpus, DupLESS may introduce greater overhead. In the worst case, when there is no deduplication whatsoever and all 1 KB files with long names of about 100 characters, the overhead will be almost 30%. Of course here one could have `canDedup` force use of non-dedupable encryption to reduce overhead for all files.

Overhead of other operations. The time to perform `DLmove`, `DLdelete`, and `DLlist` operations are reported in Figure 7 and Figure 8 for Dropbox. In these operations, the DupLESS overheads and the data sent over the network involve just the filenames, and do not depend on the length of the file. (The operations themselves may depend on file length of course.) The overhead of DupLESS therefore remains constant. For `DLlist`, DupLESS times are close to those of plain Dropbox for folders with twice as many files, since DupLESS stores an extra key

encapsulation file for each user file. We also measured the times for `DLsearch` and `DLcreate`, but in these cases the DupLESS overhead was negligible.

8 Security of DupLESS

We argued about the security of the KS protocols and client encryption algorithms in sections 5 and 6. Now, we look at the big picture, the security of DupLESS as a whole. DupLESS provides security that is usually significantly better than current, convergent encryption based deduplicated encryption architectures, and never worse. To expand, security is “hedged,” or multi-tiered, and we distinguish three tiers, always assuming that the adversary has compromised the SS and has the ciphertexts.

The optimistic or best case is that the adversary does not have authorized access to the KS. Recall that both OPRFv1 and OPRFv2 need clients to authenticate first, before requesting queries, meaning that in this setting, the attacker cannot obtain any information about message-derived keys. These keys are effectively random to the attacker. In other words, all data stored on the SS is encrypted with random keys, including file contents, names and paths. The attacker can only learn about equality of file contents and the topology of the file system (including file sizes). Thus, DupLESS provides, effectively, semantic security. In particular, security holds even for predictable messages. By using the SIV DAE scheme, and generating tags over the file names, file contents and keys, DupLESS ensures that attempts by the SS to tamper with client data will be detected.

The semi-optimistic, or next best case is that the adversary, having compromised one or more clients, has remote access to the KS but does not have the KS’s secret key. Here, security for completely predictable files is impossible. Thus, it is crucial to slow down brute-force attacks and push the feasibility threshold for the attacker. We saw in Section 5 that with the right rate-

limiting setup (Bounded, with appropriate parameters), brute-force attacks can be slowed down significantly. Importantly, attackers cannot circumvent the rate-limiting measures, by say, repeating queries.

Finally, the pessimistic case is that the adversary has compromised the KS and has obtained its key. Even then, we retain the guarantees of MLE, and specifically CE, meaning security for unpredictable messages [18]. Appropriate deployment scenarios, such as locating the KS within the boundary of a large corporate customer of a SS, make the optimistic case the most prevalent, resulting in appreciable security gains without significant increase in cost. The security of non-deduplicated files, file names, and path names is unaffected by these escalations in attack severity.

9 Conclusions

We studied the problem of providing secure outsourced storage that both supports deduplication and resists brute-force attacks. We design a system, DupLESS, that combines a CE-type base MLE scheme with the ability to obtain message-derived keys with the help of a key server (KS) shared amongst a group of clients. The clients interact with the KS by a protocol for oblivious PRFs, ensuring that the KS can cryptographically mix in secret material to the per-message keys while learning nothing about files stored by clients.

These mechanisms ensure that DupLESS provides strong security against external attacks which compromise the SS and communication channels (nothing is leaked beyond file lengths, equality, and access patterns), and that the security of DupLESS gracefully degrades in the face of comprised systems. Should a client be compromised, learning the plaintext underlying another client’s ciphertext requires mounting an online brute-force attacks (which can be slowed by a rate-limited KS). Should the KS be compromised, the attacker must still attempt an offline brute-force attack, matching the guarantees of traditional MLE schemes.

The substantial increase in security comes at a modest price in terms of performance, and a small increase in storage requirements relative to the base system. The low performance overhead results in part from optimizing the client-to-KS OPRF protocol, and also from ensuring DupLESS uses a low number of interactions with the SS. We show that DupLESS is easy to deploy: it can work transparently on top of any SS implementing a simple storage interface, as shown by our prototype for Dropbox and Google Drive.

Acknowledgements

We thank the anonymous USENIX Security 2013 reviewers for their valuable comments and feedback. We thank Matt Green for his feedback on early drafts of the paper. Ristenpart was supported in part by generous gifts from Microsoft, RSA Labs, and NetApp. Bellare and Keelveedhi were supported in part by NSF grants CNS-1228890, CNS-1116800, CNS 0904380 and CCF-0915675.

References

- [1] Bitcasa, infinite storage. <http://www.bitcasa.com/>.
- [2] Ciphertite data backup. <http://www.ciphertite.com/>.
- [3] Dropbox, a file-storage and sharing service. <http://www.dropbox.com/>.
- [4] Dupless source code. <http://cseweb.ucsd.edu/users/skeelvée/dupless>.
- [5] The Flud backup system. <http://flud.org/wiki/Architecture>.
- [6] GUNet, a framework for secure peer-to-peer networking. <https://gnunet.org/>.
- [7] Google Drive. <http://drive.google.com>.
- [8] ADYA, A., BOLOSKY, W., CASTRO, M., CERMAK, G., CHAIKEN, R., DOUCEUR, J., HOWELL, J., LORCH, J., THEIMER, M., AND WATTENHOFER, R. Farsite: Federated, available, and reliable storage for an incompletely trusted environment. *ACM SIGOPS Operating Systems Review 36*, SI (2002), 1–14.
- [9] AMAZON. Amazon Elastic Block Store (EBS). <http://aws.amazon.com/ebs>.
- [10] AMAZON. Amazon Elastic Compute Cloud (EC2). <http://aws.amazon.com/ec2>.
- [11] AMAZON. Amazon Simple Storage Service (Amazon S3). <http://aws.amazon.com/s3>.
- [12] ANDERSON, P., AND ZHANG, L. Fast and secure laptop backups with encrypted de-duplication. In *Proc. of USENIX LISA* (2010).
- [13] ATENIESE, G., BURNS, R. C., CURTMOLA, R., HERRING, J., KISSNER, L., PETERSON, Z. N. J., AND SONG, D. Provable data possession at untrusted stores. In *ACM CCS 07* (Alexandria, Virginia, USA, Oct. 28–31, 2007), P. Ning, S. D. C. di Vimercati, and P. F. Syverson, Eds., ACM Press, pp. 598–609.
- [14] BALDUZZI, M., ZADDACH, J., BALZAROTTI, D., KIRDA, E., AND LOUREIRO, S. A security analysis of amazon’s elastic compute cloud service. In *Proceedings of the 27th Annual ACM Symposium on Applied Computing* (2012), ACM, pp. 1427–1434.
- [15] BATTEEN, C., BARR, K., SARAF, A., AND TREPININ, S. pStore: A secure peer-to-peer backup system. *Unpublished report, MIT Laboratory for Computer Science* (2001).
- [16] BELLARE, M., BOLDYREVA, A., AND O’NEILL, A. Deterministic and efficiently searchable encryption. In *CRYPTO 2007* (Santa Barbara, CA, USA, Aug. 19–23, 2007), A. Menezes, Ed., vol. 4622 of *LNCS*, Springer, Berlin, Germany, pp. 535–552.
- [17] BELLARE, M., FISCHLIN, M., O’NEILL, A., AND RISTENPART, T. Deterministic encryption: Definitional equivalences and constructions without random oracles. In *CRYPTO 2008* (Santa Barbara, CA, USA, Aug. 17–21, 2008), D. Wagner, Ed., vol. 5157 of *LNCS*, Springer, Berlin, Germany, pp. 360–378.

- [18] BELLARE, M., KEELVEEDHI, S., AND RISTENPART, T. Message-locked encryption and secure deduplication. In *EUROCRYPT 2013, to appear*. Cryptology ePrint Archive, Report 2012/631, November 2012.
- [19] BELLARE, M., AND NAMPREMPRE, C. Authenticated encryption: Relations among notions and analysis of the generic composition paradigm. In *ASIACRYPT 2000* (Kyoto, Japan, Dec. 3–7, 2000), T. Okamoto, Ed., vol. 1976 of *LNCS*, Springer, Berlin, Germany, pp. 531–545.
- [20] BELLARE, M., NAMPREMPRE, C., POINTCHEVAL, D., AND SEMANKO, M. The one-more-RSA-inversion problems and the security of Chaum’s blind signature scheme. *Journal of Cryptology* 16, 3 (June 2003), 185–215.
- [21] BELLARE, M., RISTENPART, T., ROGAWAY, P., AND STEGERS, T. Format-preserving encryption. In *SAC 2009* (Calgary, Alberta, Canada, Aug. 13–14, 2009), M. J. Jacobson Jr., V. Rijmen, and R. Safavi-Naini, Eds., vol. 5867 of *LNCS*, Springer, Berlin, Germany, pp. 295–312.
- [22] BELLARE, M., AND ROGAWAY, P. Random oracles are practical: A paradigm for designing efficient protocols. In *ACM CCS 93* (Fairfax, Virginia, USA, Nov. 3–5, 1993), V. Ashby, Ed., ACM Press, pp. 62–73.
- [23] BELLARE, M., AND YUNG, M. Certifying permutations: Non-interactive zero-knowledge based on any trapdoor permutation. *Journal of Cryptology* 9, 3 (1996), 149–166.
- [24] BISSIAS, G., LIBERATORE, M., JENSEN, D., AND LEVINE, B. N. Privacy Vulnerabilities in Encrypted HTTP Streams. In *Proceedings of the Privacy Enhancing Technologies Workshop* (May 2005), pp. 1–11.
- [25] BONEH, D., GENTRY, C., HALEVI, S., WANG, F., AND WU, D. Private database queries using somewhat homomorphic encryption.
- [26] BOWERS, K. D., JUELS, A., AND OPREA, A. HAIL: a high-availability and integrity layer for cloud storage. In *ACM CCS 09* (Chicago, Illinois, USA, Nov. 9–13, 2009), E. Al-Shaer, S. Jha, and A. D. Keromytis, Eds., ACM Press, pp. 187–198.
- [27] BRAKERSKI, Z., AND SEGEV, G. Better security for deterministic public-key encryption: The auxiliary-input setting. In *CRYPTO 2011* (Santa Barbara, CA, USA, Aug. 14–18, 2011), P. Rogaway, Ed., vol. 6841 of *LNCS*, Springer, Berlin, Germany, pp. 543–560.
- [28] BUGIEL, S., NÜRNBERGER, S., PÖPPELMANN, T., SADEGI, A., AND SCHNEIDER, T. Amazonia: when elasticity snaps back. In *ACM Conference on Computer and Communications Security – CCS ’11* (2011), ACM, pp. 389–400.
- [29] CAMENISCH, J., NEVEN, G., AND SHELAT, A. Simulatable adaptive oblivious transfer. In *EUROCRYPT 2007* (Barcelona, Spain, May 20–24, 2007), M. Naor, Ed., vol. 4515 of *LNCS*, Springer, Berlin, Germany, pp. 573–590.
- [30] CHAUM, D. Blind signatures for untraceable payments. In *CRYPTO’82* (Santa Barbara, CA, USA, 1983), D. Chaum, R. L. Rivest, and A. T. Sherman, Eds., Plenum Press, New York, USA, pp. 199–203.
- [31] CHEN, S., WANG, R., WANG, X., AND ZHANG, K. Side-Channel Leaks in Web Applications: a Reality Today, a Challenge Tomorrow. In *Proceedings of the IEEE Symposium on Security and Privacy* (May 2010), pp. 191–206.
- [32] COOLEY, J., TAYLOR, C., AND PEACOCK, A. ABS: the apportioned backup system. *MIT Laboratory for Computer Science* (2004).
- [33] COX, L. P., MURRAY, C. D., AND NOBLE, B. D. Pastiche: making backup cheap and easy. *SIGOPS Oper. Syst. Rev.* 36 (Dec. 2002), 285–298.
- [34] CURTMOLA, R., GARAY, J. A., KAMARA, S., AND OSTROVSKY, R. Searchable symmetric encryption: improved definitions and efficient constructions. In *ACM CCS 06* (Alexandria, Virginia, USA, Oct. 30 – Nov. 3, 2006), A. Juels, R. N. Wright, and S. Vimercati, Eds., ACM Press, pp. 79–88.
- [35] DE CRISTOFARO, E., LU, Y., AND TSUDIK, G. Efficient techniques for privacy-preserving sharing of sensitive information. In *Proceedings of the 4th international conference on Trust and trustworthy computing* (Berlin, Heidelberg, 2011), TRUST’11, Springer-Verlag, pp. 239–253.
- [36] DE CRISTOFARO, E., SORIENTE, C., TSUDIK, G., AND WILLIAMS, A. Hummingbird: Privacy at the time of twitter. In *Security and Privacy (SP), 2012 IEEE Symposium on* (2012), IEEE, pp. 285–299.
- [37] DECANDIA, G., HASTORUN, D., JAMPANI, M., KAKULAPATI, G., LAKSHMAN, A., PILCHIN, A., SIVASUBRAMANIAN, S., VOSSHALL, P., AND VOGELS, W. Dynamo: amazon’s highly available key-value store. In *ACM SIGOPS Operating Systems Review* (2007), vol. 41, ACM, pp. 205–220.
- [38] DOUCEUR, J., ADYA, A., BOLOSKY, W., SIMON, D., AND THEIMER, M. Reclaiming space from duplicate files in a serverless distributed file system. In *Distributed Computing Systems, 2002. Proceedings. 22nd International Conference on* (2002), IEEE, pp. 617–624.
- [39] DROPBOX. Dropbox API Reference. <https://www.dropbox.com/developers/reference/api>.
- [40] DYER, K., COULL, S., RISTENPART, T., AND SHRIMPTON, T. Peek-a-boo, i still see you: Why efficient traffic analysis countermeasures fail. In *Security and Privacy (SP), 2012 IEEE Symposium on* (2012), IEEE, pp. 332–346.
- [41] ERWAY, C. C., KÜPÇÜ, A., PAPAMANTHOU, C., AND TAMASSIA, R. Dynamic provable data possession. In *ACM CCS 09* (Chicago, Illinois, USA, Nov. 9–13, 2009), E. Al-Shaer, S. Jha, and A. D. Keromytis, Eds., ACM Press, pp. 213–222.
- [42] GOH, E., SHACHAM, H., MODADUGU, N., AND BONEH, D. Sirius: Securing remote untrusted storage. *NDSS*.
- [43] GOLDWASSER, S., AND MICALI, S. Probabilistic encryption. *Journal of Computer and System Sciences* 28, 2 (1984), 270–299.
- [44] GRIBBLE, S. D., MANKU, G. S., ROSELLI, D., BREWER, E. A., GIBSON, T. J., AND MILLER, E. L. Self-similarity in file systems. In *ACM SIGMETRICS Performance Evaluation Review* (1998), vol. 26, ACM, pp. 141–150.
- [45] HALEVI, S., HARLIK, D., PINKAS, B., AND SHULMAN-PELEG, A. Proofs of ownership in remote storage systems. In *Proceedings of the 18th ACM conference on Computer and communications security* (2011), ACM, pp. 491–500.
- [46] HARLIK, D., PINKAS, B., AND SHULMAN-PELEG, A. Side channels in cloud services: Deduplication in cloud storage. *Security & Privacy, IEEE* 8, 6 (2010), 40–47.
- [47] HINTZ, A. Fingerprinting Websites Using Traffic Analysis. In *Proceedings of the Privacy Enhancing Technologies Workshop* (April 2002), pp. 171–178.
- [48] ISLAM, M., KUZU, M., AND KANTARCIOLU, M. Access pattern disclosure on searchable encryption: Ramification, attack and mitigation. In *Network and Distributed System Security Symposium (NDSS12)* (2012).
- [49] JIM GUILFORD, KIRK YAP, V. G. Fast SHA-256 Implementations on Intel Architecture Processors. <http://download.intel.com/embedded/processor/whitepaper/327457.pdf>.
- [50] JIN, K., AND MILLER, E. L. The effectiveness of deduplication on virtual machine disk images. In *Proceedings of SYSTOR 2009: The Israeli Experimental Systems Conference* (2009), ACM, p. 7.

- [51] JUELS, A., AND KALISKI JR., B. S. Pors: proofs of retrievability for large files. In *ACM CCS 07* (Alexandria, Virginia, USA, Oct. 28–31, 2007), P. Ning, S. D. C. di Vimercati, and P. F. Syverson, Eds., ACM Press, pp. 584–597.
- [52] KAKVI, S., KILTZ, E., AND MAY, A. Certifying rsa. *Advances in Cryptology—ASIACRYPT 2012* (2012), 404–414.
- [53] KALLAHALLA, M., RIEDEL, E., SWAMINATHAN, R., WANG, Q., AND FU, K. Plutus: Scalable secure file sharing on untrusted storage. In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies* (2003), pp. 29–42.
- [54] KAMARA, S., PAPAMANTHOU, C., AND ROEDER, T. Cs2: A searchable cryptographic cloud storage system. Tech. rep., Technical Report MSR-TR-2011-58, Microsoft, 2011.
- [55] KILLIJIAN, M., COURTÈS, L., POWELL, D., ET AL. A survey of cooperative backup mechanisms, 2006.
- [56] LEACH, P. J., AND NAIK, D. C. A Common Internet File System (CIFS/1.0) Protocol. <http://tools.ietf.org/html/draft-leach-cifs-v1-spec-01>.
- [57] LEUNG, A. W., PASUPATHY, S., GOODSON, G., AND MILLER, E. L. Measurement and analysis of large-scale network file system workloads. In *USENIX 2008 Annual Technical Conference on Annual Technical Conference* (2008), pp. 213–226.
- [58] LI, J., KROHN, M., MAZIÈRES, D., AND SHASHA, D. *Secure untrusted data repository (SUNDR)*. Defense Technical Information Center, 2003.
- [59] LIBERATORE, M., AND LEVINE, B. N. Inferring the Source of Encrypted HTTP Connections. In *Proceedings of the ACM Conference on Computer and Communications Security* (November 2006), pp. 255–263.
- [60] MARQUES, L., AND COSTA, C. Secure deduplication on mobile devices. In *Proceedings of the 2011 Workshop on Open Source and Design of Communication* (2011), ACM, pp. 19–26.
- [61] MEYER, D. T., AND BOLOSKY, W. J. A study of practical deduplication. *ACM Transactions on Storage (TOS)* 7, 4 (2012), 14.
- [62] MICROSYSTEMS, S. NFS: Network File System Protocol Specification. <http://tools.ietf.org/html/rfc1094>.
- [63] MOZY. Mozy, a file-storage and sharing service. <http://mozy.com/>.
- [64] NAOR, M., AND REINGOLD, O. Number-theoretic constructions of efficient pseudo-random functions. In *38th FOCS* (Miami Beach, Florida, Oct. 19–22, 1997), IEEE Computer Society Press, pp. 458–467.
- [65] PANCHENKO, A., NIJSEN, L., ZINNEN, A., AND ENGEL, T. Website Fingerprinting in Onion Routing-based Anonymization Networks. In *Proceedings of the Workshop on Privacy in the Electronic Society* (October 2011), pp. 103–114.
- [66] RAHUMED, A., CHEN, H., TANG, Y., LEE, P., AND LUI, J. A secure cloud backup system with assured deletion and version control. In *Parallel Processing Workshops (ICPPW), 2011 40th International Conference on* (2011), IEEE, pp. 160–167.
- [67] ROGAWAY, P. Authenticated-encryption with associated-data. In *ACM CCS 02* (Washington D.C., USA, Nov. 18–22, 2002), V. Atluri, Ed., ACM Press, pp. 98–107.
- [68] ROGAWAY, P., AND SHRIMPTON, T. A provable-security treatment of the key-wrap problem. In *EUROCRYPT 2006* (St. Petersburg, Russia, May 28 – June 1, 2006), S. Vaudenay, Ed., vol. 4004 of *LNCS*, Springer, Berlin, Germany, pp. 373–390.
- [69] SEARS, R., VAN INGEN, C., AND GRAY, J. To blob or not to blob: Large object storage in a database or a filesystem? *arXiv preprint cs/0701168* (2007).
- [70] SHACHAM, H., AND WATERS, B. Compact proofs of retrievability. In *ASIACRYPT 2008* (Melbourne, Australia, Dec. 7–11, 2008), J. Pieprzyk, Ed., vol. 5350 of *LNCS*, Springer, Berlin, Germany, pp. 90–107.
- [71] STORER, M., GREENAN, K., LONG, D., AND MILLER, E. Secure data deduplication. In *Proceedings of the 4th ACM international workshop on Storage security and survivability* (2008), ACM, pp. 1–10.
- [72] SUN, Q., SIMON, D. R., WANG, Y.-M., RUSSELL, W., PADMANABHAN, V. N., AND QIU, L. Statistical Identification of Encrypted Web Browsing Traffic. In *Proceedings of the IEEE Symposium on Security and Privacy* (May 2002), pp. 19–30.
- [73] VAN DER LAAN, W. Dropship. <https://github.com/drivedan/dropship>.
- [74] WALLACE, G., DOUGLIS, F., QIAN, H., SHILANE, P., SMALLDONE, S., CHAMNESS, M., AND HSU, W. Characteristics of backup workloads in production systems. In *Proceedings of the Tenth USENIX Conference on File and Storage Technologies (FAST12)* (2012).
- [75] WANG, W., LI, Z., OWENS, R., AND BHARGAVA, B. Secure and efficient access to outsourced data. In *Proceedings of the 2009 ACM workshop on Cloud computing security* (2009), ACM, pp. 55–66.
- [76] WILCOX-O’HEARN, Z. Convergent encryption reconsidered, 2011. <http://www.mail-archive.com/cryptography@metzdowd.com/msg08949.html>.
- [77] WILCOX-O’HEARN, Z., PERTTULA, D., AND WARNER, B. Confirmation Of A File Attack. https://tahoe-lafs.org/hacktahaelafs/drew_perttula.html.
- [78] WILCOX-O’HEARN, Z., AND WARNER, B. Tahoe: The least-authority filesystem. In *Proceedings of the 4th ACM international workshop on Storage security and survivability* (2008), ACM, pp. 21–26.
- [79] XU, J., CHANG, E.-C., AND ZHOU, J. Leakage-resilient client-side deduplication of encrypted data in cloud storage. *Cryptology ePrint Archive, Report 2011/538*, 2011. <http://eprint.iacr.org/>.

Trafficking Fraudulent Accounts: The Role of the Underground Market in Twitter Spam and Abuse

Kurt Thomas^{†◊} Damon McCoy[‡] Chris Grier^{†*} Alek Kolcz[◊] Vern Paxson^{†*}

[†]University of California, Berkeley [‡]George Mason University

^{*}International Computer Science Institute [◊]Twitter

{kthomas, grier, vern}@cs.berkeley.edu mccoy@cs.gmu.edu ark@twitter.com

Abstract

As web services such as Twitter, Facebook, Google, and Yahoo now dominate the daily activities of Internet users, cyber criminals have adapted their monetization strategies to engage users within these walled gardens. To facilitate access to these sites, an underground market has emerged where fraudulent accounts – automatically generated credentials used to perpetrate scams, phishing, and malware – are sold in bulk by the thousands. In order to understand this shadowy economy, we investigate the market for fraudulent Twitter accounts to monitor prices, availability, and fraud perpetrated by 27 merchants over the course of a 10-month period. We use our insights to develop a classifier to retroactively detect several million fraudulent accounts sold via this marketplace, 95% of which we disable with Twitter’s help. During active months, the 27 merchants we monitor appeared responsible for registering 10–20% of all accounts later flagged for spam by Twitter, generating \$127–459K for their efforts.

1 Introduction

As web services such as Twitter, Facebook, Google, and Yahoo now dominate the daily activities of Internet users [1], cyber criminals have adapted their monetization strategies to engage users within these walled gardens. This has lead to a proliferation of *fraudulent accounts* – automatically generated credentials used to disseminate scams, phishing, and malware. Recent studies from 2011 estimate at least 3% of active Twitter accounts are fraudulent [29]. Facebook estimates its own fraudulent account population at 1.5% of its active user base [13], and the problem extends to major web services beyond just social networks [14].

The complexities required to circumvent registration barriers such as CAPTCHAs, email confirmation, and IP

blacklists have lead to the emergence of an underground market that specializes in selling fraudulent accounts in bulk. *Account merchants* operating in this space brazenly advertise: a simple search query for “buy twitter accounts” yields a multitude of offers for fraudulent Twitter credentials with prices ranging from \$10–200 per thousand. Once purchased, accounts serve as stepping stones to more profitable spam enterprises that degrade the quality of web services, such as pharmaceutical spam [17] or fake anti-virus campaigns [25].

In this paper we describe our investigation of the underground market profiting from Twitter credentials to study how it operates, the impact the market has on Twitter spam levels, and exactly how merchants circumvent automated registration barriers.¹ In total, we identified and monitored 27 account merchants that advertise via web storefronts, blackhat forums, and freelance labor sites. With the express permission of Twitter, we conducted a longitudinal study of these merchants and purchased a total of 121,027 fraudulent Twitter accounts on a bi-weekly basis over ten months from June, 2012 – April, 2013. Throughout this process, we tracked account prices, availability, and fraud in the marketplace. Our findings show that merchants thoroughly understand Twitter’s existing defenses against automated registration, and as a result can generate thousands of accounts with little disruption in availability or instability in pricing.

In order to fulfill orders for fraudulent Twitter accounts, we find that merchants rely on CAPTCHA solving services; fraudulent email credentials from Hotmail, Yahoo, and mail.ru; and tens of thousands of hosts located around the globe to provide a diverse pool of IP addresses

¹Our study is limited to Twitter, as we were unable to acquire permission to conduct our research from other companies we saw being abused.

to evade blacklisting and throttling. In turn, merchants stockpile accounts months in advance of their sale, where “pre-aged” accounts have become a selling point in the underground market. We identify which registration barriers effectively increase the price of accounts and summarize our observations into a set of recommendations for how web services can improve existing automation barriers to increase the cost of fraudulent credentials.

Finally, to estimate the overall impact the underground market has on Twitter spam we leveraged our understanding of how merchants abuse the registration process in order to develop a classifier that retroactively detects fraudulent accounts. We applied our classifier to all accounts registered on Twitter in the last year and identify several million suspected fraudulent accounts generated and sold via the underground market. During active months, the 27 merchants we monitor appeared responsible for registering 10–20% of all accounts later flagged by Twitter as spam. For their efforts, the merchants generated an estimated total revenue between \$127,000–\$459,000 from the sale of accounts.

With Twitter’s cooperation, we disable 95% of all fraudulent accounts registered by the merchants we track, including those previously sold but not yet suspended for spamming. Throughout the suspension process, we simultaneously monitor the underground market for any fallout. While we do not observe an appreciable increase in pricing or delay in merchants delivering new accounts, we find 90% of all purchased accounts immediately after our action are suspended on arrival. We are now actively working with Twitter to integrate our defense into their real-time detection framework to help prevent abusive signups.

In summary, we frame our contributions as follows:

- We perform a 10 month longitudinal study of 27 merchants profiting from the sale of Twitter accounts.
- We develop a classifier based on registration signals that detects several million fraudulent accounts that merchants sold to generate \$127,000–\$459,000 in revenue.
- We investigate the impact that the underground market has on Twitter spam levels and find 10–20% all spam accounts originate from the merchants we study.
- We investigate the failures of existing automated registration barriers and provide a set of recommendations to increase the cost of generating fraudulent accounts.

2 Background

Fraudulent accounts are just a single facet of the menagerie of digital criminal goods and services for sale in the underground market. We provide an overview of previous investigations into the digital blackmarket, outline the role that account abuse plays in this space, and summarize existing strategies for detecting spam and abuse. Finally, in order to carry out our investigation of the market for fraudulent Twitter accounts, we adhere to a strict set of legal and ethical guidelines set down by our institutions and by Twitter, documented here.

2.1 Underground Market

At the center of the for-profit spam and malware ecosystem is an underground market that connects Internet miscreants with parties selling a range of specialized products and services including spam hosting [2, 11], CAPTCHA solving services [19], pay-per-install hosts [4], and exploit kits [9]. Even simple services such as garnering favorable reviews or writing web page content are for sale [21, 31]. Revenue generated by miscreants participating in this market varies widely based on business strategy, with spam affiliate programs generating \$12–\$92 million [17] and fake anti-virus scammers \$5–116 million [25] over the course of their operations.

Specialization within this ecosystem is the norm. Organized criminal communities include carders that siphon credit card wealth [7]; email spam affiliate programs [16]; and browser exploit developers and traffic generators [9]. The appearance of account merchants is yet another specialization where sellers enable other miscreants to penetrate walled garden services, while at the same time abstracting away the complexities of CAPTCHA solving, acquiring unique emails, and dodging IP blacklisting. These accounts can then be used for a multitude of activities, outlined below, that directly generate a profit for miscreants.

2.2 Impact of Fraudulent Accounts

Miscreants leverage fraudulent social networking accounts to expose legitimate users to scams, phishing, and malware [8, 10]. Spam monetization relies on both grey-market and legitimate affiliate URL programs, ad syndication services, and ad-based URL shortening [29]. Apart from for-profit activities, miscreants have also leveraged fraudulent accounts to launch attacks from within Twitter for the express purposes of censoring political speech [28]. All of these examples serve to illustrate the deleterious effect that fraudulent accounts have on social networks and user safety.

2.3 Spam Detection Strategies

The pervasive nuisance of spam in social networks has lead to a multitude of detection strategies. These include analyzing social graph properties of sybil accounts [6, 33, 34], characterizing the arrival rate and distribution of posts [8], analyzing statistical properties of account profiles [3, 26], detecting spam URLs posted by accounts [27], and identifying common spam redirect paths [15]. While effective, all of these approaches rely on *at-abuse* time metrics that target strong signals such as sending a spam URL or forming hundreds of relationships in a short period. Consequently, at-abuse time classifiers delay detection until an attack is underway, potentially exposing legitimate users to spam activities before enough evidence of nefarious behavior triggers detection. Furthermore, dormant accounts registered by account merchants will go undetected until miscreants purchase the accounts and subsequently send spam. Overcoming these shortcomings requires *at-registration* abuse detection that flags fraudulent accounts during the registration process before any further interaction with a web service can occur.

2.4 Ethical Considerations

Our study hinges on infiltrating the market for fraudulent Twitter credentials where we interact with account merchants and potentially galvanize the abuse of Twitter. We do so with the express intent of understanding how sellers register accounts and to disrupt their future efforts, but that does not allay our legal or ethical obligations. Prior to conducting our study, we worked with Twitter and our institutions to set down guidelines for interacting with merchants. A detailed summary of the restrictions placed on our study is available in Appendix A

3 Marketplace for Twitter Accounts

We infiltrate the market for Twitter accounts to understand its organization, pricing structure, and the availability of accounts over time. Through the course of our study, we identify 27 account merchants (or sellers) whom we purchase from on a bi-weekly basis from June, 2012 – April, 2013. We determine that merchants can provide thousands of accounts within 24 hours at a price of \$0.02 – \$0.10 per account.

3.1 Identifying Merchants

With no central operation of the underground market, we resort to investigating common haunts: advertisements via search engines, blackhat forums such as *blackhat-*

world.com, and freelance labor pages including Fiverr and Freelancer [20, 21]. In total, we identify a disparate group of 27 merchants. Of these, 10 operate their own websites and allow purchases via automated forms, 5 solicit via blackhat forums, and 12 advertise via freelance sites that take a cut from sales. Advertisements for Twitter accounts range in offerings from credentials for accounts with no profile or picture, to “pre-aged” accounts² that are months old with unique biographies and profile data. Merchants even offer 48 hours of support, during which miscreants can request replacements for accounts that are dysfunctional. We provide a detailed breakdown of the merchants we identify and their source of solicitation in Table 1. We make no claim our search for merchants is exhaustive; nevertheless, the sellers we identify provide an insightful cross-section of the varying levels of sophistication required to circumvent automated account registration barriers, outlined in detail in Section 4.

3.2 Purchasing from Merchants

Once we identify a merchant, we place an initial test purchase to determine the authenticity of the accounts being sold. If genuine, we then determine whether to repeatedly purchase from the merchant based on the quality of accounts provided (discussed in Section 4) and the overall impact the seller has on Twitter spam (discussed in Section 6). As such, our purchasing is an iterative process where each new set of accounts improves our understanding of the market and subsequently directs our investigation.

Once we vet a merchant, we conduct purchases on a bi-weekly basis beginning in June, 2012 (at the earliest) up to the time of our analysis in April, 2013, detailed in Table 1. We note that purchasing at regular intervals is not always feasible due to logistical issues such as merchants delaying delivery or failing to respond to requests for accounts. In summary, we place 144 orders (140 of which merchants successfully respond to and fulfill) for a total of 120,019 accounts. Purchases typically consist of a bulk order for 1,000 accounts, though sellers on Fiverr operate in far less volume.

Throughout this process, we protect our identity from merchants by using a number of email and Skype pseudonyms. We conduct payments through multiple identities tied to PayPal, WebMoney, and pre-paid credit

²Pre-aged accounts allow miscreants to evade heuristics that disable newly minted accounts based upon weak, early signs of misbehavior. In contrast, in order to limit the impact on legitimate users, disabling older accounts only occurs in the face of much stronger signals of malefice.

Merchant	Period	#	Accts	Price
alexissmalley [†]	06/12–03/13	14	13,000	\$4
naveedakhtar [†]	01/13–03/13	4	2,044	\$5
truepals [†]	02/13–03/13	3	820	\$8
victoryservices [†]	06/12–03/13	15	15,819	\$6
webmentors2009 [†]	10/12–03/13	9	9,006	\$3–4
buuman ^{II}	10/12–10/12	1	75	\$7
danyelgallu ^{II}	10/12–10/12	1	74	\$7
denial93 ^{II}	10/12–10/12	1	255	\$20
formefor ^{II}	09/12–11/12	3	408	\$2–10
ghetumarian ^{II}	09/12–10/12	3	320	\$4–5
jackhack08 ^{II}	09/12–09/12	2	755	\$1
kathlyn ^{II}	10/12–10/12	1	74	\$7
smokinbluelady ^{II}	08/12–08/12	1	275	\$2
twitfollowers ^{II}	10/12–10/12	1	80	\$6
twitter007 ^{II}	10/12–10/12	1	75	\$7
kamalkishover [◦]	06/12–03/13	14	12,094	\$4–7
shivnagsudhakar [◦]	06/12–06/12	1	1,002	\$4
accs.biz [‡]	05/12–03/13	15	17,984	\$2–3
buyaccountsnow.com [‡]	06/12–11/12	8	7,999	\$5–8
buyaccs.com [‡]	06/12–03/13	14	13,794	\$1–3
buytwitteraccounts.biz [‡]	09/12–10/12	3	2,875	\$5
buwtwitteraccounts.info [‡]	10/12–03/13	9	9,200	\$3–4
dataentryassistant.com [‡]	10/12–03/13	9	5,498	\$10
getbulkaccounts.com [‡]	09/12–09/12	1	1,000	\$2
quickaccounts.bigcartel [‡]	11/12–11/12	2	1,501	\$3
spamvilla.com [‡]	06/12–10/12	3	2,992	\$4
xlinternetmarketing.com [‡]	10/12–10/12	1	1,000	\$7
Total	05/12–03/13	140	120,019	\$1–20

Table 1: List of the merchants we track, the months monitored, total purchases performed (#), accounts purchased, and the price per 100 accounts. Source of solicitations include blackhat forums[†], Fiverr^{II}, and Freelancer[◦] and web storefronts[‡].

cards. Finally, we access all web content on a virtual machine through a network proxy.

3.3 Account Pricing & Availability

Prices through the course of our analysis range from \$0.01 to \$0.20 per Twitter account, with a median cost of \$0.04 for all merchants. Despite the large overall span, prices charged by individual merchants remain roughly stable. Table 1 shows the variation in prices for six merchants we tracked over the longest period of time. Price hikes are a rare occurrence and no increase is more than \$0.03 per account. So long as miscreants have money on hand, availability of accounts is a non-issue. Of the orders we placed, merchants fulfilled 70% in a day and 90% within 3 days. We believe the stable pricing and ready availability of fraudulent accounts is a direct result

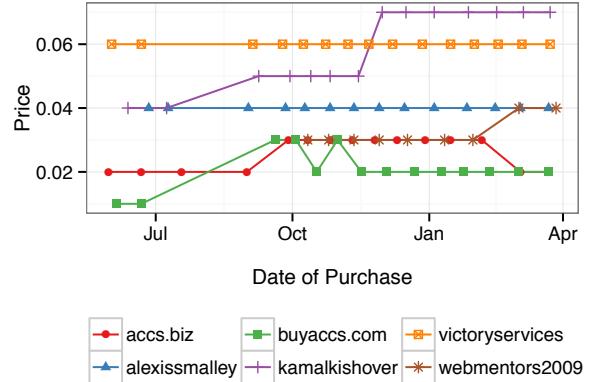


Figure 1: Variation in prices over time for six merchants we track over the longest period of time.

of minimal adversarial pressures on account merchants, a hypothesis we explore further in Section 4.

3.4 Other Credentials For Sale

Our permission to purchase accounts is limited to Twitter credentials, but many of the merchants we interact with also sell accounts for Facebook, Google, Hotmail, and Yahoo. We compare prices between web services, but note that as we cannot vet non-Twitter credentials, some prices may represent scams.

Facebook Prices for Facebook accounts range from \$0.45–1.50 per *phone verified account* (PVA) and \$0.10 for non-PVA accounts. Phone verification requires that miscreants tie a SIM card to a newly minted Facebook account and verify the receipt of a text message, the complexities of which vastly increase the price of an account.³ For those sellers that advertise their registration process, SIM cards originate from Estonia or Ukraine.

Google Prices for Google PVA accounts range from \$0.03–0.50 per account.

Hotmail Prices for Hotmail accounts cost \$0.004 – 0.03 per account, a steep reduction over social networking or PVA credentials. We see similar prices for a multitude of web mail providers, indicating that email accounts are in demand and cheaper to create.

Yahoo Yahoo accounts, like Hotmail, are widely available, with prices ranging from \$0.006 – 0.015 per account.

³Advertisements that we encountered for phone verification services ranged in price from \$.10 – \$.15 per verification for bulk orders of 100,000 verifications, and \$.25 per verification for smaller orders.

Merchant	Reaccessed	Resold
getbulkaccounts.com	100%	100%
formeфор	100%	99%
denial93	100%	97%
shivnagsudhakar	98%	98%
quickaccounts.bigcartel.com	67%	64%
buytwitteraccounts.info	39%	31%
ghetumarian	30%	28%
buytwitteraccounts.biz	20%	18%
jackhack08	12%	11%
buyaccountsnow.com	10%	1%
kamalkishover	8%	0%
buyaccts.com	7%	4%
alexissmalley	6%	0%
victoryservices	3%	2%
Total	10%	6%

Table 2: List of dishonest merchants that reaccessed and resold credentials we purchased to other parties.

3.5 Merchant Fraud

Operating in the underground market is not without risk of fraud and dishonesty on the part of account merchants. For instance, eight of the merchants we contacted attempted to sell us a total of 3,317 duplicate accounts. One merchant even schemed to resell us the same 1,000 accounts three times. For those merchants willing to honor their “48 hours of support”, we requested replacement accounts for duplicates, bringing our account total up to 121,027 unique credentials.

Apart from duplicate credentials, some merchants were quick to resell accounts we purchased to third parties. In order to detect resales, we coordinate with Twitter to monitor all successful logins to accounts we purchase after they come under our control. We denote these accounts *reaccessed*. We repeat this same process to detect new tweets or the formation of relationships. Such behaviors should only occur when an account changes hands to a spammer, so we denote these accounts as *resold*. Such surreptitious behavior is possible because we make a decision not to change the passwords of accounts we purchase.

Table 2 shows the fraction of purchased accounts per seller that merchants reaccessed and resold. A total of 10% of accounts in our dataset were logged into (either by the seller or a third party; it is not possible to distinguish the two) within a median of 3 days from our purchase. We find that 6% of all accounts go on to be resold in a median of 5 days from our purchase. This serves to highlight that some merchants are by no means shy about scamming potential customers.

4 Fraudulent Registration Analysis

Account merchants readily evade existing abuse safeguards to register thousands of accounts on a recurring basis. To understand these failings, we delve into the tools and techniques required to operate in the account marketplace. We find that merchants leverage thousands of compromised hosts, CAPTCHA solvers, and access to fraudulent email accounts. We identify what registration barriers increase the price of accounts and summarize our observations into a set of recommendations for how web services can improve existing automation barriers to increase the cost of fraudulent credentials in the future.

4.1 Dataset Summary

To carry out our analysis, we combine intelligence gathered from the underground market with private data provided through a collaboration with Twitter. Due to the sensitivity of this data, we strictly adhere to a data policy set down by Twitter, documented in Appendix A. In total, we have the credentials for 121,027 purchased accounts, each of which we annotate with the seller and source of solicitation. Furthermore, we obtain access to each account’s associated email address; login history going back one year including IP addresses and timestamps; signup information including the IP and user agent used to register the account; the history of each account’s activities including tweeting or the formation of social connections, if any; and finally whether Twitter has flagged the account as spam (independent of our analysis).

4.2 Circumventing IP Defenses

Unique IP addresses are a fundamental resource for registering accounts in bulk. Without a diverse IP pool, fraudulent accounts would fall easy prey to network-based blacklisting and throttling [12, 18, 35]. Our analysis leads us to believe that account merchants either own or rent access to thousands of compromised hosts to evade IP defenses.

IP Address Diversity & Geolocation As a whole, miscreants registered 79% of the accounts we purchase from unique IP addresses located across the globe. No single subnet captures the majority of abused IPs; the top ten /24 subnets account for only 3% of signup IPs, while the top ten /16 subnets account for only 8% of registrations. We provide a breakdown of geolocations tied to addresses under the control of merchants in Table 3. India is the most popular origin of registration, accounting for 8.5% of all fraudulent accounts in our dataset.

Registration Origin	Unique IPs	Popularity
India	6,029	8.50%
Ukraine	6,671	7.23%
Turkey	5,984	5.93%
Thailand	5,836	5.40%
Mexico	4,547	4.61%
Viet Nam	4,470	4.20%
Indonesia	4,014	4.10%
Pakistan	4,476	4.05%
Japan	3,185	3.73%
Belarus	3,901	3.72%
Other	46,850	48.52%

Table 3: Top 10 most popular geolocations of IP addresses used to register fraudulent accounts.

Other ‘low-quality’ IP addresses (e.g. inexpensive hosts from the perspective of the underground market [4]) follow in popularity. In summary, registrations come from 164 countries, the majority of which serve as the origin of fewer than 1% of accounts in our dataset. However, in aggregate, these small contributors account for 48.5% of all registered accounts.

Merchants that advertise on blackhat forums or operate their own web storefronts have the most resources at their disposal, registering all but 15% of their accounts via unique IPs from hundreds of countries. Conversely, merchants operating on Fiverr and Freelancer tend to operate solely out of the United States or India and reuse IPs for at least 30% of the accounts they register.

Long-term IP Abuse To understand the long-term abuse of IP addresses, we analyze data provided by Twitter that includes *all* registered accounts (not just our purchases) from June, 2012 – April, 2013. From this, we select a random sample of 100,000 unique IPs belonging to accounts that Twitter has disabled for spamming (e.g. suspended) and an equally sized sample of IPs used to register legitimate Twitter accounts. We add a third category to our sample that includes all the unique IP addresses used by merchants to register the accounts we purchased. For each of these IPs, we calculate the total number of Twitter accounts registered from the same IP.

A CDF of our results, shown in Figure 2, indicates merchants use the IP addresses under their control to register an abnormal number of accounts. Furthermore, the merchants we track are more cautious than other Twitter spammers who register a larger volume of accounts from a single IP address, making the merchants harder to detect. In total, merchants use 50% of the IP addresses under their control to register fewer than 10 accounts,

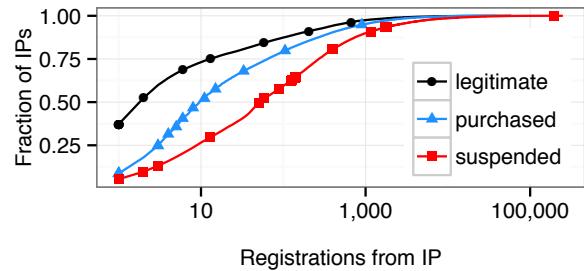


Figure 2: CDF of registrations per IP tied to purchased accounts, legitimate accounts, and suspended (spam) accounts.

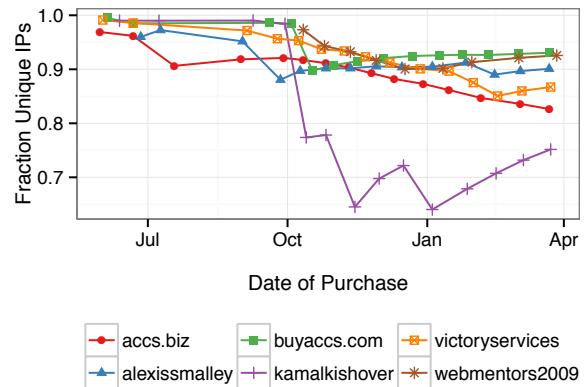


Figure 3: Availability of unique IPs over time for the six merchants we track over the longest period. All but one seller we repeatedly purchase from are able to acquire new IP addresses to register accounts from over time.

compared to 73% of IPs tied to legitimate users and only 26% for other spammers. We note that the small fraction of legitimate IP addresses used to register thousands of accounts likely belong to mobile providers or other middleboxes.

IP Churn & Pool Size In order to sustain demand for new accounts without overextending the abuse of a single IP address, merchants obtain access to tens of thousands of IP addresses that change over time. Figure 3 shows the fraction of accounts we purchase that appear from a unique IP address⁴ as a function of time. We restrict our analysis to the six merchants we track over the longest period. Despite successive purchases of 1,000 accounts, all but one seller maintains IP uniqueness above roughly 80% of registered accounts, indicating that the IPs available to merchants change over time.

⁴We calculate uniqueness over the IP addresses in our dataset, not over all IPs used to register accounts on Twitter.

We calculate the number of IP addresses under each merchant’s control by treating IP reuse as a *closed capture-recapture* problem. Closed capture-recapture measurements – used to estimate an unknown population size – require (1) the availability of independent samples and (2) that the population size under study remains fixed. To begin, we assume each purchase we make is an independent sample of the IP addresses under a merchant’s control, satisfying the first requirement. The second requirement is more restrictive. If we assume that merchants use IP addresses tied to compromised hosts, then there is an inherent instability in the population size of IPs due to hosts becoming uninfected, new hosts becoming infected, and ISPs reallocating dynamic IPs. As such, comparisons over long periods are not possible. Nevertheless, if we restrict our analysis to batches of accounts from a single seller that were all registered within 24 hours, we can minimize the imprecision introduced by IP churn.

To this end, we select clusters of over 300 accounts registered by merchants within a 24 hour window. We split each cluster in half by time, with the first half m acting as the set of marked IPs and the second set c as the captured IPs, where there are r overlapping, or re-captured, IPs between both sets. We can then estimate the entire population size \hat{N} (e.g. the number of unique IPs available to a merchant) according to the Chapman-Petersen method [24]:

$$\hat{N} = \frac{(m+1)(c+1)}{(r+1)} - 1$$

And standard error according to:

$$SE = \sqrt{\frac{\hat{N}^2(c-r)}{(c+1)(r+2)}}$$

For 95% confidence intervals, we calculate the error of \hat{N} as $\pm 1.96 \times SE$. We detail our results in Table 4. We find that sellers like *accs.biz* and *victoryservices* have tens of thousands of IPs at their disposal on any given day, while even the smallest web storefront merchants have thousands of IPs on hand to avoid network-based blacklisting and throttling.

4.3 CAPTCHAs & Email Confirmation

Web services frequently inhibit automated account creation by requiring new users to solve a CAPTCHA or confirm an email address. Unsurprisingly, we find neither of these barriers are insurmountable, but they *do* impact the pricing and rate of generation of accounts, warranting their continued use.

Merchant	\hat{N} Estimate	\pm Error
accs.biz	21,798	4,783
victoryservices	17,029	2,264
dataentryassistant.com	16,887	4,508
alexissmalley	16,568	3,749
webmentors2009	10,019	2,052
buyaccts.com	9,770	3,344
buylwitteraccounts.info	6,082	1,661
buyaccountsnow.com	5,438	1,843
spamvilla.com	4,646	1,337
kamalkishover	4,416	1,170

Table 4: Top 10 merchants with the largest estimated pool of IP addresses under their control on a single day.

Email Confirmation All but 5 of the merchants we purchase from readily comply with requirements to confirm email addresses through the receipt of a secret token. In total, merchants email confirm 77% of accounts we acquire, all of which they seeded with a unique email. The failure of email confirmation as a barrier directly stems from pervasive account abuse tied to web mail providers. Table 5 details a list of the email services frequently tied to fraudulent Twitter accounts. Merchants abuse Hotmail addresses to confirm 60% of Twitter accounts, followed in popularity by Yahoo and mail.ru. This highlights the interconnected nature of account abuse, where credentials from one service can serve as keys to abusing yet another.

While the ability of merchants to verify email addresses may raise questions of the processes validity, we find that email confirmation positively impacts the price of accounts. Anecdotally, Hotmail and Yahoo accounts are available on *blackhatworld.com* for \$6 per thousand, while Twitter accounts from the same forum are \$40 per thousand. This is also true of web storefront such as *buy-accts.com* where mail.ru and Hotmail accounts are \$5 per thousand, compared to \$20 per thousand for Twitter accounts. Within our own dataset, we find that Twitter accounts purchased without email confirmation cost on average \$30 per thousand compared to \$47 per thousand for accounts with a confirmed email address. This difference likely includes the base cost of an email address and any related overhead due to the complexity of responding to a confirmation email.

CAPTCHA Solving Twitter throttles multiple registrations originating from a single IP address by requiring a CAPTCHA solution. Merchants solved a CAPTCHA for 35% of the accounts we purchase; the remaining accounts were registered from fresh IPs that did not trigger throttling. While there are a variety of CAPTCHA solving

Email Provider	Accounts	Popularity
hotmail.com	64,050	60.08%
yahoo.com	12,339	11.57%
mail.ru	12,189	11.43%
gmail.com	2,013	1.89%
nokiamail.com	996	0.93%
Other	2,157	0.14%

Table 5: Top 5 email providers used to confirm fraudulent Twitter accounts.

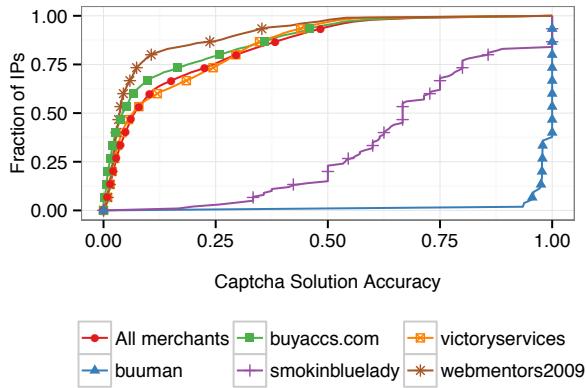


Figure 4: CAPTCHA solution rates per each IP address abused by a variety of merchants as well as the rates for all merchants combined.

services available in the underground market [19], none are free and thus requiring a CAPTCHA slightly increases the cost of creating fraudulent accounts.

A second aspect of CAPTCHAs is the success rate of automated or human solvers. By virtue of only buying successfully registered accounts, we cannot exactly measure CAPTCHA failure rates (unless account sellers fail and re-try a CAPTCHA during the same registration session, something we find rare in practice). However, we can examine registration attempts that occur from the same IPs as the accounts we purchase to estimate the rate of failure. To carry out this analysis, we examine all registrations within the previous year, calculating the fraction of registrations that fail due to incorrect CAPTCHA solutions per IP address.

We show a CDF of CAPTCHA solution rates for a sample of merchants in Figure 4. The median CAPTCHA solution rate for all sellers is 7%, well below estimates for automated CAPTCHA solving software of 18–30% [19], a discrepancy we currently have no explanation for. For two of the Fiverr sellers, *buuman* and *smokinbluelady*, the median CAPTCHA solution rate per IP is 100% and

67% respectively, which would indicate a human solver. In total, 92% of all throttled registration attempts from merchants fail. Despite this fact, account sellers are still able to register thousands accounts over the course of time, simply playing a game of odds.

4.4 Stockpiling & Suspension

Without effective defenses against fraudulent account registration, merchants are free to stockpile accounts and sell them at a whim. For many solicitations, merchants consider “pre-aged” accounts a selling point, not a detraction. To highlight this problem, we examine the failure of at-abuse time metrics for detecting dormant accounts and the resulting account stockpiles that occur.

Account Suspension Twitter suspends (e.g. disables) spam accounts due to at-abuse time metrics such as sending spam URLs or generating too many relationships, as outlined in Twitter’s rules [30]. In our case, we are interested in whether fraudulent accounts that do *not* perform visible spam actions (e.g. are dormant) nevertheless become suspended. While for miscreants this should ideally be impossible, there are multiple avenues for guilt by association, such as clustering accounts based on registration IP addresses or other features. As such, when Twitter suspends a large volume of active fraudulent accounts for spamming, it is possible for Twitter to catch dormant accounts in the same net.

Of the dormant accounts we purchase, only 8% are eventually detected and suspended. We exclude accounts that were resold and used to send spam (outlined in Section 3.5) from this metric in order to not skew our results. Of the merchants we track, Fiverr sellers take the least caution in registering unlinkable accounts, resulting in 57% of our purchases becoming suspended by the time of our analysis. In contrast, web storefronts leverage the vast resources at their disposal to create unlinkable accounts, where only 5% of our purchased accounts are eventually detected as fraudulent. These poor detection rates highlight the limitation of at-abuse time metrics against automated account registration. Without more sophisticated at-registration abuse signals, merchants are free to create thousands of accounts with minimal risk of Twitter suspending back stock.

Account Aging & Stockpiling We examine the age of accounts, measured as the time between their registration and subsequent date of purchase, and find that accounts are commonly stockpiled for a median of 31 days. While most merchants deal exclusively in back stock, some merchants operate in an on-demand fashion. At the far end of this spectrum is a merchant *spamvilla.com*

that sold us accounts registered a median of 323 days ago – nearly a year in advance of our purchase. In contrast, webstores such as *buyaccts.com* and Fiverr merchants including *smokinbluelady* sell accounts less than a day old. Even though these merchants operate purely on-demand, they are still able to fulfill large requests in short order (within a day in our experience). Both modes of operation illustrate the ease that merchants circumvent existing defenses and the need for at-registration time abuse detection.

4.5 Recommendations

Web services that rely on automation barriers must strike a tenuous balance between promoting user growth and preventing the proliferation of fraudulent accounts and spam behavior. We summarize our findings in this section with a number of potential improvements to existing barriers that should not impede legitimate users. While we draw many of our observations from the Twitter account abuse problem, we believe our recommendations should generalize across web services.

Email Confirmation While account merchants have cheap, disposable emails on hand to perform email confirmation, confirmation helps to increase the cost of fraudulent accounts. In the case of Twitter, email confirmation raises the cost of accounts by 56%. Furthermore, in the absence of clear abuse signals, services can use *email reconfirmation* as a *soft action* against automation, similar to requiring a CAPTCHA before sending an email or tweet. Of the Twitter accounts we purchased, only 47% included the email address and password used to confirm the account. Merchants will sometimes re-appropriate these email addresses and sell them as “second-hand” at a discount of 20%. Without the original credentials, miscreants will be unable to perform email reconfirmation. Even if merchants adapt and begin to provide email credentials as part of their sale, the possibility of reselling email addresses disappears, cutting into a merchant’s revenue.

CAPTCHAs CAPTCHAs serve to both increase the cost of accounts due to the requirement of a CAPTCHA solving service as well as to throttle the rate of account creation. In our experience, when required, CAPTCHAs prevent merchants from registering 92% of fraudulent accounts. Services could also leverage this failure rate as a signal for blacklisting an IP address in real-time, cutting into the number of accounts merchants can register from a single IP.

IP Blacklisting While miscreants have thousands of IP addresses at their disposal that rapidly change, IP blacklisting is not without merit. Our results show that merchants use a small fraction of IPs to register tens of thousands of accounts, which services could curb with real-time blacklisting. While public and commercial IP blacklists exist such as CBL [5], previous work has shown these generate too many false positives in the case of social spam [28], requiring service providers to generate and maintain their own blacklists.

Phone Verification While Twitter does not require phone verification, we observe the positive impact phone verification has on increasing the cost of fraudulent accounts for other services. Facebook and GMail accounts that are phone verified cost up to 150x more than their Twitter, non-PVA counterpart. As with CAPTCHAs or email reconfirmation, phone verification can serve as a soft action against spammers who do not clearly fall into the set of accounts that should be automatically disabled.

5 Detecting Fraudulent Registrations

To understand the impact account merchants have on Twitter spam, we develop a classifier trained on purchased accounts to *retroactively* identify abusive registrations. Our technique relies on identifying patterns in the naming conventions and registration process used by merchants to automatically generate accounts. We apply our classifier to *all* Twitter accounts registered in the last year (overlapping with our investigation) and identify several million accounts which appear to be fraudulent. We note this approach is *not* meant to sustain accuracy in an adversarial setting; we only apply it to historical registrations where adaptation to our signals is impossible.

5.1 Automatic Pattern Recognition

Our detection framework begins by leveraging the limited variability in naming patterns used by account generation algorithms which enables us to automatically construct regular expressions that fingerprint fraudulent accounts. Our approach for generating these expressions is similar to previous techniques for identifying spam emails based on URL patterns [32] or spam text templates [22, 23]. However, these previous approaches fail on small text corpuses (e.g. screennames), especially when samples cannot be linked by repeating substrings. For this reason, we develop a technique explicitly for account naming patterns. Algorithm 1 shows a sketch of our approach which we use to guide our discussion.

Algorithm 1 Generate Merchant Pattern

Input: List of accounts for a single merchant
Parameters: τ (minimum cluster size)
clusters \leftarrow GROUP accounts BY
 (Σ -Seq, repeatedNames, emailDomain)
for all cluster \in clusters **do**
 if cluster.size() $>$ τ **then**
 patterns \leftarrow MINMAX Σ -SEQ (cluster)
 OUTPUTREGEX(patterns, repeatedNames)
 end if
end for

Common Character Classes To capture accounts that all share the same naming structure, we begin by defining a set of character classes:

$$\Sigma = \{p\{Lu\}, p\{Ll\}, p\{Lo\}, d, \dots\}$$

composed of disjoint sets of characters including uppercase Unicode letters, lowercase Unicode letters, non-cased Unicode letters (e.g., Arabic), and digits.⁵ We treat all other characters as distinct classes (e.g., +, -, _). We chose these character classes based on the naming patterns of accounts we purchase, a sample of which we show in Table 6. We must support Unicode as registration algorithms draw account names from English, Cyrillic, and Arabic.

From these classes we define a function Σ -Seq that captures transitions between character classes and produces an ordered set $\sigma_1\sigma_2\dots\sigma_n$ of arbitrary length, where σ_i represents the i -th character class in a string. For example, we interpret the account Wendy Hunt from *accs.biz* as a sequence $p\{Lu\}p\{Ll\} p\{Lu\}p\{Ll\}$. We repeat this process for the name, screenname, and email of each account. We note that for emails, we strip the email domain (e.g. @hotmail.com) prior to processing and use this as a separate feature in the process for pattern generation.

Repeated Substrings While repeated text stems between multiple accounts are uncommon due to randomly selected dictionary names, we find the algorithms used to generate accounts often reuse portions of text for names, screennames, and emails. For instance, all of the accounts in Table 6 from *victoryservices* have repeated substrings between an account’s first name and screenname.

To codify these patterns, we define a function *repeatedNames* that canonicalizes text from an account’s fields, brute forces a search of repeated substrings, and then codifies the resulting patterns as invariants. Canonicalization entails segmenting a string into multiple sub-

strings based on Σ -Seq transitions. We preserve full names by ignoring transitions between upper and lowercase letters; spaces are also omitted from canonicalization. We then convert all substrings to their lowercase equivalent, when applicable. To illustrate this process, consider the screenname WendyHunt5. Canonicalization produces an ordered list [wendy,hunt,5], while the name Wendy Hunt is converted to [wendy,hunt].

The function repeatedNames proceeds by performing a brute force search for repeated substrings between all canonicalized fields of an account. For our previous example of WendyHunt5, one successful match exists between name[1] and screenname[1], where [i] indicates the i -th position of a fields substring list; this same pattern also holds for the name and screenname for Kristina Levy. We use this positional search to construct invariants that hold across accounts from a single merchant. Without canonicalization, we could not specify what relationship exists between Wendy and Kristina due to differing text and lengths. When searching, we employ both exact pattern matching as well as partial matches (e.g. neff found in brindagtneff for *buyaccs.com*). We use the search results to construct invariants for both strings that must repeat as well as strings that never repeat.

Clustering Similar Accounts Once we know the Σ -Seq, repeatedNames, and email domain of every account from a merchant, we cluster accounts into non-overlapping groups with identical patterns, as described in Algorithm 1. We do this on a per-merchant basis rather than for every merchant simultaneously to distinguish which merchant an account originates from. We prune small clusters based on a empirically determined τ to reduce false positives, with our current implementation dropping clusters with fewer than 10 associated accounts.

Bounding Character Lengths The final phase of our algorithm strengthens the invariants tied to Σ -Seq transitions by determining a minimum length $\min(\sigma_i)$ and maximum length $\max(\sigma_i)$ of each character class σ_i . We use these to define a bound $\{l_{\min}, l_{\max}\}$ that captures all accounts with the same Σ -Seq. Returning to our examples in Table 6, we group the account names from *accs.biz* and produce an expression $p\{Lu\}\{1, 1\}p\{Ll\}\{5, 8\} \{1, 1\}p\{Lu\}\{1, 1\}p\{Ll\}\{4, 4\}$. We combine these patterns with the invariants produced by repeatedNames to construct a regular expression that fingerprints a cluster. We refer to these patterns for the rest of this paper as *merchant patterns*.

⁵We use Java character class notation, where p{*} indicates a class of letters and Lu indicates uppercase, Ll lowercase, and Lo non-case.

Seller	Popularity	Name	Screenname	Email
victoryservices	57%	Trstram Aiken	Trstramsse912	KareyKay34251@hotmail.com
		Millicent Comolli	Millicentrpq645	DanHald46927@hotmail.com
accs.biz	46%	Wendy Hunt	WendyHunt5	imawzgaf7083@hotmail.com
		Kristina Levy	KristinaLevy6	exraytj8143@hotmail.com
formefor	43%	ola dingess	olawhdingess	TimeffTicnisha@hotmail.com
		brinda neff	brindagtgneff	ScujheShananan@hotmail.com
spamvilla.com	38%	Kiera Barbo	Kieravyvdb	LinJose344@hotmail.com
		Jeannine Allegrini	Jeanninewoqzg	OpheliaStar461@hotmail.com

Table 6: Obfuscated sample of names, screennames, and emails of purchased accounts used to automatically generate merchant patterns. Popularity denotes the fraction of accounts that match the pattern for an individual merchant.

5.2 Pattern Refinement

We refine our merchant patterns by including abuse-orientated signals that detect automated signup behavior based on the registration process, user-agent data, and timing events.

Signup Flow Events We begin our refinement of merchant patterns by analyzing the activities of purchased accounts during and immediately after the signup work flow. These activities include events such as a user importing contacts and accessing a new user tutorial. The complete list of these events is sensitive information and is omitted from discussion. Many of these events go untriggered by the automated algorithms used by account sellers, allowing us to distinguish automated registrations from legitimate users.

Given a cluster of accounts belonging to a single merchant, we generate a binary feature vector $\mathbf{e}_{sig} = \{0, 1\}^n$ of the n possible events triggered during signup. A value of 1 indicates that at least ρ accounts in the cluster triggered the event e . For our experiments, we specify a cutoff $\rho = 5\%$ based on reducing false positives. Subsequently, we determine whether a new account with event vector \mathbf{e} matches a seller’s signup flow signature \mathbf{e}_{sig} by computing whether $\mathbf{e} \subseteq \mathbf{e}_{sig}$ holds. The majority of legitimate accounts have $|\mathbf{e}| \gg |\mathbf{e}_{sig}|$, so we reject the possibility they are automated even though their naming conventions may match a merchant’s.

User Agents A second component of signups is the user agent associated with a form submission. Direct matching of user agents used by a seller with new subsequent signups is infeasible due to sellers randomizing user agents. For instance, *buytwitteraccounts.info* uses a unique (faked) agent for every account in our purchased dataset. Nevertheless, we can identify uniformity in the naming conventions of user agents just as we did with account names and screennames.

Given a cluster of accounts from a single seller, we generate a *prefix tree* containing every account’s user agent. A node in the tree represents a single character from a user agent string while the node’s depth mirrors the character’s position in the user agent string. Each node also contains the fraction of agents that match the substring terminated at the given node. Rather than find the longest common substring between all accounts, we prune the tree so that every substring terminating at a node has a fraction of at least ϕ accounts in the cluster (in practice, 5%). We then generate the set of all substrings in the prefix tree and use them to match against the agents of newly registered accounts. The resulting substrings include patterns such as Mozilla/5.0 (X11; Linux i686 which, if not truncated, would include multiple spurious browser toolbars and plugins and be distinct from subsequent signups. While in theory the resulting user agent substrings can be broad, in practice we find they capture browser variants and operating systems before being truncated.

Form Submission Timing The final feature from the signup process we use measures the time between Twitter serving a signup form to the time the form is submitted. We then compute a bound $\{\min_{ts}, \max_{ts}\}$ for each seller to determine how quickly a seller’s algorithm completes a form. To counter outliers, we opt for the 99% for both minimum and maximum time. For instance, the Fiverr merchant *kathlyn* registers accounts within $\{0, 1\}$ seconds. A newly minted account can match a seller’s algorithm if its form completion time is within the sellers bound.

5.3 Alternative Signals

There were a number of alternative signals we considered, but ultimately rejected as features for classification. We omitted the delay between an account’s registration and subsequent activation as we lacked training

data to measure this period; all our accounts remain dormant after purchase (minus the small fraction that were resold). We also analyzed both the timing of registrations as well as the interarrival times between successive registrations. We found that merchants sell accounts in blocks that sometimes span months, preventing any interarrival analysis. Furthermore, merchants register accounts at uniformly random hours and minutes. Finally, as merchants create accounts from IP addresses around the globe, no subnet or country accurately captures a substantive portion of abusive registrations.

5.4 Evaluation

To demonstrate the efficacy of our model, we retroactively apply our classifier to *all* Twitter accounts registered in the last year. In total, we identify several million⁶ distinct accounts that match one of our merchant patterns and thus are potentially fraudulent. We validate these findings by analyzing both the *precision* and *recall* of our model as well measuring the impact of time on the model’s overall accuracy.

Precision & Recall Precision measures the fraction of identified accounts that are in fact fraudulent (e.g., not misclassified, legitimate users), while recall measures the fraction of all possible fraudulent accounts that we identify, limited to the merchants that we study. To estimate the precision of each merchant pattern, we select a random sample of 200 accounts matching each of 26 merchant patterns,⁷ for a total of 4,800 samples. We then manually analyze the login history, geographic distribution of IPs, activities, and registration process tied to each of these accounts and label them as spam or benign. From this process, we estimate our overall precision at 99.99%, with the breakdown of the most popular merchant pattern precisions shown in Table 7. In a similar vein, we estimate recall by calculating the fraction of all accounts we purchase that match our classifier. In total, we correctly identify 95% of all purchased accounts; the remaining 5% of missed accounts did not form large enough clusters to be included in a merchant’s pattern, and as a result, we incorrectly classified them as legitimate.

Performance Over Time The performance of our model is directly tied to accurately tracking adaptations in the

⁶Due to operational concerns, we are unable to provide exact numbers on the volume of spam accounts registered. As such, we reference merchants and the impact they have on Twitter as a *relative volume* of all several million accounts that we detect.

⁷We omit accounts purchased from the Freelancer merchant *shivnagsudhakar* as these were registered over a year ago and thus lay outside the range of data to which we had access.

Service	Rel. Volume	P	R
buuman	0.00%	100.00%	70.67%
smokinbluelady	0.08%	100.00%	98.91%
danyelgallu	0.12%	100.00%	100.00%
twitter007	0.13%	100.00%	97.33%
kathlyn	0.13%	100.00%	93.24%
jackhack08	0.41%	100.00%	100.00%
twitfollowers	0.72%	100.00%	92.50%
denial93	2.18%	100.00%	100.00%
ghetumarian	3.05%	100.00%	85.94%
formefor	4.75%	100.00%	100.00%
shivnagsudhakar	—	—	—
kamalkishover	29.90%	99.60%	92.73%
naveedakhtar	0.24%	100.00%	98.40%
webmentors2009	0.85%	100.00%	99.64%
truepals	1.02%	100.00%	93.08%
alexissmalley	1.68%	100.00%	98.62%
victoryservices	6.33%	99.70%	99.03%
spamvilla.com	0.71%	99.00%	98.70%
getbulkaccounts.com	2.97%	100.00%	100.00%
xlinternetmarketing.com	3.12%	100.00%	95.13%
accs.biz	4.48%	100.00%	97.62%
buytwitteraccounts.biz	6.10%	100.00%	84.27%
quickaccounts.bigcartel	10.91%	100.00%	99.73%
buytwitteraccounts.info	20.45%	99.60%	81.85%
dataentryassistant.com	24.01%	100.00%	96.57%
buyaccountsnow.com	30.75%	99.10%	95.10%
buyaccs.com	58.39%	100.00%	91.66%
Total	100.00%	99.99%	95.08%

Table 7: Breakdown of the merchants, the relative volume of all detected accounts in the last year that match their pattern, precision (P) and recall (R).

algorithms used by merchants to register accounts. To understand how frequently these adaptations occur, we evaluate the performance of our classifier as a function of time. Figure 5 shows the overall recall of each of our merchant patterns for the sellers we track over the longest period of time. For each merchant, we train a classifier on accounts acquired up to time t and evaluate it on all accounts from the merchant, regardless of when we purchased the account. We find that some sellers such as *alexissmalley* rarely alter their registration algorithm throughout our study, allowing only two purchases to suffice for accurate detection. In contrast, we see a shift in registration algorithms for a number of merchants around October and January, but otherwise patterns remain stable for long periods. The several million accounts we identify as fraudulent should thus be viewed as a lower bound in the event we missed an adaptation.

Pattern Overlap & Resale The simultaneous adaptation of merchant patterns in Figure 5 around October and

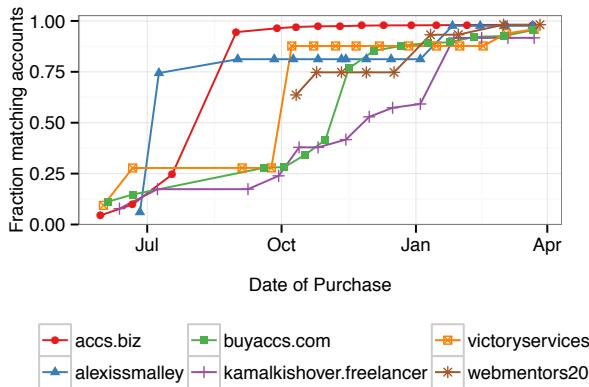


Figure 5: Recall of generated merchant patterns for all purchased accounts as a function of training the classifier on data only prior to time t .

other periods leads us to believe that a multitude of merchants are using the same software to register accounts and that an update was distributed. Alternatively, the account marketplace may have multiple levels of resale (or even arbitrage) where accounts from one merchant are resold by another for an increased cost, leading to correlated adaptations. Further evidence of correlated patterns appears in the merchant patterns we construct, where a classifier for one merchant will accurately detect accounts sold to us by a second merchant. For instance, the accounts sold by *kamalkishover* from Freelancer overlap with the patterns of 9 other merchants, the most popular of which is *buyaccountsnow.com*. We find most Fiverr sellers are independent with the exception of *denial93*, *ghetumarian*, and *formefor*, whose patterns overlap with the major account web storefronts. This would explain why these three Fiverr sellers appear to be much larger (from the perspective of Table 7) compared to other Fiverr merchants. As a result, our estimates for the number of accounts registered by each merchant may be inflated, though our final total counts only unique matches and is thus globally accurate.

6 Impact of the Underground Market

We analyze the several million accounts we flag as registered by merchants operating in the underground market and estimate the fraction that have been sold and used to generate Twitter spam. We find that, during active months, the underground market was responsible for registering 10–20% of all accounts that Twitter later flagged as spam. For their efforts, we estimate that merchants generated a combined revenue between \$127,000–\$459,000.

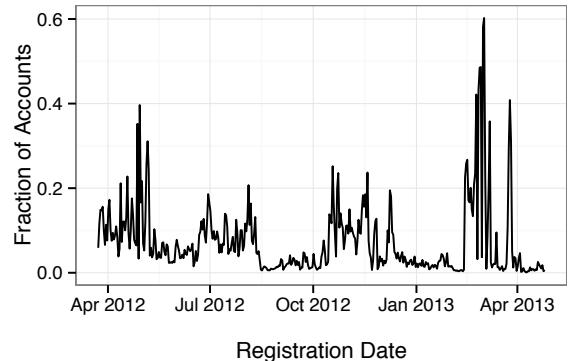


Figure 6: Fraction of all suspended accounts over time that originate from the underground market.

6.1 Impact on Twitter Spam

From our seed set of 121,027 accounts purchased from 27 merchants, we are able to identify several million fraudulent accounts that were registered by the same merchants. Of these, 73% were sold and actively tweeting or forming relationships at one point in time, while the remaining 37% remained dormant and were yet to be purchased.

In cooperation with Twitter, we analyzed the total fraction of all suspended accounts that appear to originate from the merchants we track, shown in Figure 6. At its peak, the underground marketplace was responsible for registering 60% of all accounts that would go on to be suspended for spamming. During more typical periods of activity, the merchants we track contribute 10–20% of all spam accounts. We note that the drop-off around April does not indicate a lack of recent activity; rather, as accounts are stockpiled for months at a time, they have yet to be released into the hands of spammers, which would lead to their suspension. The most damaging merchants from our impact analysis operate out of blackhat forums and web storefronts, while Fiverr and Freelancer sellers generate orders of magnitude fewer accounts.⁸

6.2 Estimating Revenue

We estimate the revenue generated by the underground market based on the total accounts sold and the prices charged during their sale. We distinguish accounts that have been sold from those that lay dormant and await sale based on whether an account has sent tweets or formed relationships. For sold accounts, we identify which mer-

⁸The exception to this is a Freelancer merchant *kamalkishover*, but based on their merchant pattern overlapping with 9 other merchants, we believe they are simply reselling accounts.

chant created the account and determine the minimum and maximum price the merchant would have charged for that account based on our historical pricing data.⁹ In the event multiple merchants could have generated the account (due to overlapping registration patterns), we simply take the minimum and maximum price of the set of matching merchants.

We estimate that the total revenue generated by the underground account market through the sale of Twitter credentials is between the range of \$127,000– \$459,000 over the course of a year. We note that many of the merchants we track simultaneously sell accounts for a variety of web services, so this value likely represents only a fraction of their overall revenue. Nevertheless, our estimated income is far less than the revenue generated from actually sending spam [17] or selling fake antivirus [25], where revenue is estimated in the tens of millions. As such, account merchants are merely stepping stones for larger criminal enterprises, which in turn disseminate scams, phishing, and malware throughout Twitter.

7 Disrupting the Underground Market

With Twitter’s cooperation, we disable 95% of all fraudulent accounts registered by the 27 merchants we track, including those previously sold but not yet suspended for spamming. Throughout this process, we simultaneously monitor the underground market to track fallout and recovery. While we do not observe an appreciable increase in pricing or delay in merchant’s delivering new accounts, we find 90% of all purchased accounts immediately after our actioning are suspended on arrival. While we successfully deplete merchant stockpiles containing fraudulent accounts, we find that within two weeks merchants were able to create fresh accounts and resume selling working credentials.

7.1 Suspending Identified Accounts

In order to disrupt the abusive activities of account merchants, we worked with Twitter’s Anti-spam, SpamOps, and Trust and Safety teams to manually validate the accuracy of our classifier and tune parameters to set an acceptable bounds on false positives (legitimate users incorrectly identified as fraudulent accounts). Once tuned, we applied the classifier outlined in Section 5 to every account registered on Twitter going back to March, 2012,

⁹Determining the exact time of sale for an account is not possible due to the potential of miscreants stockpiling their purchases; as such, we calculate revenue for both the minimum and maximum possible price.

filtering out accounts that were already suspended for abusive behavior.

From the set of accounts we identified¹⁰, Twitter iteratively suspended accounts in batches of ten thousand and a hundred thousand before finally suspending all the remaining identified accounts. At each step we monitored the rate of users that requested their accounts be unsuspended as a metric for false positives, where unsuspension requests require a valid CAPTCHA solution. Of the accounts we suspended, only 0.08% requested to be unsuspended. However, 93% of these requests were performed by fraudulent accounts abusing the unsuspend process, as determined by manual analysis performed by Twitter. Filtering these requests out, we estimate the final precision of our classifier to be 99.9942%. The tuned classifier has a recall of 95%, the evaluation of which is identical to the method presented in Section 5. Assuming our purchases are a random sample of the accounts controlled by the underground market, we estimate that 95% of all fraudulent accounts registered by the 27 merchants we track were disabled by our actioning.

7.2 Marketplace Fallout and Recovery

Immediately after Twitter suspended the last of the underground market’s accounts, we placed 16 new orders for accounts from the 10 merchants we suspected of controlling the largest stockpiles. Of 14,067 accounts we purchased, 90% were suspended on arrival due to Twitter’s previous intervention. When we requested working replacements, one merchant responded with:

All of the stock got suspended ... Not just mine .. It happened with all of the sellers .. Don’t know what twitter has done ...

Similarly, immediately after suspension, *buyaccts.com* put up a notice on their website stating “Временно не продаем аккаунты Twitter.com”, translating via Google roughly to “Temporarily not selling Twitter.com accounts”.

While Twitter’s initial intervention was a success, the market has begun to recover. Of 6,879 accounts we purchased two weeks after Twitter’s intervention, only 54% were suspended on arrival. As such, long term disruption of the account marketplace requires both increasing the cost of account registration (as outlined in Section 4) and integrating at-signup time abuse classification into the account registration process (similar to the classifier

¹⁰Due to operational concerns, we cannot specify the exact volume of accounts we detect that were not previously suspended by Twitter’s existing defenses.

outlined in Section 5). We are now working with Twitter to integrate our findings and existing classifier into their abuse detection infrastructure.

8 Summary

We have presented a longitudinal investigation of the underground market tied to fraudulent Twitter credentials, monitoring pricing, availability, and fraud perpetrated by 27 account merchants. These merchants specialize in circumventing automated registration barriers by leveraging thousands of compromised hosts, CAPTCHA solvers, and access to fraudulent Hotmail, Yahoo, and mail.ru credentials. We identified which registration barriers positively influenced the price of accounts and distilled our observations into a set of recommendations for how web services can improve existing barriers to bulk signups. Furthermore, we developed a classifier based on at-registration abuse patterns to successfully detect several million fraudulent accounts generated by the underground market. During active months, the 27 merchants we monitor appeared responsible for registering 10–20% of all accounts later flagged by Twitter as spam. For their efforts, these merchants generated an estimated revenue between \$127,000–\$459,000. With Twitter’s help, we successfully suspended 95% of all accounts registered by the 27 merchants we track, depleting the account stockpiles of numerous criminals. We are now working with Twitter to integrate our findings and existing classifier into their abuse detection infrastructure.

Acknowledgments

This work was supported by the National Science Foundation under grants 1237076 and 1237265, by the Office of Naval Research under MURI grant N000140911081, and by a gift from Microsoft Research. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the sponsors.

References

- [1] Alexa. Alexa top 500 global sites. <http://www.alexa.com/topsites>, 2012.
- [2] D.S. Anderson, C. Fleizach, S. Savage, and G.M. Voelker. Spammer scatter: Characterizing internet scam hosting infrastructure. In *USENIX Security*, 2007.
- [3] F. Benevenuto, G. Magno, T. Rodrigues, and V. Almeida. Detecting Spammers on Twitter. In *Proceedings of the Conference on Email and Anti-Spam (CEAS)*, 2010.
- [4] J. Caballero, C. Grier, C. Kreibich, and V. Paxson. Measuring pay-per-install: The commoditization of malware distribution. In *USENIX Security Symposium*, 2011.
- [5] CBL. Composite Blocking List. <http://cbl.abuseat.org/>, 2012.
- [6] G. Danezis and P. Mittal. Sybilinfer: Detecting sybil nodes using social networks. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2009.
- [7] J. Franklin, V. Paxson, A. Perrig, and S. Savage. An inquiry into the nature and causes of the wealth of Internet miscreants. In *Proceedings of ACM Conference on Computer and Communications Security*, pages 375–388, October 2007.
- [8] H. Gao, J. Hu, C. Wilson, Z. Li, Y. Chen, and B.Y. Zhao. Detecting and characterizing social spam campaigns. In *Proceedings of the Internet Measurement Conference (IMC)*, 2010.
- [9] C. Grier, L. Ballard, J. Caballero, N. Chachra, C.J. Dietrich, K. Levchenko, P. Mavrommatis, D. McCoy, A. Nappa, A. Pitsillidis, et al. Manufacturing compromise: The emergence of exploit-as-a-service. 2012.
- [10] C. Grier, K. Thomas, V. Paxson, and M. Zhang. @spam: the underground on 140 characters or less. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2010.
- [11] T. Holz, C. Gorecki, F. Freiling, and K. Rieck. Detection and mitigation of fast-flux service networks. In *Proceedings of the 15th Annual Network and Distributed System Security Symposium (NDSS)*, 2008.
- [12] C.Y. Hong, F. Yu, and Y. Xie. Populated ip addresses—classification and applications. 2012.
- [13] Heather Kelley. 83 million facebook accounts are fakes and dupes. <http://www.cnn.com/2012/08/02/tech/social-media/facebook-fake-accounts/index.html>, 2012.
- [14] Brian Krebs. Spam volumes: Past & present, global & local. <http://krebsonsecurity.com/2013/01/spam-volumes-past-present-global-local/>, 2012.
- [15] S. Lee and J. Kim. Warningbird: Detecting Suspicious URLs in Twitter Stream. In *Symposium on Network and Distributed System Security (NDSS)*, 2012.
- [16] K. Levchenko, A. Pitsillidis, N. Chachra, B. Enright, M. Felegyhazi, C. Grier, T. Halvorson, C. Kanich, C. Kreibich, H. Liu, D. McCoy, N. Weaver, V. Paxson, G.M. Voelker, and S. Savage. Click Trajectories: End-to-End Analysis of the Spam Value Chain. In *Proceedings of the 32nd IEEE Symposium on Security and Privacy*, 2011.
- [17] D. McCoy, A. Pitsillidis, G. Jordan, N. Weaver, C. Kreibich, B. Krebs, G.M. Voelker, S. Savage, and K. Levchenko. PharmaLeaks: Understanding the business of online pharmaceutical affiliate programs. In *Proceedings of the 21st USENIX conference on Security symposium*. USENIX Association, 2012.
- [18] A. Metwally and M. Paduano. Estimating the number of users behind ip addresses for combating abusive traffic. In *Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 2011.
- [19] M. Motoyama, K. Levchenko, C. Kanich, D. McCoy, G.M. Voelker, and S. Savage. Re: Captchas—understanding captcha-solving services in an economic context. In *USENIX Security Symposium*, volume 10, 2010.

- [20] M. Motoyama, D. McCoy, K. Levchenko, S. Savage, and G.M. Voelker. An analysis of underground forums. In *Proceedings of the Internet Measurement Conference (IMC)*. ACM, 2011.
- [21] M. Motoyama, D. McCoy, K. Levchenko, S. Savage, and G.M. Voelker. Dirty jobs: The role of freelance labor in web service abuse. In *Proceedings of the 20th USENIX Security Symposium*, 2011.
- [22] A. Pitsillidis, K. Levchenko, C. Kreibich, C. Kanich, G.M. Voelker, V. Paxson, N. Weaver, and S. Savage. Botnet Judo: Fighting spam with itself. 2010.
- [23] P. Prasse, C. Sawade, N. Landwehr, and T. Scheffer. Learning to identify regular expressions that describe email campaigns. 2012.
- [24] W.E. Ricker. *Computation and interpretation of biological statistics of fish populations*, volume 191. Department of the Environment, Fisheries and Marine Service Ottawa, 1975.
- [25] B. Stone-Gross, R. Abman, R. Kemmerer, C. Kruegel, D. Steigerwald, and G. Vigna. The Underground Economy of Fake Antivirus Software. In *Proceedings of the Workshop on Economics of Information Security (WEIS)*, 2011.
- [26] G. Stringhini, C. Kruegel, and G. Vigna. Detecting Spammers on Social Networks. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, 2010.
- [27] K. Thomas, C. Grier, J. Ma, V. Paxson, and D. Song. Design and Evaluation of a Real-time URL Spam Filtering Service. In *Proceedings of the 32nd IEEE Symposium on Security and Privacy*, 2011.
- [28] K. Thomas, C. Grier, and V. Paxson. Adapting social spam infrastructure for political censorship. In *Proceedings of the 5th USENIX conference on Large-Scale Exploits and Emergent Threats*. USENIX Association, 2012.
- [29] K. Thomas, C. Grier, V. Paxson, and D. Song. Suspended Accounts In Retrospect: An Analysis of Twitter Spam. In *Proceedings of the Internet Measurement Conference*, November 2011.
- [30] Twitter. The Twitter Rules. <http://support.twitter.com/entries/18311-the-twitter-rules>, 2010.
- [31] G. Wang, C. Wilson, X. Zhao, Y. Zhu, M. Mohanlal, H. Zheng, and B.Y. Zhao. Serf and Turf: Crowdurfing for Fun and Profit. In *Proceedings of the International World Wide Web Conference*, 2011.
- [32] Y. Xie, F. Yu, K. Achan, R. Panigrahy, G. Hulten, and I. Osipkov. Spamming botnets: Signatures and characteristics. *Proceedings of ACM SIGCOMM*, 2008.
- [33] 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 Conference on World Wide Web*, 2012.
- [34] H. Yu, M. Kaminsky, P.B. Gibbons, and A. Flaxman. Sybilguard: defending against sybil attacks via social networks. *ACM SIGCOMM Computer Communication Review*, 2006.
- [35] Y. Zhao, Y. Xie, F. Yu, Q. Ke, Y. Yu, Y. Chen, and E. Gillum. Botgraph: Large scale spamming botnet detection. 2009.

A Legal and Ethical Guidelines

To minimize the risk posed to Twitter or its users by our investigation of the account market, we follow a set of policies set down by our institutions and Twitter, reproduced here to serve as a note of caution to other researchers conducting similar research.

Twitter & Users Some of the account merchants we deal with work in an on-demand fashion, where purchases we place directly result in abusive registrations on Twitter (e.g. harm) in violation of the site's Terms of Services. Even purchases from existing stockpiles might be misconstrued as galvanizing further abuse of Twitter. As such, we directly contacted Twitter to receive permission to conduct our study. In the process, we determined that any interactions with the underground market should not result in harm to Twitter's user base. In particular, accounts we purchased should *never* be used to tweet or form relationships while under our control. Furthermore, we take no special action to guarantee our accounts are not *suspended* (e.g disabled) by Twitter; our goal is to observe the natural registration process, not to interact with or impede Twitter's service in any way.

Account Merchants We do not interact with merchants anymore than necessary to perform transactions. To this end, we only purchase from merchants that advertise their goods publicly and never contact merchants outside the web sites or forums they provide to conduct a sale (or to request replacement accounts in the event of a bad batch). Our goal is not to study the merchants themselves or to collect personal information on them; only to analyze the algorithms they use to generate accounts.

Sensitive User Data Personal data logged by Twitter is subject to a multitude of controls, while user names and passwords sold by merchants also carry controls to prevent fraud, abuse, and unauthorized access. First, we *never* log into accounts; instead, we rely on Twitter to verify the authenticity of credentials we purchase. Furthermore, all personal data such as IP addresses or activities tied to an account are never accessed outside of Twitter's infrastructure, requiring researchers involved in this study to work on site at Twitter and to follow all relevant Twitter security practices. This also serves to remove any risk in the event an account is *compromised* rather than registered by an account merchant, as no personal data ever leaves Twitter. To our knowledge, we never obtained credentials for compromised accounts.

Impression Fraud in Online Advertising via Pay-Per-View Networks

Kevin Springborn
Broadcast Interactive Media
kspringborn@bimlocal.com

Paul Barford
Broadcast Interactive Media
University of Wisconsin-Madison
pbarford@bimlocal.com

ABSTRACT

Advertising is one of the primary means for revenue generation for millions of websites and mobile apps. While the majority of online advertising revenues are based on pay-per-click, alternative forms such as impression-based display and video advertising have been growing rapidly over the past several years. In this paper, we investigate the problem of invalid traffic generation that aims to inflate advertising impressions on websites. Our study begins with an analysis of purchased traffic for a set of honeypot websites. Data collected from these sites provides a window into the basic mechanisms used for impression fraud and in particular enables us to identify *pay-per-view (PPV) networks*. PPV networks are comprised of legitimate websites that use JavaScript provided by PPV network service providers to render unwanted web pages "underneath" requested content on a real user's browser so that additional advertising impressions are registered. We describe the characteristics of the PPV network ecosystem and the typical methods for delivering fraudulent impressions. We also provide a case study of scope of PPV networks in the Internet. Our results show that these networks deliver hundreds of millions of fraudulent impressions per day, resulting in hundreds of millions of lost advertising dollars annually. Characteristics unique to traffic delivered via PPV networks are also discussed. We conclude with recommendations for countermeasures that can reduce the scope and impact of PPV networks.

1. INTRODUCTION

Advertising is one of the primary methods for generating revenues from websites and mobile apps. A recent report from the Internet Advertising Bureau (IAB) places ad revenues in the US for the first half of 2012 at \$17B, which represents a 14% increase over the previous year [15]. While the majority of that revenue is search-based, ad words advertising, display and video advertising have been growing. Indeed, a recent report places display and video advertising in the US at \$12.7B for FY2012, growing at 17% annually [27]. At a high level the basic notion of selling space on web pages and apps for advertising is simple. However, the mechanisms and infrastructure that are required for online advertising are highly diverse and complex.

The online ad ecosystem can roughly be divided into three groups: *advertisers*, *publishers* and *intermediaries*. Advertisers pay publishers to place a specified volume of creative content with embedded links (*i.e.*, text, display or video ads) on websites and apps. Intermediaries (*e.g.*, ad servers and ad exchanges) are often used to facilitate connections between publishers and advertisers. Intermediaries typically place a surcharge on the fees paid by advertisers to publishers for ad placements and/or ad clicks. What is immediately obvious from this simple description is that publisher and intermediary platform revenues are directly tied to the number of daily visits to a website or app. Thus, there are strong incentives for publishers and intermediaries to use any means available to drive user traffic to publisher sites.

There are certainly legitimate methods for traffic generation for publisher sites. The most widely used are the text-based ad words that appear in search results *e.g.*, from Google or Bing. However, it can be quite difficult and expensive to drive large traffic volumes to target sites using ad words alone.¹ Thus, other methods for traffic generation have emerged, many of which are deemed as fraudulent by advertisers and intermediaries. Google defines invalid (fraudulent) traffic as follows:²

Invalid traffic includes both clicks and impressions that Google suspects to not be the result of genuine user interest [21].

Standard methods for generating invalid traffic includes (*i*) using employees at publisher companies to view sites and click on ads, (*ii*) hiring 3rd parties to view sites and click on ads, (*iii*) click/view pyramid schemes and (*iv*) using software and/or botnets to automate views/clicks [21]. The challenges for advertisers and intermediaries focused on offering trustworthy platforms are to understand these and potentially other threats so that effective countermeasures can be deployed.

In this paper, we investigate a relatively new threat for display and video advertising called Pay-Per-View (PPV) net-

¹This has led to the emergence of a large number of Search Engine Optimization companies in recent years.

²While Google is not the only company in this domain, we refer to them as an authoritative source of information due to their size and experience in online advertising.

works. The basic idea for PPV networks is to pay legitimate publishers to run specialized JavaScript when users access their sites that will display other publishers websites in a camouflaged fashion. This will result in impressions and potentially even clicks that are registered on the camouflaged pages without "genuine user interest" *i.e.*, invalid traffic generation. Legitimate publishers view this as another way to monetize their sites without impact to their users. PPV networks sell their traffic generation capability by touting real and unique users, geolocation and context specificity among other things. The fact that pages are appearing on real users' systems makes detecting and preventing PPV traffic generation challenging.

To study PPV networks, we employ a small set of honeypot websites that we use as the target for traffic generation. These sites were constructed to include what appears to be legitimate content and advertising. We then use search to identify a wide variety of traffic generation offerings on the Internet. We purchased impressions for our honeypot sites in various quantities from a selection of different traffic generation services over the course of a 3.5 month period. By engaging with traffic generation services directly, we were able to uncover the basic mechanisms of PPV networks and initiate additional measurements to characterize their deployments.

The characteristics of the traffic purchased for our honeypot sites is dictated at a high level by the service offerings, which enable volume, time frame and geographic location, etc. of users to be specified. Our results show that impressions are typically spread in a somewhat bursty fashion over the specified time frame and that user characteristics are well matched with specifications. By considering the referer field of the incoming traffic, we were able to identify the fact that our honeypot sites were being loaded into a frame (along with as many as ten other sites) for display on remote systems. By considering names of a small selection of traffic generation services, we use a recent, publicly-available, Internet-wide web crawl to identify the scope of PPV networks. We find tags from these services are, in fact, widely deployed – on tens of thousands of sites. By appealing to MuStat [29], we conservatively estimate the number of invalid impressions that are generated from this small set of PPV networks to be on the order of 500 million per day. Assuming a modest quality level for sites that are part of PPV networks, we estimate the annual cost to advertisers for this invalid traffic to be on the order of \$180 million annually.

Finally, we offer three different methods to defend against PPV networks. First, observing viewport dimensions of ad requests can determine if the end user can possibly view the advertisement. In an effort to increase traffic, PPV networks commonly display destinations in zero sized frames. Second, blacklists of websites that participate in PPV networks can potentially be used. The idea is to block advertising on websites that commonly receive PPV traffic until the publisher discontinues purchasing PPV traffic. Such blacklists

can be compiled through programmatic enumeration of PPV destinations. Finally, referer fields can be queried at the time of advertisement load in order to identify traffic originating from known PPV domains.

The remainder of this paper is organized as follows. In Section 2 we provide a description of the online advertising ecosystem and an overview of invalid traffic generation threats. In Section 3, we describe the details of our honeypot websites and our traffic purchases for these sites. In Section 4, we describe the details of the evaluations that we conduct on our data including analyses of additional data sets and measurements that enable us to project some of the broader characteristics of PPV networks. We provide recommendations for counter measures that can be employed to reduce the impact of PPV networks in Section 5. We discuss prior studies that inform our work in Section 6. We summarize, conclude and discuss future work in Section 7.

2. ONLINE ADVERTISING ECOSYSTEM

In this section we provide an overview of the online advertising ecosystem including both the business framework and technical framework for delivering advertisements to publisher websites and apps. Some prior studies have provided similar overviews including [16, 34, 41]. We also provide an overview of invalid traffic generation threats and the challenges they pose in the ecosystem.

2.1 Business Framework

As mentioned in Section 1, there are three main participant groups in ad networks: advertisers, intermediaries and publishers. As shown in Figure 1 there are two other important groups: brands and users. Brands pay advertisers to help them sell their products and services. Internet-based campaigns are attractive to brands and advertisers since consumers/users spend a growing proportion of their time online. An important appeal of online advertising (especially for consumer goods) is that it offers the opportunity to tie ad campaigns and associated costs directly to sales *e.g.*, by tracking clicks from online ads to purchases on a brand's ecommerce site.

Advertisers are companies that create and manage advertising campaigns for brands. Advertisers pay publishers to make ad placements on websites and apps using one of several different models. One is the widely used Pay-Per-Click (PPC) model, where an advertiser only pays a publisher for an ad when a user clicks on it. PPC campaigns are typically associated with ad words (short, text-based ads) campaigns. An alternative payment method that is common in display and video advertising is Cost Per Mille/Thousand (CPM), where advertisers pay publishers whenever users view an ad (CPM prices are given per thousand impressions). The CPM-based payment model is the primary focus for this paper. The goal for advertisers is to place ads on sites that they believe attract a brand's target demographic in a cost-effective fashion. Thus, their challenge is in identifying these

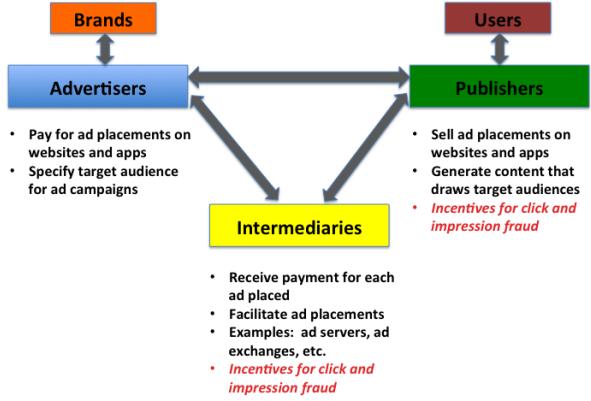


Figure 1: Key participants in the online advertising ecosystem. Payments flow from brands to advertisers to intermediaries and publishers.

sites and facilitating ad placement.

In addition to working with publishers directly, advertisers often work with intermediaries in order to actually place ads on websites and apps. The two main reasons for this are the complexity of Internet advertising's technical landscape (see below) and the enormous and growing diversity of websites and apps. Among other things, intermediaries offer "one-stop shopping" for advertisers, and competitive CPM rates to publishers who may not be able to fill all of their placements via direct campaigns.

The scope of intermediaries is quite broad. The most common offerings include targeting services, ad servers and ad exchanges to facilitate placements. One of the most widely used intermediaries in the display advertising space is Google AdExchange (AdX) [20,30]. The revenue model that is most commonly used by intermediaries is to take a small CPM payment for each ad that they participate in serving and then to pass the remainder of the CPM paid by the advertiser to the publisher.

Internet publishers are companies that create content that is of interest to users. Publishers display ads on their pages using standard sized creatives that typically appear in an iframe. A publisher's goal is to maximize their revenue yield by attracting (*i*) premium advertisers that pay high CPM's and (*ii*) a high volume of users, some whom will click through on ads. It is important to note that while ad words-based advertising (*e.g.*, through AdSense) is widely available, display and video ads are typically only available to sites that have somewhat higher volumes of users.

2.2 Technical Framework

Displaying an advertisement on a publisher's page includes potentially a large number of data exchanges between participants in the advertising ecosystem. A simple example is depicted in Figure 2. The process begins with the placement of an *ad tag* in a section of a publisher page. Ad tags (often supplied by intermediaries that manage ad servers) are simple HREF strings that typically reference JavaScript code

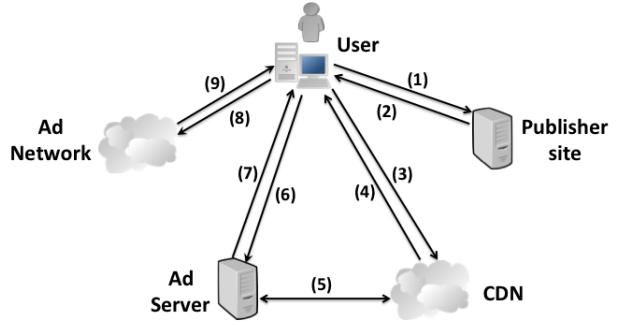


Figure 2: Typical data exchanges required to render an ad in a user's browser. (1) User request to publisher page. (2) Base page delivered. (3) Ad tag request to CDN. (4) JavaScript delivered. (5) Update to JavaScript in CDN if necessary. (6) Request to ad server. (7) Redirected request delivered. (8) Request to exchange or 3rd party ad server. (9) Ad creative delivered.

hosted in a CDN infrastructure.

The JavaScript typically gathers context keywords and other information from the publisher page or user browser and then sends an ad request to the target ad server infrastructure. Ad servers process the ad request and either respond with an ad directly (*e.g.*, from a direct advertiser campaign) or send a redirect to a third party such as an ad exchange. The redirect is forwarded by browser to the target server or exchange, which will respond with an ad that is rendered in the browser. The redirect usually includes sufficient information for ad targeting and billing. This entire process must take place quickly (typically on the order of tens of milliseconds) in order to ensure a good user experience. When the ad is delivered, an impression is registered for the ad serving entity. Click tracking is typically managed by directing clicks to the ad server, which then redirects to the advertiser.

2.3 Invalid Traffic Generation Threats

Impression-based advertising has a number of potential threats. The focus of this paper is on traffic generation that causes invalid impression and thereby inflates publisher and (some) intermediary revenues. Specifically, we focus on invalid traffic generation via PPV networks, which we describe in detail in Section 4.

Valid methods for traffic generation include search and ad words-based advertising. However, web search reveals that there is a wide variety of other traffic generation offerings available. Many offer a specified volume of traffic at a target site over a specified time period. Many also include guarantees of specific features in the traffic such as geographic locations of host systems. Most do not describe their method-

ology in detail if at all. One of the important objectives of traffic generation is that it appear to come from real users. Appealing to the definition of invalid traffic given in Section 1 above, there are many ways in which such traffic might be generated.

Common methods for invalid traffic generation have been borrowed directly from click generation services that have been offered for some time. Examples include hiring people to view pages, bots of various types, and using expired domains to divert users to 3rd-party pages.

PPV networks are sites that load 3rd-party pages in an obfuscated fashion when accessed by users. Publishers become part of a PPV network simply by placing a tag on their site that looks very much like a standard ad tag. We define a "network" as a series of sites that run tags from the same PPV service. Participating publishers are paid on a CPM basis for something that appears to be low or no impact on their site.

Since the third party pages that are rendered via PPV networks are clearly not the interest of the users, all of the resulting impressions are invalid. Beyond lacking the intent necessary to qualify as valid traffic, we show that PPV network traffic has characteristics unlike organic traffic. For example, natural traffic displays a diurnal traffic pattern, while the PPV traffic we observed often showed highly artificial delivery patterns.

3. DATA COLLECTION ON HONEYBOT WEBSITES

To begin our investigation of traffic generation and impression fraud we established a set of honeypot websites. We then purchased traffic from a number of different services and captured a diverse set of data from the resulting hits on our sites. In this section we provide details on our honeypot websites and traffic purchases. The results of these activities are described in detail in Section 4.

3.1 Honeypot Websites

We created three websites as the starting point for our investigation of traffic generation service providers. The sites differed only in styling, formatting, and deployment. The content on each site was identical. The reason for creating three different sites was to enable us to conduct A-B comparisons between different traffic generation services.

The design objective for our honeypots was to create sites that looked relatively "legitimate". To that end, they have a standard layout, content changes regularly and the deployment is standard. A second objective was that the sites were instrumented to gather as much data as possible on arriving traffic.

Each site consisted of a base landing page and four sub-pages. Three of the pages displayed RSS content from the news feeds of topwirenews.com or espn.com. One page listed links to popular news sites. The final page was a non-functional search result. Every page contained four adver-

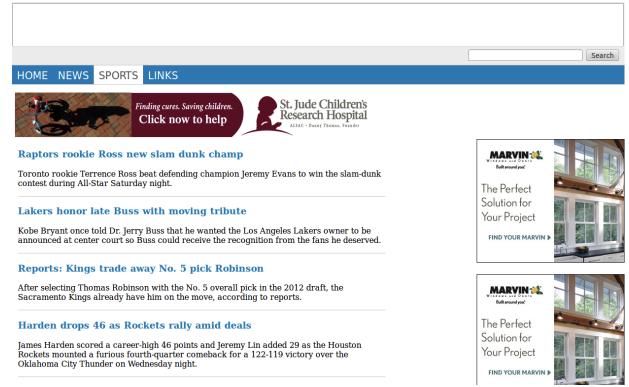


Figure 3: Screenshot of one of the honeypot websites that was a target for traffic generation purchases.

tisement placements, identical to standard CPM placements except they contained dummy creatives instead of displaying paying advertisers' placements. All of the ads have embedded links to dummy landing pages that we also monitor.

Domain names were registered for each site with GoDaddy using their anonymous registration option. We attempted to give the sites names that sounded interesting and connoted the news-related content of the sites. The sites were created using dotCMS inside Amazon EC2. Amazon's CloudFront CDN was enabled for the sites in order to handle larger bursts of traffic. We used a "noindex,nofollow" meta tag and a robots.txt file to attempt to prevent inclusion in search engine results.

Instrumentation was facilitated in several ways. Google Analytics tags were deployed on all pages for general monitoring. Logs from the serving infrastructure were used to understand the details of individual connections. A series of JavaScript blocks collected information about the site visitors. The instrumentation reported viewer characteristics (See Table 1) using 1x1 pixels. Each advertisement on the sites was instrumented with code that reported the three key events in the life cycle of every ad: (1) JavaScript load (2) JavaScript execution and (3) successful delivery. Finally, the pages contained JavaScript that tracked user interaction on the site. Similar to [41] the interaction metrics reported mouse movements and clicks. The mouse position was collected every time the cursor moved at least 20 pixels.

3.2 Purchased Traffic

We identified and reviewed 34 traffic generation service providers for this study. These service providers were identified using web search. We manually reviewed each service provider's site to catalog available purchasing options. Details of the sites and options are given in Table 3. We make no claims on the completeness of this list of traffic generation service providers. However, given the commonality of their offerings, we believe that they are a representative cross section.

We also investigated the service provider websites themselves to gain some insights on their legitimacy. Their do-

main names were checked with McAfee SiteAdvisor [6]. The DNS record was inspected using Network Solutions' Whois tool [8]. Finally, a tool available from SameID.net [9] was used to search for sites sharing the same IP address or Google Analytics tag.

Table 1: Visitor information collected from honeypot websites.

Timestamp	Client IP
URL	User Agent
User UID	Page Load UID
Viewport Dimensions	Referer

From the set of 34 traffic generation services, we selected 5 from which we made purchases. Services were selected to get a diversity of delivery rates and price points. The characteristics of our purchased traffic indicated the selected services were independent networks. The purchased traffic was directed to the honeypot sites between November 11th, 2012 and February 18th, 2013, resulting in over 69K delivered impressions. We used target URL's including Google Analytics campaign parameters [5] to help to differentiate overlapping purchases.

Our purchasing strategy was oriented around diversity and not volume. Details of the purchased traffic can be found in Table 2. With the exception of BuildTraffic all traffic purchased was designated as only traffic from United States and labeled as news and information. The intended delivery rate of purchased traffic varied between 333 visitors per day to 25,000 visitors per day. We intend to investigate further diversity and higher volume purchases in future work.

Table 2: Traffic purchases made for this study.

Vendor	Amount	Runtime	Price
MaxVisits	10,000	5 days	\$11.99
BuildTraffic	20,000	60 days	\$55.00
AeTraffic	10,000	7 days	\$39.95
BuyBulkVisitor	20,000	5 days	\$53.00
TrafficMasters	50,000	2 days	\$70.00

3.3 Pay-Per-View Publisher Signup

In addition to traffic generation itself, PPV service providers also offer publishers the opportunity to participate as a traffic source in their network (this was our initial indication of PPV networks). To further investigate the mechanisms of traffic generation, we enrolled as a website owner willing to display content with a PPV service provider called InfinityAds. The signup was completed using InfinityAds' fully automated publisher signup system on their website. Upon signup we were given a block of JavaScript to load on our site. In return for running this tag, the website owner is assured of a relatively attractive CPM (quoted and qualified

at \$1.80) and that "...pop under ads will not block any of your site content and do not lead to actions where users might be led to leave your site." [23]. In this case, pop-under windows are the method that InfinityAds uses to generate traffic. We describe these in more detail below.

4. PAY-PER-VIEW NETWORK CHARACTERISTICS

In this section we report the results of our analysis of purchased traffic at our honeypot sites. This analysis reveals the mechanisms used to drive traffic to target sites and opens the door to a broader analysis of PPV networks, which is also reported below.

4.1 Traffic Generation Offerings

We reviewed the details of the 34 traffic generation/ecommerce sites that we identified via web search using strings like "website traffic", "buying web traffic", "web trafficking", etc. Features such as traffic characteristics, pricing, timing, reseller information, and DNS entries were noted for each site. Details are listed in Table 3.

4.1.1 Pricing

There is no uniform pricing for traffic providers. The pricing given in Table 3 was normalized to the cost of delivering 25,000 visitors from the United States for comparison. Of the 34 traffic generation services that we investigated, five of them did not allow purchasing traffic originating exclusively from the United States. One site was deemed fraudulent because it did not have a space to enter a traffic destination prior to checkout completion. The remaining 28 sites charged between \$29.99 and \$200 to purchase 25k visitors.

4.1.2 Overlap/Reselling

There were significant similarities between many of the traffic purchase sites. Many of the providers made multiple copies of their site in order to target different publisher segments or to simply use another attractive domain name. All of the provider domains were assessed using the sameid.net domain investigation tool [9]. Seven of the providers appeared to be repackaging another site (handytraffic, cmkmarketing, visitorboost, revisitors, buybulkvisitor, highurlstats, xrealvisitors). Four of the repackaged sites shared a Google Analytics account with another traffic provider site (handytraffic, cmkmarketing, visitorboost, revisitors). Three of the repackaged sites shared an IP address with another traffic purchase site (buybulkvisitor, highurlstats, xrealvisitors). Shared website hosting could cause IP overlap, but it is unlikely that 3 sites in our 34 site sample are randomly hosted on the same IP. Furthermore an implementation error caused *highurlstats.com* to load *buybulkvisitor.com*, making it plausible that these sites are related.

Four of the PPV sellers investigated offered the ability to become a traffic reseller (hitpro, ineedhits, toptrafficwholesaler, traffic-masters). A reseller sells traffic without having

Table 3: Traffic provider details.

Site	Price ²	Geotargeting	Category	Pacing	Adult	Allow Pop-up/Sound
aetraffic.com	\$75	Yes	Yes	Option	Option	Yes ²
allseostar.com	NA	No	No	No	Opion ²	No
bringvisitor.com	NA	No	No	No	?	Yes ²
buildtraffic.com	\$119	Yes	Yes	30 days	?	No
buybulkvisitor.com	\$53	Yes	Yes	Option	?	No
buvhitscheap.com	\$110	Yes	No	No	?	Yes
cheapadvertising.biz	NA	No	No	No	Option	?
cmkmarketing.com	\$82	Yes	Yes	No	?	No
cybertrafficstore.com	\$70	Yes	Yes	30 days	Option	?
easytraffic.biz	\$100	Yes	Yes	60 days	?	No
fulltraffic.net	\$220	Yes	No	No	?	?
getwebsitetraffic.org	\$75	Yes	Yes	Option	Option	Yes ²
growstats.com	\$84	Yes	Yes	Option	?	Yes ²
handytraffic.com	\$99	Yes	Yes	Option ²	?	Yes
highurlstats.com	\$200	Yes	Yes	30 days	?	?
hitpro.us	\$60	Yes	Yes	30 days	?	No
ineedhits.com	\$120	Yes	Yes	30 days	?	No
masvisitas.net			No information, nowhere to enter website URL			
maxvisits.com	\$30	Yes	Yes	Option	?	Yes
meantraffic.com	\$30	Yes	Yes	No	Option ²	?
perfecttraffic.com	\$43	Yes	Yes	Option	?	?
plusvisites.com	\$30	Yes	Yes	Option	?	?
purchasewebsitetraffic.net	\$99	No	No	No	?	No
realtrafficsource.com	\$55	Yes	Yes	No	?	?
revisitors.com	\$119	Yes	Yes	Option ²	Option ²	Yes ²
source4traffic.com	\$88	Yes	Yes	30 days	?	No
thewebtrafficdominator.com	\$32	Yes	Yes	No	Option ²	?
toptrafficwholesaler.com	\$111	Yes	Yes ²	30 days	Option ²	No
traffic-masters.com	\$35	Yes	Yes	Option	Option ²	No
trafficchamp.com	\$89	Yes	Yes	30 days	No	No
trafficelf.com	\$55	Yes	Yes	Option	Option ²	Yes
trafixtech.com	\$35	Yes	Yes	Option	Option ²	No
visitorboost.com	\$116	Yes	Yes	30 days	?	No
xrealvisitors.com	NA	No	No	No	?	?

¹ Cost to purchase 25,000 United States visitors (normalized where needed)

² Extra cost

to manage traffic delivery infrastructure or payment processing. The reseller acts only as an intermediary forwarding orders along to the true traffic provider. As per the descriptions, the reseller is charged a fixed rate for the traffic and can resell the traffic at the price of their choosing. Two of the reseller packages offered prepackaged websites where the reseller only needs to supply their branding and marketing.

4.1.3 Provider Site Analysis

Given the potentially fraudulent nature of traffic generation, we were interested in a general measure of the trustworthiness of providers sites. McAfee’s SiteAdvisor [6] rated most of the provider websites as safe. Specifically, out of

the 34 sites investigated 22 were labeled as Safe, 11 had not yet been reviewed by SiteAdvisor, and 1 was labeled as suspicious.

4.1.4 DNS Registration

A Whois lookup was performed on each of the traffic providers websites to gain insights on deployments. 14 out of the 34 sites listed a DNS anonymization service as their primary contact. Four of the sites were registered or renewed in the previous 12 months. Expiration and creation dates give the period the domain registration. On average the sites were registered for 5.71 years. The longest registration was for 16 years. Six sites are registered for only 1 year.

Looking at the contract information of the sites not us-

ing anonymization gave the following breakdown of country residency: 10 United States, 2 Australia, 2 Canada, 2 Spain, 1 France, 1 Italy, 1 Singapore, 1 China.

4.1.5 Features

Providers offer a variety of options for purchased traffic. Many provide assurances that only "real" traffic will be delivered and no "black hat techniques" are used. Every site promises unique views, such that the same user will not be directed to the site multiple times in 24 hours. Six sites were more precise, specifying that a user's IP address will only be directed to the destination once in a 24-hour period. Typical traffic volumes range between 10K and 1M visitors per campaign. Direct email was required for campaigns larger than 1M visitors. See Table 4 for other options offered by the traffic providers that we evaluated.

Table 4: Traffic provider features.

Adsense Safe	Safe to use with Google AdSense
Adult Traffic	Deliver users interested in porn
Alexa Boost	Traffic to increase Alexa ranking
Allow Pop-ups/Sound	No restrictions on destination
Campaign Pacing	Select length of campaign
Geo-targeting	Deliver users from a region
Clicks	Deliver clicks on target website
Mobile Traffic	Deliver users of mobile devices
Traffic Classes	Deliver users with specific interest

4.2 Purchased Traffic Characteristics

One of our purchases did not deliver any appreciable volume of traffic. The reason for the failure of traffic delivery is not clear. The provider may have decided not to deliver due to the instrumentation of the destination site. The provider still collected payment for the traffic which was not delivered. See Tables 5 and 6 for a summary of our measurements. Of the target of 110,000 visits that we purchased, we received 69,567. At the time of writing AeTraffic was still delivering visitors beyond the campaign end. The BuildTraffic purchase stopped delivering visitors abruptly at the end of January, 28 days into the 60-day campaign.

We analyzed traffic delivered to our honeypot websites for a variety of characteristics. Before processing, the data was filtered to remove any events originating from our honeypot server's IP address. Also any user agent containing case-insensitive 'bot' was excluded. This was done to remove the effects of web crawler traffic from our results. All of the traffic observed appeared to originate from our purchases. We did not see any indications of natural traffic.

4.2.1 Blacklist Comparison

The IP addresses of the purchased traffic showed some overlap with public IP blacklists. Every morning at 7 GMT IP blocklists were pulled from DShield.org [3] and UceProtect [10] as points of comparison. The count of blacklisted IP

addresses from these sources averaged 303,968 (or 0.007% of the entire IP space) for January 2013. On average, source IP addresses of the purchased data matched the blacklists 0.97% of the time. This is perhaps more than would be expected by chance, but too low to draw a strong conclusion about overlap between the set of sources from traffic generation services and malicious sources.

4.2.2 Interaction

Each of our honeypot pages tracked four JavaScript events: onmousemove, onmousedown, onblur, onfocus. There was an extremely small number of activity events (190) reported for all purchased traffic. There are a few explanations for such low interaction: (i) it may be an accurate reflection of reality, (ii) the site was 0 sized and the user could not interact with it (see 4.2.7) or (iii) it could be the result of JavaScript events not firing as expected. Unfortunately we cannot rule out JavaScript failure. We cannot draw strong conclusions from the lack of interaction events other than the fact that we did not pay for anything other than impressions.

4.2.3 Temporal Distribution

The pacing of visitor delivery varied greatly depending on traffic service provider. As is described below service providers traffic millions if not billions of visitors a day, but individual purchases can require delivery of less than 100 visitors a day to a destination. Furthermore, the network throughput is not guaranteed. So the deliveries need to be slightly front-loaded to ensure full delivery in the case of lower than expected throughput. The problem of pacing manifested itself in both the time of arrivals within a day and the arrival distribution over the entire campaign.

The daily arrival patterns of visitors showed some unusual artifacts. AeTraffic delivered consistently though the entire day as can be seen in Figure 4. It is well known that typical user traffic follows a diurnal cycle, reaching the high peak during the day and low peak overnight when users are sleeping. A more obvious example of artificial delivery is BuildTraffic, which delivered only during the first 10 minutes of the hour, as can be seen in Figure 5.

The arrival of users throughout the campaign was quite bursty in some cases. With periods of high delivery followed by periods of low delivery. MaxVisists delivered traffic primarily in the first half of every day as can be seen in Figure 6. Meanwhile, TrafficMasters delivery primarily consisted of two large spikes with little delivery between, as can be seen in Figure 7.

4.2.4 Incomplete Loads

Every page on our honeypot sites contained four JavaScript blocks which loaded advertising creatives. Each creative was independently instrumented to report when it had been loaded. Four blocks of JavaScript need to complete in order to successfully load all of the ads on the pages. Using this information, we can calculate the percentage of page loads

Table 5: Purchased traffic delivery.

Vendor	Expected Visitors	Delivered Visitors	Expected Duration	Actual Duration	% Loading all 4 Ads
AeTraffic	10,000	17,205	7 days	8 days ¹	16.40
BuildTraffic	20,000	1,086	60 days	29 days	60.75
BuyBulkVisitor	20,000	1	5 days	1 day	Unknown ²
MaxVisits	10,000	9,635	5 days	5 days	12.80
TrafficMasters	50,000	41,640	2 days	3 days	58.34

¹ Still sending traffic at the time of submission

² User failed to load JavaScript

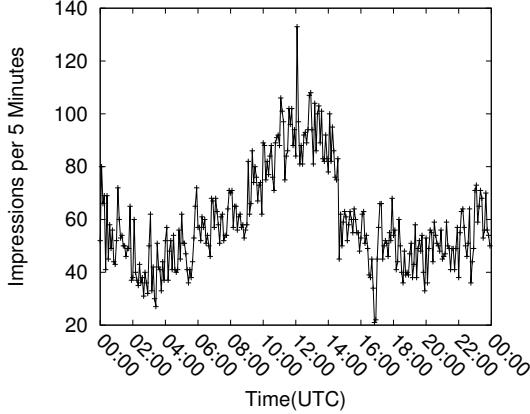


Figure 4: Traffic distribution from AeTraffic.

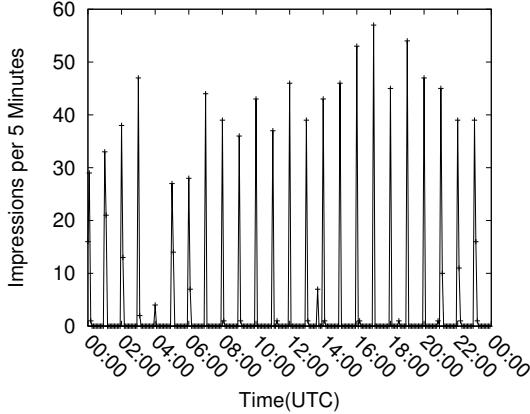


Figure 5: Traffic distribution from BuildTraffic.

that completed for all four ads.

Traffic from BuildTraffic and TrafficMasters resulted in ads completely loading approximately 60% of the time. AeTraffic and MaxVisits only loaded approximately 15% of the time. Reasons for failure to load all the ads include: JavaScript blockers, JavaScript errors, JavaScript execution timeout, and navigation away from the page.

4.2.5 IP Address Distribution

IP addresses from an entire traffic generation campaign were checked for duplicates to get an idea of the distribution of traffic sources. According to the advertised 24-hour-unique policy an IP address could be used once per day. For small purchases our data showed very little over-

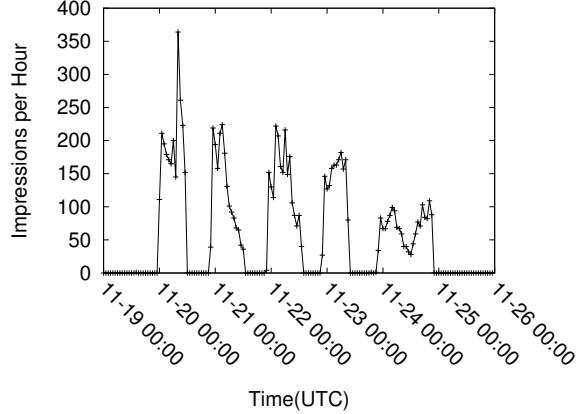


Figure 6: Traffic distribution from MaxVisits.

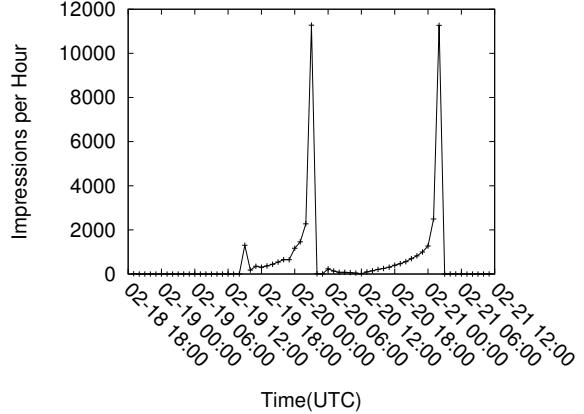


Figure 7: Traffic Distribution from TrafficMasters.

lap of IP address for the campaigns: AeTraffic reused 0.75% of IP addresses, BuildTraffic reused 0.64% and MaxVisits reused 11.25%. The larger purchase from TrafficMasters showed significantly more IP address overlap with 65% of IPs reused. The majority of the IPs geolocated inside of the US, with the exception of the BuildTraffic IPs.

4.2.6 User Agents

The number of unique user agents across the purchases shows the traffic came from a diverse set of browsers. An alternative explanation could be that artificial traffic generators utilized a large set of User Agent strings. However, combined with the diverse set of IP addresses, it appears the traffic could well be generated from genuine viewers.

Table 6: Purchased traffic characteristics.

Vendor	IP Sources	% US IPs	% Blacklisted	Unique User Agents	% Mobile User Agents	% Zero Sized
AeTraffic	17,075	93.14	.99	3,331	5.44	34.08
BuildTraffic	1,079	.17	1.75	312	14.42	NA ¹
BuyBulkVisitor	1	100	0	Unknown ²	0	Unknown ²
MaxVisits	8,551	98.83	.47	1,883	4.03	NA ¹
TrafficMasters	14,489	99.29	1.65	3,096	4.59	47.34

¹ Detailed tracking not implemented at time of purchase

² User failed to load JavaScript

About 5% of the traffic from AeTraffic, MaxVisists and TrafficMasters had the User Agent signature of a mobile device. BuildTraffic traffic contained a much higher percentage of mobile device User Agents. Possibly due to the increased geographic diversity of the traffic.

4.2.7 Viewport Size

Halfway through our purchases we instrumented the code to record the element height and width.³ Overall 46.51% of ad views had a height or width of 0, meaning that the advertisement could not possibly be viewed by the user. 13.42% of views had both a height and width of zero. These results corroborate the BuildTraffic delivery technique of zero-sized frames described in 4.3.1.

4.3 Pay-Per-View Networks

By examining the JavaScript provided by traffic generation services and the referer fields from traffic on our honeypot sites, we were able to identify the fact that traffic was generated primarily from pop-under windows. Interestingly, while we did see evidence of traffic from expired domains, we saw no evidence of traffic from botnets. This observation led to our deeper investigation of the use of pop-unders for traffic generation and our characterization of PPV networks.

As noted above when publishers participate in a traffic generation service *i.e.*, a PPV network, they are given a block of JavaScript to place on their site, which looks very much like a standard ad tag. In the case of PPV networks, when a user accesses a PPV network publisher page, the JavaScript opens a new window (typically behind the active browser window, hence a pop-under) and loads the PPV server URL. The publisher running the tag gets a share of the revenue for every PPV URL that is subsequently loaded. The PPV network solves two problems with respect to marshaling users: (*i*) it delivers the JavaScript which creates the pop-under window and (*ii*) it determines the site to display in the window.

In response to prevalent pop-up advertising, web browsers give users the option to prevent pages from opening unsolicited windows. PPV networks need to circumvent this restriction. One option is the PPV code can explicitly bypass browser protections. A review of the issue trackers for Chrome or Firefox does not list many bugs related to the browsers' pop-up blockers, thus this is likely to be a difficult coding challenge. Our empirical data did not show any PPV network tags that attempt to bypass the pop-up blocker directly. The common approach is to tie pop-under creation to a user action since browsers typically allow creation of new windows on these events. Typically the pop-under action is attached to the onclick event of the body of the page. This causes the pop-under action to fire whenever the user clicks anywhere on the site.

After creation, the pop-under window is directed to load a specific URL pointing to the network's ad server. The ad server URL contains a number of parameters describing targeting and attribution of the visitor. The parameters always include an identifier for the originating site so that the publisher can get paid for the traffic. The list of parameters is clearly dependent on individual implementations, but some of the more common targeting parameters are: (*i*) userToken, (*ii*) indication if adult sites are allowed, (*iii*) user IP/geolocation, and (*iv*) viewport size. Using these parameters the ad server selects and returns the most profitable 3rd-party web sites (*i.e.*, the publishers that have purchased traffic) available. This is presumably the point where the 24 hour unique user guarantee is enforced.

Manually loading a publisher's PPV network tag often showed multiple redirections through a network of PPV servers. This mimics what is seen in standard advertising networks where an individual ad can be redirected across many networks in order to optimize the return from each user. For example, repeatedly loading the InfinityAds publisher tag showed network connections being made to ads.lzjl.com, cpxcenter.com, and 199.21.148.39. Whois and reverse IP lookups on these all indicate YesUp eCommerce Solutions Inc. for the contact information. YesUp is located in Ontario Canada and has a host of eCommerce offerings.

Ideally we would have identified the referer to the main pop-under page in our purchased traffic. This would enable us to identify the sites hosting pop-under tags. Un-

³Using `document.documentElement.clientHeight`, `document.body.clientHeight`, `window.innerHeight` depending on browser type.

fortunately, the sandboxing of child frames (especially child frames with different domains than the parent) protects the Document Object Model of the parent frame. Therefore, the `document.referrer` node of the parent is inaccessible to the child frame. None of our traffic purchases had a value for `parent.document.referrer`. The best we can do is the referer value of the frame loading our honeypot sites. This referer points to the origin of the pop-under window code (originating from the PPV service provider).

4.3.1 Delivery Analysis

In order to gain a better understanding of how traffic is delivered to purchasing sites, we reviewed the pages listed in the referer fields for the traffic arriving at our honeypot sites. A closer examination of two of the referer sites (BuildTraffic and RealTrafficSource) showed methods for increasing the number of "page views" for every user delivered.

Loading the referer of traffic purchased from BuildTraffic resulted in a simple HTML page, including 11 frames (see Appendix for example code). The main frame loads the primary target destination in 100% of the browser viewport. Following the primary frame there are 10 frames with a height of 0 pixels. Each of these frames loads the URL of a PPV network customer. Eight of the frames load paths from a link shortening service (itssl.com), which resolve directly to a number of sites (presumably those purchasing traffic). One of the frames loads another targeting link from the same network. The final frame loads a targeting link from yet another domain. Resulting in a total of 11 "page loads" each time the PPV network URL is loaded. Ten of those page loads are invisible to the end user because they are loaded in a frame 0 pixels high.

The page listed as the referer in traffic from RealTrafficSource also used a frame to load the final destination. In this case only a single frame covered the entire viewport, but the outer page reloaded itself every 15 seconds. When the page is displayed as a pop-under it will continue to load a different site every 15 seconds even if the pop-under window is not visible to the user.

4.4 PPV Network Throughput

Based on our evaluation of the pop-under mechanisms used by PPV networks, we endeavored to assess the broader issues of the scale of these networks (*e.g.*, number of publisher sites and number of users) and the potential volume of impressions that are being delivered on a daily basis. While all of this analysis is approximate and is based on certain assumptions, we take a conservative approach and argue that our results provide a meaningful depiction of this threat.

4.4.1 Self Reported Network Data

Many of the PPV providers list the throughput details (unique users and page views) of their network in advertising materials. Clearly, when self reporting these numbers, PPV network providers have incentive to over state in or-

der to make their network appear larger than their competitors. Nonetheless, the self reported numbers give an insight into at least the approximate size of the networks. None of the providers publish throughput numbers broken down by features or delivery mechanisms. Thus, the numbers include pop-unders, expired domains and any other generation techniques. As shown in Table 4, 8 of the providers offer throughput information. An average of 17.16M unique visitors and 6.29B page views per provider per day are claimed. While the self report by TrafficMasters on page views is much higher than others and could be false, it may be due to an extensive affiliate network. Indeed, the use of affiliate networks means that simple summation of throughput to assess scope is unlikely to be accurate. However, the self reported numbers still point to a sizable capacity for PPV networks.

Table 7: Self reported network throughput from PPV providers.

Site	Daily Visitors	Daily Deliveries
CMK Marketing	2M	25M
HitPro	40.5M	112M
TrafficElf	20M	45M
BuildTraffic	3.3M	-
FullTraffic	20M	-
TopTrafficWholesaler	-	30M
BringVisitor	-	26.6M
TrafficMasters	-	37.5B

4.4.2 Volume Estimation

In order to estimate throughput of the networks we investigated the scope of the deployment of the PPV network tags across publisher sites. Given the publisher sites where the PPV network tags are present along with the estimated traffic for those sites we create a conservative estimate for the daily traffic across PPV networks.

The first step in determining where the PPV network tags are deployed is identifying the tag URLs. The PPV networks we considered commonly used a domain name for their delivery infrastructure that was different from the public facing websites that market to publishers. We used three techniques to identify 10 active PPV network tag URLs: (*i*) subscribing to a PPV network as a publisher, (*ii*) investigating referer fields and (*iii*) searching for ad code on public forums.

Where possible we utilized automated signup processes to harvest PPV tags directly from the publishers. This is a trivial case where the code to be run on the publisher site is directly supplied.

Using referer fields to identify PPV tags was more challenging. Typically the destination is loaded inside a frame, so the referer references the outer page hosting the frame. The display page is typically not loaded directly from the publisher site. The publisher loads JavaScript which handles

Table 8: Estimated pop-under window loads per day.

Network	Tag Count	Domains	Domain Traffic	Subdomain Traffic	Total Estimate (Views/Day)
adsrevenue.net	1,797	21	802,815	128	802,943
adversalservers.com	93,060	269	1,185,769	14,168	1,199,937
clicksor.com	855,268	2,801	24,741,249	909,649	25,650,898
edomz.com	21,750	62	971,409	11,244	982,653
ero-advertising.com	2,691,930	5,830	100,664,523	69,110	100,733,633
flagads.net	36,382	102	2,294,143	2,023	2,296,166
lzjl.com	195,406	1,192	17,427,379	425,839	17,853,218
popadscdn.net	245,302	1,029	17,016,554	124,463	17,141,017
poponclick.com	28,521	164	2,651,188	2,467	2,653,655
visit-tracker.com	90	38	623,344	0	623,344
Total	4,169,506	11,508	168,378,373	1,559,091	169,937,464

the pop-under creation and then calls the display page to fill the newly created window. In some cases, both the display page and the pop-under JavaScript are hosted on the same infrastructure. Searching the Common Crawl [2] database for the infrastructure domain lead to the identification of a number of PPV tags.

Finally, entering PPV network names into search engines resulted in a number of forum posts discussing pop-under tags. Many of the tags collected this way were no longer in use, but there were a few that were still active.

The next step is identifying the publisher sites that have deployed PPV network tags. To do this we used the Common Crawl repository of web crawl data. The August 2012 (see Table 4 for details provided by [11]) dataset included derived metadata about all of the crawled URLs. The metadata dataset contained a list of all outgoing links for each crawled page (including loading of JavaScript files). Amazon’s Elastic MapReduce was used to list all paths with egress links pointing to the serving domains. The egress links were then manually reviewed to identify JavaScript files resulting in pop-under advertising. Selecting only pop-under tags from the MapReduce results gives a list of domains running those tags. We argue that this results in a conservative estimate of PPV networks that use pop-unders and an even more conservative estimate of PPV networks in general.

Estimates on traffic volumes on the identified publisher sites was done using public web analytics data. Alexa and Compete did not have traffic estimates for many of the domains. Thus, mustats.com was used to estimate domain traffic. A script was used to programmatically query mustats.com for traffic estimates on the identified PPV sites. We collected issued queries for 11,629 domains. MuStats returned an estimate for 10,737 of the queries. 2,635 of the returned queries estimated 0 views per day for the domain.

Subdomains posed an additional problem for traffic estimation. The web analytics products did not estimate traffic per subdomain. They only gave an estimate for the entire domain. For example, it is clear that just because *blogsofnote.blogspot.com* hosts a PPV network tag, not every do-

Table 9: August 2012 CommonCrawl database summary.

Crawl Date: January-June 2012
Data Size: 40.1TB (compressed)
Parsed URLs: 3,005,629,093
Domains: 40,600,000

main on *blogspot.com* hosts that same ad tag. Attributing all of the traffic for *blogspot.com* to a PPV network would be inaccurate.

To estimate the impact of subdomains on PPV networks, we again utilize the Common Crawl database. Our analysis counts the total number of URLs crawled for each domain that lists PPV tags. URLs with file extensions jpg, png, gif, js were removed from the total count. The final total count approximates the number of webpages and page fragments crawled for a given domain. Dividing the link count by the total crawled pages results in the percentage of pages in a domain containing links to the PPV code. This is likely a significant underestimation of reality for two reasons First, many of the URLs crawled were page fragments (where a full page is the combination of many fragments). Second, each path is given even weight despite the fact that tags are more likely to be found on high traffic pages. In any case, subdomain traffic is estimated by taking the estimated traffic for the whole domain and multiplying that by the percentage of pages inside the domain linking to the tag.

$$\text{domains} = \{123lyrics.info, serverhk.net, \dots\} \quad (1)$$

$$\text{subDomains} = \{\text{site1.blogspot.com}, \text{site2.blogspot.com}, \dots\} \quad (2)$$

$$\begin{aligned}
& \text{estimate} = \\
& \sum_{\text{domains}} \text{domainTraffic} + \\
& \sum_{\text{subDomains}} \frac{\text{linkedPages}}{\text{totalPages}} * \text{domainTraffic}
\end{aligned} \tag{3}$$

Our final algorithm for calculating PPV network throughput is then the estimated traffic for domains hosting PPV tags plus the proportional estimated traffic for subdomains containing PPV tags as shown in Equation 3. Our estimates only include the traffic expected from pop-under tags. Obviously, by including traffic from expired domains and typo squatting domains and bots would likely increase the estimated throughput substantially.

Table 8 shows throughput estimates for a selection of 10 PPV networks using our algorithm. As is expected from our conservative approach, the dominant portion of estimated traffic was to full domains with subdomain estimates making up a small portion of the total estimate. The PPV tags from ero-advertising.com, which is the largest PPV network, were displayed predominantly on publishers hosting adult content. It is possible that visitors browsing adult content are more tolerant of pop-under advertising.

So far we have estimated the number of times that pop-under code is executed per day. In reality many users have browser add-ons that prevent the creation of the pop-under window. One such popular extension for Firefox and Chrome is Adblock Plus [1]. The Firefox add-ons page for Adblock Plus lists 15.6M users [4]. Firefox claims 450M users [7], giving an install rate of 3.5% for Adblock Plus on Firefox. We conservatively estimate one quarter of all page loads prevent pop-up/pop-under creation due to plugins. Given this, we still would expect 75% of the estimated loads to result in a pop-under window. Our investigation of delivery mechanisms shows that PPV networks can load up to 11 destinations or more (in the case of auto refresh) in a single pop-under window. To maintain our conservative approach we assume four destinations loaded per pop-under window. Combining the effect of pop-up blockers and multiple loads we expect each view of a page hosting pop-under code will deliver 3 ($0.75 * 4$) impressions to the PPV network.

Our calculation of throughput for just 10 publisher networks resulted in more than 160M estimated tag loads per day, thus more than 500M visitor deliveries per day. Assuming a modest price of \$25 per 25k visitors, the PPV providers make a minimum of \$15M in sales of targeted traffic per month. Those 15B page views per month are delivered to purchasing websites. Assume the purchasing websites contain an average of 4 ads and each of those ads pays a \$0.25 CPM. Advertisers spend \$15M a month advertising to pop-under viewers on these 10 networks alone.

5. PAY-PER-VIEW NETWORK COUNTERMEASURES

In this section, we describe three potential counter measures to address the problem of invalid impressions gener-

ated by PPV networks. Each method offers a different perspective on the threat and each offers a different capability in terms of what can be done about the threat. While there could certainly be other viable counter measures, the following methods can be implemented by participants in the ad ecosystem who would benefit by detection and/or prevention of invalid impressions via PPV networks.

5.1 Viewport Size Filters

Advertisers who run their own ad server or intermediaries who run ad servers who are interested in removing impressions from PPV networks can filter ad requests based on viewport size. An advertiser or intermediary could implement a viewport size check countermeasure by augmenting their current JavaScript tag to include code that ensures a minimum sized viewport. This simple check code would prevent display of the advertisement for viewports which are too small to reasonably be seen by users on target platforms. In addition to reducing invalid impressions, this approach would save advertisers the bandwidth costs of delivering creatives in PPV networks.

JavaScript that detects zero-sized viewports could prevent a large amount of invalid impressions. Over 46% of the impressions in our data corpus are delivered via zero-sized viewports. Assuming this approach is used by PPV networks writ large, we estimate that a zero size viewport filter could block impressions from loading on over 200M pages per day from just the 10 PPV networks we investigated.

5.2 Referer Blacklist

Participants in the ad ecosystem could also use blacklists to identify and block traffic originating from PPV networks. We found that the referer field identifies a source in the majority of the traffic that we purchased. Over time, a blacklist of referers could be built that identifies traffic originating from a large number of PPV networks. This is similar to browser ad-blocking add-ons or in-network solutions that utilize a blacklist to remove undesired traffic. The difference with the referer blacklist is that the advertiser or intermediary implements the list directly. One limitation of this approach is that it will only work if no iframes are in use since iframes would prevent the advertiser code from accessing the referer.

Similar to viewport size filters, an advertiser/intermediary could incorporate the blacklist into their ad tags in order to prevent display to questionable viewers. As a passive alternative an advertiser could simply log the referers and compare them against the blacklist at a later time. Then the advertiser can use the information in negotiations with their advertising network.

The blacklist will need continual tuning as new PPV networks emerge and old networks disappear. One drawback of this approach is that a savvy adversary can trivially defeat this method by clearing or altering the referer field. There is some evidence that this is already happening. A few of

the referer strings in our data corpus contained direct IP addresses instead of DNS names, possibly to thwart existing or suspected blacklist methodology or simply to obfuscate their behavior. Even so a referer blacklist based on domain names would have prevented 99.51% of our purchased traffic.

5.3 Publisher Blacklists

An alternative approach is to create and maintain a blacklist of publishers that participate in PPV networks. Similar to countermeasures described above, this list could be used by advertisers to avoid running their display advertising on sites sourcing traffic from the PPV networks. This somewhat strong-armed approach would be likely to get the attention of publishers very quickly since we assume at least some percentage may not be aware of the negative aspects of their participation. Even if a publisher was aware, such an approach would discourage them from engaging with invalid traffic. Thus, this method could have potential benefits to the entire advertising ecosystem.

Publisher blacklists can be implemented by the advertiser in their tag as either preventative or informative, similar to the referer blacklist. Again this list will need continual updates as publisher behavior changes. One method of generating a publisher blacklist is to isolate and repeatedly call the PPV destination selection code block. This would enumerate all possible destinations for that PPV network over time.

6. RELATED WORK

General aspects of online advertising have been discussed in a large number of studies over the past decade. These studies have focused on wide variety of issues including the economic aspects of advertising *e.g.*, [17, 18], theoretical or analytical evaluations of sponsored search and ad auctions *e.g.*, [13, 35, 37] and more recently ad exchanges *e.g.*, [14, 30]. However, there are relatively few examples of empirical characterization studies of online advertising, most likely due to the private nature of advertising data. Relatively recent empirical studies include [19, 26, 31, 32, 39], which provide informative insights on key assumptions made in theoretical studies as well as recommendations that improve the effectiveness of online advertising.

Google, Microsoft, Yahoo and other large industry players have online documentation about their invalid traffic monitoring activities (although no significant technical details are disclosed) [21, 24, 38]. This is given to raise trust for advertisers. However, many platforms offered by intermediaries have almost no documentation on fraud. What is clear is that detecting and preventing fraud in advertising networks presents significant challenges [33, 36].

The problem of fraud in online advertising has been the subject of many different studies over the years. The majority of these studies have focused on fraud in PPC-based environments. Botnets are well known to be used for click fraud. One example of a large-scale botnet focus on click

fraud was the Bamital botnet, which was recently dismantled [25]. Similarly, the ZeroAccess botnet can generate fraudulent clicks estimated to cost advertisers over \$900K/day in lost revenue [12]. Other studies have focused on developing methods for detecting click-fraud *e.g.*, [28, 40]. Haddadi describes bluff ads as a means for measuring click fraud activity and creating blacklists for IP addresses to reduce click fraud [22]. Dave *et al.* [16] developed a novel measurement methodology to gather data on click fraud in ad networks. Their work informs our measurement efforts. Another recent empirical study by Zhang *et al.* is perhaps most similar to our work in terms of measurement methods [41]. In that study, the authors purchased traffic aimed at a honeypot website, and reported on a range of characteristics. Our findings on the characteristics of purchased traffic are in line with theirs, although we only purchased impression traffic and did not focus on click-through in our study.

Finally, several recent studies have included brief discussions of impression fraud. In particular, Stone-Gross *et al.* use logs from a large online ad exchange to investigate a variety of characteristics that relate to invalid activity, including behaviors related to impression spam [34]. Our work differs from prior studies principally in its focus on impression fraud. To the best of our knowledge there are no prior studies that investigate impression fraud in depth from an empirical perspective, or that investigate PPV networks and their characteristics.

7. SUMMARY AND CONCLUSIONS

Internet-based advertising is a large and growing industry. Search-based advertising still dominates in terms of annual expenditures, however display and video advertising have seen significant growth over the past several years. While publishers have always been motivated to use diverse methods to drive users to their sites, the fact that payments for display and video ads are often based on impressions motivates new offerings from 3rd-party traffic generation services.

In this paper, we investigate the problem of invalid traffic generation that is aimed at inflating impressions on publisher websites and apps. We address this problem empirically by setting up several honeypot websites that were used as the targets for traffic generation purchases, which we made over the course of several months. This traffic provides the baseline from which we were able to identify a particular form of impression generation that we call pay-per-view networks. A PPV network is a series of legitimate publisher sites that include a common embedded reference from a particular traffic generation service. When users access publisher sites that participate in PPV networks, 3rd-party websites are rendered in an obfuscated and often invisible fashion. By evaluating the JavaScript associated with PPV networks, we find that the predominate mechanism used is pop-under windows. We also find that PPV networks place multiple 3rd-party pages on pop-unders using frames or use periodic refresh to leverage every user access. This approach

preserves the user experience on the publisher’s site and generates invalid impressions on the 3rd-party sites in a way that is difficult to detect.

Next, we investigate aspects of the broader scope of PPV networks by gathering information from a small selection of ten traffic generation services. We search for tags from these services in a publicly available Internet-wide crawl database to estimate deployments on publisher sites. We couple these estimates with estimates for daily unique page views from those sites and find tag throughput above 150M per day. Combined with conservative estimates of 3rd-party displays per tag and ad placements per page, this easily pushes the number of invalid impressions above 500M per day from these ten PPV networks alone. Based on the fact that our sampling is so small, the impact of PPV networks is likely to be much larger.

To address the threat of PPV networks, we describe three different counter measures. Each offers a different constituency an opportunity to block the display of the unwanted 3rd-party content. In future work, we plan to focus on developing implementations of the proposed counter measures as well as developing other techniques to address this threat. Our measurement and characterization work are ongoing and will soon focus on traffic generation services outside of North America.

Acknowledgements

The authors would like to thank our shepherd, Chris Grier, for his input and multiple reviews of the paper. This work was supported in part by NSF grants CNS-0831427, CNS-0905186, ARL/ARO grant W911NF110227 and the DHS PREDICT Project. Any opinions, findings, conclusions or other recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF, ARO or DHS.

8. REFERENCES

- [1] Adblock Plus. <http://adblockplus.org>, 2013.
- [2] Common Crawl. <http://commoncrawl.org/>, 2013.
- [3] Dshield Daily Sources.
http://www.dshield.org/feeds/daily_sources, 2013.
- [4] Firefox Add-Ons - Adblock Plus.
<https://addons.mozilla.org/en-US/firefox/addon/adblock-plus/>, February 2013.
- [5] Google Analytics URL Builder.
<http://support.google.com/analytics/bin/answer.py?hl=en&answer=1033867&topic=1032998&ctx=topic>, 2013.
- [6] McAfee SiteAdvisor. <http://www.siteadvisor.com/>, 2013.
- [7] Mozilla Press Center.
<http://blog.mozilla.org/press/ataglance/>, February 2013.
- [8] Network Solutions Whois.
<http://www.networksolutions.com/whois>, 2013.
- [9] SameID.net. <http://sameid.net/>, 2013.
- [10] UCEPROTECT Blacklist.
<http://rsync-mirrors.uceprotect.net/rbldnsd-all/ips.backscatterer.org.gz>, 2013.
- [11] Web Data Commons - Extraction Results from the August 2012 Common Crawl Corpus.
<http://webdatacommons.org/#toc2>, 2013.
- [12] ZeroAccess is Top Bot in Home Networks.
<http://www.infosecurity-magazine.com>, February 2013.
- [13] G. Aggarwal, J. Feldman, S. Muthukrishnan, and M. Pai. Sponsored Search Auctions with Markovian Users. *Internet and Network Economics, Lecture Notes in Computer Science*, 5385, 2008.
- [14] S. Balseiro, J. Feldman, v. Mirrokni, and S. Muthukrishnan. Yield Optimization of Display Advertising with Ad Exchange. In *Proceedings of the ACM Electronic Commerce '11*, San Jose, CA, June 2011.
- [15] Internet Advertising Board. IAB Internet Advertising Revenue Report 2012 First Six Months Results.
<http://www.iab.net>, October 2012.
- [16] V. Dave, S. Guha, and Y. Zhang. Measuring and Fingerprinting Click-Spam in Ad Networks. In *Proceedings of ACM SIGCOMM '12*, Helsinki, Finland, August 2012.
- [17] B. Edelman, M. Ostrovsky, and M. Schwarz. Internet Advertising and the Generalized Second Price Auctions: Selling Billions of Dollars Worth of Keywords. *American Economic Review*, 2007.
- [18] D. Evans. The Economics of the Online Advertising Industry. *Review of Network Economics*, 7(3), 2008.
- [19] A. Ghose and S. Yang. An Empirical Analysis of Search Engine Advertising: Sponsored Search in Electronic Markets. *Management Science*, 55(10), July 2009.
- [20] Google. Adexchange.
<http://www.google.com/adwords/watchthisspace/solutions/ad-exchange/>, 2013.
- [21] Inc Google. Ad Traffic Quality Resource Center.
<http://www.google.com/ads/adtrafficquality/index.html>, 2013.
- [22] H. Haddadi. Fighting Online Click-fraud Using Bluff Ads. *ACM SIGCOMM Computer Communications Review*, 40(2), 2010.
- [23] InfinityAds. Publisher Signup.
<http://www.infinityads.com>, 2013.
- [24] T. Kelleher. How Microsoft Advertising Helps Protect Advertisers from Invalid Traffic.
<http://community.bingads.microsoft.com>, November 2011.
- [25] J. Kirk. Microsoft, Symantec Take Down Bamital

- Click-fraud Botnet. <http://www.infoworld.com>, February 2013.
- [26] S. Lahaie and P. McAfee. Efficient Ranking in Sponsored Search. In *Proceedings of the Seventh Workshop on Ad Auctions*, San Jose, CA, July 2011.
- [27] I. Lunden. Forrester: US Online Display Ad Spend \$12.7B In 2012, Rich Media and Video Leading The Charge. <http://www.techcrunch.com>, October 2012.
- [28] A. Metwally, D. Agrawal, and A. El Abbadi. Using Association Rules for Fraud Detection in Web Advertising Networks. In *Proceedings of the International Conference on Very Large Databases*, Trondheim, Norway, August 2005.
- [29] MuStat. MuStat. <http://www.mustat.com>, 2013.
- [30] S. Muthukrishnan. AdX: A Model for Ad Exchanges. *ACM SIGEcon Exchanges*, 8(2), 2009.
- [31] M. Ostrovsky and M. Schwarz. Reserve Prices in Internet Advertising Auctions: A Field Experiment. In *Proceedings of the Sixth Workshop on Ad Auctions*, Cambridge, MA, July 2010.
- [32] N. Vallina Rodriguez, J. Shah, A. Finamore, Y. Grunenberger, K. Papaginnaki, H. Haddadi, and J. Crowcroft. Breaking for commercials: characterizing mobile advertising. In *Proceedings of ACM Internet Measurement Conference (IMC '12)*, Boston, MA, November 2012.
- [33] B. Schwartz. Google: Investigating Invalid AdSense Traffic is Extremely Difficult. <http://www.seroundtable.com>, April 2012.
- [34] B. Stone-Gross, R. Stevens, R. Kemmerer, C. Kruegel, G. Vigna, and A. Zarras. Understanding Fraudulent Activities in Online Ad Exchanges. In *Proceedings of ACM Internet Measurement Conference (IMC '11)*, Berlin, Germany, November 2011.
- [35] C. Tucker and A. Goldfarb. Search Engine Advertising: Pricing ads to context. In *Proceedings of the Fourth Workshop on Ad Auctions*, Chicago, IL, July 2008.
- [36] A. Tuzhilin. The Lane's Gifts v. Google Report. http://googleblog.blogspot.com/pdf/Tuzhilin_Report.pdf, 2006.
- [37] H. Varian. Position Auctions. *International Journal of Industrial Organization*, 25, 2007.
- [38] Yahoo. Traffic Quality: We Work to Protect You in a Variety of Ways. <http://advertisingcentral.yahoo.com>, 2013.
- [39] J. Yan, N. Liu, G. Wang, W. Zhang, Y. Jiang, and Z. Chen. How Much can Behavioral Targeting Help Online Advertising? In *Proceedings of WWW '09*, Madrid, Spain, April 2009.
- [40] L. Zhang and Y. Guan. Detecting Click Fraud in Pay Per Click Streams of Online Advertising Networks. In *Proceedings of the International Conference on Distributed Computing Systems*, Beijing, China, June 2008.
- [41] Q. Zhang, T. Ristenpart, S. Savage, and G. Voelker. Got Traffic? An Evaluation of Click Traffic Providers. In *Proceedings of WebQuality '11*, Hyderabad, India, March 2011.

APPENDIX

Traffic Delivery Code

```
...
<frameset rows='0,*' framespacing='0' border='0' frameborder='0'>
<frame name='header' src='about:blank' scrolling='no' noresize>
<frame name='main' src=" +rurl+ " scrolling='auto'>
<frameset rows='0,*' framespacing='0' border='0' frameborder='0'>
<frame name='raaj1' src='http://itssl.com/37kt' scrolling='no' noresize>
<frameset rows='0,*' framespacing='0' border='0' frameborder='0'>
<frame name='house2' src='http://stats.itssl.com/?VFJDSz0zNA==' scrolling='no' noresize>
<frameset rows='0,*' framespacing='0' border='0' frameborder='0'>
<frame name='house3' src='http://stats.itssl.com/?VFJDSz0zNA==' scrolling='no' noresize>
<frameset rows='0,*' framespacing='0' border='0' frameborder='0'>
<frame name='usnopop' src='http://stats.itssl.com/?VFJDSz00' scrolling='no' noresize>
<frameset rows='0,*' framespacing='0' border='0' frameborder='0'>
<frame name='usnopop2' src='http://stats.itssl.com/?VFJDSz00' scrolling='no' noresize>
<frameset rows='0,*' framespacing='0' border='0' frameborder='0'>
<frame name='usnopop3' src='http://stats.itssl.com/?VFJDSz00' scrolling='no' noresize>
<frameset rows='0,*' framespacing='0' border='0' frameborder='0'>
<frame name='geo1' src='http://www.itssl.com/georedirect/main.html' scrolling='no' noresize>
<frameset rows='0,*' framespacing='0' border='0' frameborder='0'>
<frame name='geo2' src='http://www.itssl.com/georedirect/main.html' scrolling='no' noresize>
<frameset rows='0,*' framespacing='0' border='0' frameborder='0'>
<frame name='raaj2' src='http://stats.buildtraffic.com/?VFJDSz01OA==' scrolling='no' noresize>
<frameset rows='0,*' framespacing='0' border='0' frameborder='0'>
<frame name='georedirect' src='http://adzay.com/redirect.php' scrolling='no' noresize>
...
...
```

The Velocity of Censorship: High-Fidelity Detection of Microblog Post Deletions

Tao Zhu

zhusuo777@gmail.com

Independent Researcher

David Phipps

Computer Science

Bowdoin College

Adam Pridgen

Computer Science

Rice University

Jedidiah R. Crandall

Computer Science

University of New Mexico

Dan S. Wallach

Computer Science

Rice University

Abstract

Weibo and other popular Chinese microblogging sites are well known for exercising internal censorship, to comply with Chinese government requirements. This research seeks to quantify the mechanisms of this censorship: how fast and how comprehensively posts are deleted. Our analysis considered 2.38 million posts gathered over roughly two months in 2012, with our attention focused on repeatedly visiting “sensitive” users. This gives us a view of censorship events within minutes of their occurrence, albeit at a cost of our data no longer representing a random sample of the general Weibo population. We also have a larger 470 million post sampling from Weibo’s public timeline, taken over a longer time period, that is more representative of a random sample.

We found that deletions happen most heavily in the first hour after a post has been submitted. Focusing on original posts, not reposts/reweets, we observed that nearly 30% of the total deletion events occur within 5–30 minutes. Nearly 90% of the deletions happen within the first 24 hours. Leveraging our data, we also considered a variety of hypotheses about the mechanisms used by Weibo for censorship, such as the extent to which Weibo’s censors use retrospective keyword-based censorship, and how repost/reweet popularity interacts with censorship. We also used natural language processing techniques to analyze which topics were more likely to be censored.

1 Introduction

Virtually all measurements of Internet censorship are biased in some way, simply because it is not feasible to test every keyword or check every post at small increments of time. In this paper, we describe our method for tracking censorship on Weibo, a popular microblogging platform in China, and the results of our measurements. Our system focuses on a core set of users who are in-

terconnected through their social graph and tend to post about sensitive topics. This biases us towards the content posted by these particular users, but enables us to measure with high fidelity the speed of the censorship and discern interesting patterns in censor behaviors.

Sina Weibo (weibo.com, referred to in this paper simply as “Weibo”) has the most active user community of any microblog site in China [39]. Weibo provides services which are similar to Twitter, with @usernames, #hashtags, reposting, and URL shortening. In February 2012, Weibo had over 300 million users, and about 100 million messages sent daily [3]. Like Twitter in other countries, Weibo plays an important role in the discourse surrounding current events in China. Both professional reporters and amateurs can provide immediate, first-hand accounts and opinions of events as they unfold. Also like Twitter, Weibo limits posts to 140 characters, but 140 characters in Chinese can convey significantly more information than in English. Weibo also allows embedded photos and videos, as well as comment threads attached to posts.

China employs both backbone-level filtering of IP packets [5, 6, 11, 23, 37, 43] and higher level filtering implemented in the software of, for example, blog platforms [15, 20, 28], chat programs [13, 29] and search engines [30, 41]. Work specific to Weibo [2, 9] is discussed in more detail in Section 2. To our knowledge ours is the first work to focus on how *quickly* microblog posts are removed—on a scale of minutes after they are posted. This fidelity in measurement allows us to not only accurately measure the speed of the censorship, but also to compare censorship speeds with respect to topics, censor methods, censor work schedules, and other illuminating patterns.

What our results illustrate is that Weibo employs “defense-in-depth” in their strategy for filtering content. Internet censorship represents a conflict between the censors, who seek to filter content according to some policy, and the users who are subject to that censorship. Censor-

ship can serve to squelch conversations directly as well as to chill future discussion with the threat of state surveillance. Our goal in this paper is to catalog the wide variety of mechanisms that Weibo’s censors employ.

This research has several major contributions:

- We describe the implementation of a method that can detect a censorship event within 1–2 minutes of its occurrence. A large amount of Weibo posts are collected constantly via two APIs [26]. There are more than 470 million posts from the public timeline and 2.38 million posts from the user timeline in our database.
- To further understand how the Weibo system can react so quickly in terms of deleting posts with sensitive content, we propose four hypotheses and attempt to support each with our data. We also describe several experiments that shed light on censorship practices on Weibo. The overall picture we illuminate in this paper is that Weibo employs a distributed, defense-in-depth strategy for removing sensitive content.
- Using natural language processing techniques that overcome the usage of neologisms, named entities, and informal language which typifies Chinese social media, we perform a topical analysis of the deleted posts and compare the deletion speeds for different topics. We find that the topics where mass removal happens the fastest are those that are hot topics in Weibo as a whole (*e.g.*, the Beijing rainstorms or a sex scandal). We also find that our sensitive user group has overarching themes throughout all topics that suggest discussion of state power (*e.g.*, Beijing, government, China, and the police).

The rest of this paper is structured as follows. Section 2 gives some basic background information about microblogging and Internet censorship in China. Then Section 3 describes the methods we used for our measurement and analysis, followed by Section 4 that describes the timing of censorship events. Section 5 introduces the natural language processing we applied to the data and presents results from topical analysis. Finally, we conclude with a discussion of various Weibo filtering mechanisms in Section 6.

2 Background

Starting from 2010, when microblogs debuted in China, not only have there been many top news stories where the reporting was driven by social media, but social media has also been part of the story itself for a number of prominent events [21, 38], including the protests of

Wukan [33], the Deng Yujiao incident [32], the Yao Jiaxin murder case [35], and the Shifang protest [36]. There have also been events where social media has forced the government to address issues directly, such as the Beijing rainstorms in July 2012.

Chinese social media analysis is challenging [27]. One of many concerns that can hinder this work is the general difficulty of mechanically processing Chinese text. Western speakers (and algorithms) expect words to be separated by whitespace or punctuation. In written Chinese, however, there are no such word boundary delimiters. The word segmentation problem in Chinese is exacerbated by the existence of unknown words such as named entities (*e.g.*, people, companies, movies) or neologisms (substituting characters that appear similar to others, or otherwise coining new euphemisms or slang expressions, to defeat keyword-based censorship) [12]. Furthermore, since social media is heavily centered around current events, it may well contain new named entities that will not appear in any static lexicon [8].

Despite these concerns, Weibo censorship has been the subject of previous research. Bamman *et al.* [2] performed a statistical analysis of deleted posts, showing that the presence of some sensitive terms indicated a higher probability of the deletion of a post. Their work also showed some geographic patterns in post deletion, with posts from the provinces of Tibet and Qinghai exhibiting a higher deletion rate than other provinces. WeiboScope [9] also collects deleted posts from Weibo, but their strategy is to follow all users with a high number of followers. This is in contrast to our strategy which is to follow a core set of users who have a high rate of post deletions, some of which have many followers and some of which have few. The deletion events in these works are measured with a resolution of hours or days. Our system is able to detect deletion events at the resolution of minutes.

3 Methodology

To have a better understanding of what the Weibo system is targeting for censorship deletions, and how fast they do so, we have developed a system which collects removed posts on targeted users in almost real time.

3.1 Identifying the sensitive user group

In Weibo each IP address and Application Programming Interface (API) has a rate limit for access to the service. This forced us to make a number of engineering compromises, notably focusing our attention where we felt we could find those posts most likely to be subject to censorship. We decided to focus on users who we have

seen being censored in the past, under the assumption that they will be more likely to be censored in the future. We call this group of users the *sensitive group*.

We started with 25 sensitive users that we discovered manually, leveraging a list from China Digital Times [4] of sensitive keywords which are not allowed to be searched on Weibo’s server. To find our initial sample, we searched using out-dated keywords that were later un-banned. For example, 党产共 (Reverse of 共产党, which means “Communist Party”) was found to be banned on 4 April 2011, but found to not be banned on 20 October 2011, which means the we were able to obtain some posts containing 党产共 when we searched for this keyword after 20 October 2011. From the search results, we picked 25 users who stood out for posting about sensitive topics.

Next, we needed to broaden our search to a larger group of users. We assumed that anybody who has been reposted more than five times by our sensitive users must be sensitive as well. We followed them for a period of time and manually measured how often their posts were deleted. Any user with more than 5 deleted posts was added to our pool of sensitive users.

After 15 days of this process, our sensitive group included 3,567 users, and within this group we observed more than 4,500 post deletions daily, including about 1,500 “permission denied” deletions. (See Section 3.3 for discussion on different types of deletion events.) Roughly 12% of the total posts from our sensitive users were eventually deleted. Further, we have enough of these posts to be able to run topical analysis algorithms, letting us extract the main subjects that Weibo’s censors seemed concerned with on any given day.

We contrast these statistics with WeiboScope [9], developed at the University of Hong Kong in order to track trends on Weibo concurrently with our own study. The core difference between our work and WeiboScope is that they track a large sample: around 300 thousand users who each have more than 1000 followers. Despite this, they report observing no more than 100 “permission denied” deletions per day. WeiboScope’s results, therefore, are perhaps more representative of the overall impact of Weibo’s censorship as a fraction of total Weibo traffic, while our work has more resolving power to consider the speed and techniques employed by Weibo’s censors.

Because we do not have access to WeiboScope’s data, we are limited in our ability to make direct comparisons of our datasets. They did briefly support data downloads, and we extracted their “2,500 last permission denied data” on 20 July 2012. This service has since been closed. Our system went live following user timelines on the same date, giving us a single day from which we might compare our data. For 20 July 2012, WeiboScope observed 54 permission-denied posts, while our system

observed 1,056.

(Our own system does not yet support public, real-time downloads of our data, which among other issues could make it easier for Weibo to shut it down. An appropriate means of disseminating real-time results or regular summaries is future work for our group.)

While our methodology cannot be considered to yield a representative sample of Weibo users overall, we believe it is representative of how users who discuss sensitive topics will experience Weibo’s censorship. We also believe our methodology enables us to measure the topics that Weibo is censoring on any given day.

3.2 Crawling

Once we settled on our list of users to follow, we wanted to follow them with sufficient fidelity to see posts as they were made and measure how long they last prior to being deleted. Our target sampling resolution was one minute.

We use two APIs provided by Weibo, allowing us to query individual user timelines as well as the public timeline¹. Starting in July 2012, we queried each of our 3,500 users, once per minute, for which Weibo returns the most recent 50 posts. Deleted posts outside of this 50-post window are not detected by our system, meaning that we may be underestimating the number of older posts that get deleted.

We also queried the public timeline roughly once every four seconds, for which Weibo returns 200 recent posts. Half of these posts appear to be 1–5 minutes older than real-time, and the other half are hours older.

Weibo does not support anonymous queries to its servers, requiring us to create fake accounts on the service. Weibo further enforces rate limits both on these users’ queries as well as on source IP addresses, regardless of what user account is being used for the query. To overcome these concerns, we used roughly 300 concurrent Tor circuits [24], driven from our research computing cluster. Our resulting data was stored and processed on a four-node cluster using Hadoop and HBase [1].

If and when Weibo might make a concerted effort to block us, it is easy to imagine a ongoing game where they invent new detection strategies and we invent new workarounds. So far, this has not been an issue.

3.3 Detecting deletions

An absent post may have been censored, or it may have been deleted for any of a variety of other reasons. User

¹The user timeline returns both original posts and retweeted posts by that user, while the public timeline only returns original posts. Also, the public timeline appears to be only a sampling of the total public traffic.

accounts can also be closed, possibly for censorship purposes. Users cannot delete their own account, only the system can delete accounts. We conducted a variety of short empirical tests to see if we could distinguish the different cases. We concluded that we can detect two kinds of deletions.

If a user deletes his or her own post, a query for that post’s unique identifier will return a “post does not exist” error. We have observed this same error code returned from censorship events and we refer to these, in the remainder of the paper as *general deletion*. However, there is another error code, “permission denied,” which seems to indicate that the relevant database record still exists but has been flagged by some censorship event. We refer to these as *permission-denied deletions* or *system deletions*. In either case, the post is no longer visible to Weibo users.

The ratio of system deletions to general deletions in our user timeline data set is roughly 1:2. In this paper, we generally focus on posts that have been system deleted, because there appears to be no way for a user to induce this state. It can only be the result of a censorship event (*i.e.*, there are no censorship false positives in our system deletion dataset). Because we followed a core set of users who post on sensitive subjects, we did not find it necessary to account for spam in our user timeline dataset.

Our crawler, which repeatedly fetches each sensitive user’s personal timeline, is searching for posts that appear and then are subsequently deleted. If a post is in our database but is not returned from Weibo, then we issue a secondary query for that post’s unique ID to determine what error message is returned. Ultimately, with the speed of our crawler, we can detect a censorship event within 1–2 minutes of its occurrence.

For each returned post from Weibo, there is a field which records the creation time of the post. The lifetime of a post is the time difference between the time our system detected the post being deleted and the creation time. Therefore a post’s lifetime recorded by our system is never shorter than its real lifetime, and never longer than its real lifetime by more than two minutes.

4 Timing of censorship

For easier explanation we first give some definitions. A post can be a repost of another post, and can have embedded images. Also other users can repost reposts. If post *A* is a repost of post *B*, we call post *A* a *child post* and post *B* a *parent post*. If post *A* is not a repost of another post, we call post *A* a *regular post*.

Using our user tracking method, from 20 July 2012 to 8 September 2012, we have collected 2.38 million user timeline posts, with a 12.8% total deletion rate (4.5% for system deletions and 8.3% for general deletions). Note

that this deletion rate is specific to our users and not representative of Weibo as a whole. With a brief analysis, we found that 82% of the total deletions are child posts, and 75% of the total deletions have pictures either in themselves or in their parent post.

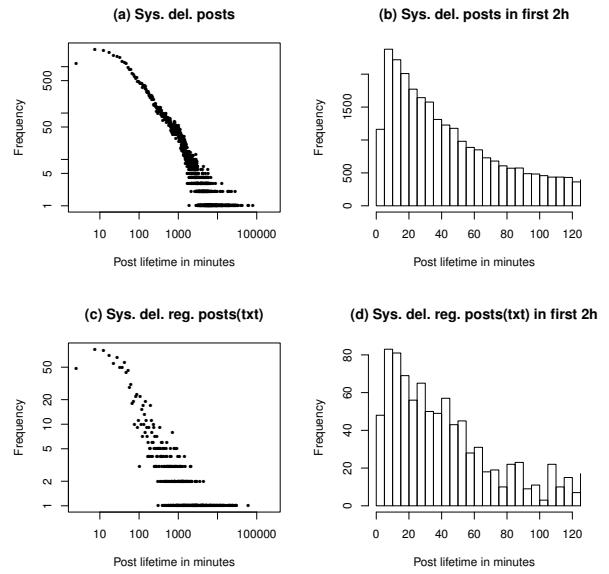


Figure 1: Lifetime histograms. (a) and (b) are the lifetime histograms of all system deletions. (c) and (d) are the lifetime histograms of regular text-only posts. (a) and (c) show the histogram of the whole lifetime, (b) and (d) only show the first two hours of the lifetime histogram.

To demonstrate how long a post survives before it gets deleted, we analyze the system deletion data set (see Section 3.3). Figure 1 gives us a big picture of how fast the Weibo system works for censorship purposes. The *x* axes are the length of the lifetime divided into 5-minute bins, and the *y* axes are the count of the deleted posts having the lifetime in the corresponding bin. We note that these figures have the distinctive shape of a power law or long tailed distribution, implying that there is no particular time bound on Weibo’s censorship activity, despite the bulk of it happening quickly, and that metrics like mean and median are not as meaningful as they are in a normal distribution.

We can see that the post bins with small lifetimes are large. We zoom into the first 2 hours of data, which is plotted in Figure 1 (c) and (d). This tells us that system deletions start within 5 minutes, the same as text-only regular posts. For both of them, the modal deletion age appears to be between 5–10 minutes.

In our data set, 5% of the deletions happened in the first 8 minutes, and within 30 minutes, almost 30% of

the deletions were finished. More than 90% of deletions happened within one day after a post was submitted. This demonstrates why a measurement fidelity on the order of minutes, rather than days, is critical.

Considering the big data set that Weibo has to process, the speed, especially the 5 to 10 minutes peak, is fast, especially considering that the data cannot be processed in a fully automated way. How can the Weibo system find sensitive posts and remove them so quickly? On the other hand, the long tails suggest that sensitive posts can still be deleted even after an extended period. How are those sensitive posts located by the moderators after a month in their huge database? What factors affect a post’s lifetime?

In this section, to find the answers to these questions, we propose four hypotheses and then test them against our data. Hypotheses 1 and 2 try to explain how the speed of censorship on Weibo can be so fast. Hypothesis 3 explains why we see the long tails of the post lifetime for censored posts in Figure 1. Hypothesis 4 tells us that the deletion speed does not appear to be strongly related to particular conversation topics, but rather to popular topics (*i.e.*, those that are being discussed on Weibo as a whole according to our public timeline) where our core sensitive users are putting a spin on the discussion that involves themes of government power (see Section 5).

4.1 Post lifetime regression

Before we give our hypotheses, we first consider what factors affect a post’s lifetime, regardless of the content of the post.

For each post, besides the basic information about the post itself, we also see an embedded picture, if present, as well as a parent post identifier, if it is a repost. Also, we know the number of followers and friends of each user, as well as of any parent post’s user.

From the graphs in Figure 1, we decided to experimentally fit a negative binomial regression to it to see which factors affect the lifetime of a post. Table 1 and Table 2 show the results for the regular posts and child posts, respectively. Three asterisks (“***”) indicates statistical significance, one asterisk (“*”) indicates a coefficient that is not statistically significant, and no coefficient is indicated with a dash (“-”). We can regress the log lifetime for a regular post or a child post via:

$$\begin{aligned} \ln(\widehat{\text{Regular}_{\text{lifetime}}}) &= \text{Intercept} + b_1(P_{\text{HasPic}}) \\ &\quad + b_2(\text{Friends \#}) + b_3(\text{Posts \#}) \\ \ln(\widehat{\text{Child}_{\text{lifetime}}}) &= \text{Intercept} + b_1(P_{\text{HasPic}}) \\ &\quad + b_2(P.\text{Friends \#}) + b_3(P.\text{Posts \#}) \end{aligned}$$

We examine the effect on post lifetime of: the existence of a picture, the number of friends and followers, and the number of posts sent by this user. We found that, for both regular and child posts, the existence of a picture affects the post’s lifetime the most. That is, posts with pictures have shorter lifetimes than posts without pictures. Some of the user attributes, such as number of friends or number of posts, also affect post lifetime. We note that the coefficients for these are relatively small, but for users with large numbers of friends or who write large numbers of posts, these factors can have a significant impact on the speed of that users’ posts being censored. However other attributes of a user, such as whether a Weibo user is “verified” by Weibo (*i.e.*, Weibo knows who they are as part of newer Chinese requirements that crack down on pseudonyms unconnected to real world identities) or the number of followers of a user, are not statistically significant factors in a post’s lifetime.

Table 1: Factors affecting post lifetime (regular posts).

Factors	Coef	Stat. Sig.
(Intercept)	7.41	***
Has picture	-4.07×10^{-1}	***
Number of friends	-2.42×10^{-4}	***
Number of posts	-5.23×10^{-5}	***
User verified	-	-
Number of followers	-	-

Table 2: Factors affecting post lifetime (child posts)

Factors	Coef	Stat. Sig.
(Intercept)	6.27	***
Parent has picture	-1.01×10^{-1}	***
Parent friends number	-4.76×10^{-5}	***
Parent posts number	6.84×10^{-6}	***
Parent user verified	2.01×10^{-1}	*
Parent followers number	-	-

4.2 Hypotheses

As a distributed system with 70,000 posts per minute, Weibo has above a 10% rate of deletion in the public timeline (first observed by Bamman *et al.* [2]; we have seen similar behavior). This high deletion rate can be the result of many processes, including anti-spam features, user deletions, as well as anti-censorship features. Within the deletions that we believe are censorship events, we note that 40% of the deletions in our user timeline data set occur within the first hour after a post has appeared. Clearly, Weibo exerts significant controls over its content.

Before censors deal with the sensitive posts which are already in the system, are there filters which do not allow certain posts to enter the Weibo system? This question leads to our first hypothesis.

Hypothesis 1 *Weibo has filtering mechanisms as a proactive, automated defense.*

To find out if there are filtering mechanisms, we attempted to post posts containing sensitive words from the China Digital Times [4] and Tao *et al.* [41]. Here we summarize the filtering mechanisms Weibo was found to apply based on our observations.

- **Explicit filtering:** Weibo will inform a poster that their post cannot be released because of sensitive content.

For example, on 1 August 2012, we tried to post “政法委书记” (Secretary of the Political and Legislative Committee). When we submitted a post with this character string in it, a warning message says “Sorry, since this content violates ‘Sina Weibo regulation rules’ or a related regulation or policy, this operation cannot be processed. If you need help, please contact customer service.”

- **Implicit filtering:** Weibo sometimes suspends posts until they can be manually checked, telling the user that the delay is due to “server data synchronization.”

For example when we submitted the post ‘youshenmefalundebanfa’ on the same day, 1 August 2012, Weibo responded with the message “Your post has been submitted successfully. Currently, there is a delay caused by server data synchronization. Please wait for 1 to 2 minutes. Thank you very much.” This delay, which frequently takes much longer than the 1–2 minutes suggested by Weibo, was triggered by our use of the substring “falun”, pertaining to the Falun Gong religion. In this example, it took more than 5 hours for the post to appear.

- **Camouflaged posts:** Weibo also sometimes makes it appear to a user that their post was successfully posted, but other users are not able to see the post. The poster receives no warning message in this case.

On 1 August 2012 we submitted a post containing the substring “cgc” (Chen Guangcheng [31]), and received no warning messages, so it seemed to be published successfully to our user. When we tried to access that post from another user account, however, we were redirected to Weibo’s error page which claimed the post does not exist.

We found these phenomena to be repeatable. Over the course of our experiments, we selected a number of different subsets of the keyword list published by the China Digital Times [4], trying to post them to Weibo manually. We consistently found all of these same phenomena, although the specific keywords on any list vary over time.

Figure 1 shows that the deletions happen most heavily for a regular post within 5 to 10 minutes of it being posted. While we believe this process to happen largely via automation, it is instructive to estimate how much unaided human labor would otherwise be necessary. Suppose an efficient worker could read 50 posts per minute, including the reposts and figures included in the posts. Then to read Weibo’s full 70,000 new posts [34] in one minute, 1,400 simultaneous workers would be needed. Assuming 8 hour shifts, 4,200 workers would then be required. We can imagine that such a staff would have a high error rate, owing to the repetitive nature of their work. Such a labor force would also be relatively expensive compared to automation. *We instead conclude that Weibo must be using a large amount of automation*, perhaps keyword-based as has been found in other systems in China such as TOM-Skype [16]. This is likely complemented with human efforts to evolve and refine the filtering process.

Some of this refinement certainly results from a centralized list of topics. Other refinement may occur internally, through a smaller number of censors who look for users finding new ways to misspell words or otherwise work around existing filters. Our subsequent hypotheses consider how this refinement occurs and delve into how Weibo’s automation operates.

Hypothesis 2 *Weibo targets specific users, such as those who frequently post sensitive content.*

Another way to achieve prompt response to sensitive posts is to track users who are likely to post sensitive content, using techniques similar to what we are doing. The posts from those sensitive users could then be read by moderators more often and more promptly than the posts of other users.

To test this hypothesis, we plotted Figure 2. We grouped users together who have the same number of censorship events occurring to their posts. The x-axis is the number of such deletions for each cohort of users. The y-axis shows how long these to-be-censored posts live. The clear downward trend is evidence that users with larger deletion frequencies tend to observe faster censorship of their work, supporting our hypothesis.

Even though this figure shows us that the more deletion posts a user has, the faster the users’ posts tend to be deleted, we cannot rule out other features which those users have in common and that those features may lead to the fast deletions. For example, they may tend to use the

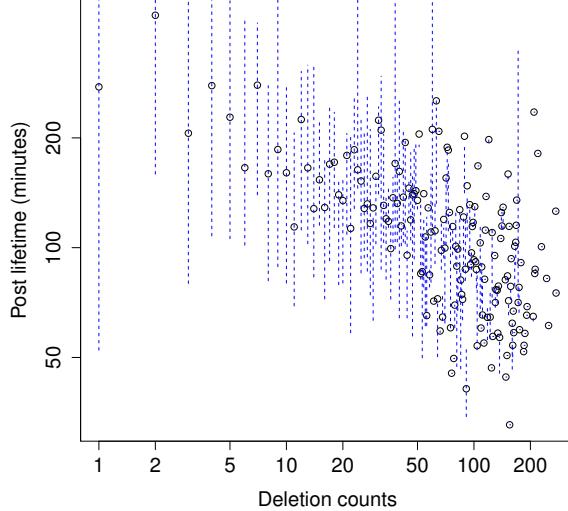


Figure 2: Users’ median post lifetime in minutes vs. the number of deletions for that user on a log-log scale. Black circles show the median lifetime of posts in the cohort, and the dotted blue bars show the 25%–75% range.

same keywords, post from the same geographical area, use the same kind of client platform, and so on. There is a clear correlation between post lifetime and post deletion counts, but correlation does not imply causation.

If the surveillance keyword list and targeting of specific users were the only mechanisms for removing sensitive posts, then the histograms in Figure 1 would stop at a certain time, say 1 or 2 days. However, 10% of the deletions happen after one day, with some deletions occurring one month or more after the post was posted. Clearly, other mechanisms are in use for these long-tail censorship events, which leads to our next hypothesis.

Hypothesis 3 *When a sensitive post is found, a moderator will use automated searching tools to find all of its related reposts (parent, child, etc.), and delete them all at once.*

If this hypothesis is true, then the child posts which repost a censored parent post should all be removed at the same time. To test this hypothesis, we plot the histogram of the standard deviation of the deletion time of the posts sharing the same Repost Identification Number (rpid) in Figure 3. In our system deleted posts dataset, over 82% of reposted posts have a deletion time standard deviation of less than 5 minutes, meaning that a sensitive post is detected and then most of the other posts in a chain of reposts are immediately deleted.

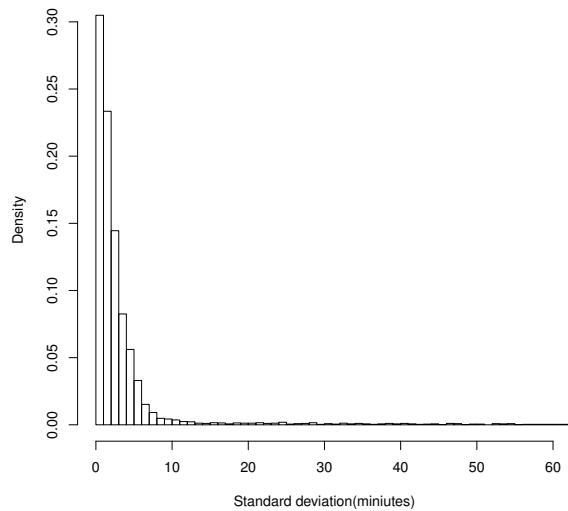


Figure 3: Reposts standard deviation histogram.

There are outliers with standard deviations as high as 5 days which suggest that the mass deletion strategy mentioned here is not the only method Weibo employs to delete sensitive reposts. This leads to our next hypothesis.

Hypothesis 4 *Deletion speed is related to the topic. That is, particular topics are targeted for deletion based on how sensitive they are.*

We performed topical analysis on the deleted posts. The topical analysis methods we use are described in Section 5.1. Here, to save space, we only list the top topic in Table 3. (For further topical discussion, please refer to our technical report [42].) The third column is the response time for the censor to discover a sensitive topic. Specifically, the response time here refers to the period between the time when the first post on this topic appeared in our user timeline data set and the time when the Weibo system starts to delete the posts on this topic heavily. These times were identified through manual analysis. Even when a topic is still being actively censored, it does not necessarily disappear. People may still discuss the topic only to have their posts deleted. That is why some topics appear twice or more in the table. When a topic showed up again, there is no response time for it and we indicate this with a dash (‘-’).

The main five topics extracted by Independent Component Analysis (ICA, see Section 5) are: Qidong, Qian Yunhui, Beijing Rainstorm, Diaoyu Island² and Group Sex. From Table 3, we can see that these topics have a

²Diaoyu Island is the number 3 top topic on 16 August.

Table 3: Blocked topics.

Date	Top 1	Response Time (hours)
7-20	Support Syrian rebels	21.32
7-21	Lying of gov. (Jixian)	12.20
7-22	Beijing rainstorms	2.55
7-23	Beijing rainstorms (Subway)	1.62
7-24	Beijing rainstorms ^a	2.65
7-25	Beijing rainstorms (Fangshan)	2.58
7-27	Beijing rainstorms (37 death)	0.82
7-28	Qidong	1.18
7-29	Qidong (Japanese reporter)	2.25
7-30	Complain gov. (Zhou Jun)	5.73
7-31	Judicial independence	2.00
8-01	Complain gov. (Hongkong)	45.30
8-02	Freedom of speech	7.35
8-03	Qidong (Block the village)	31.58
8-04	One-Child Policy Abuse	33.42
8-05	Human Rights News	24.63
8-07	Qian Yunhui Accident	10.87
8-08	Qian Yunhui Accident	—
8-09	Group sex	0.78
8-10	RTL ^b	3.65
8-11	Tang Hui	33.42
8-12	Group sex	—
8-13	Corpse Plants in Dalian	532.50
8-14	Hongkong	70.98
8-15	Corpse Plants in Dalian	—
8-16	Corpse Plants in Dalian	—
8-17	Complain gov. (North Korea)	19.83
8-18	Zhou Kehua (faked)	16.37

^aRefuse to donate for Beijing rainstorms.

^bRe-education through labor.

relatively short lifetime compared to other topics. These five topics were also hot topics in our public timeline during this period.

This suggests that when sensitive users and a large number of regular Weibo users are discussing the same general topic, *i.e.*, the topic is popular in both the user timeline and public timeline, then extra resources are devoted to finding and deleting such posts³. In Section 5 we will show that the sensitive users in the user timeline combine topics with common themes related to state power (Beijing, government, China, country, police, and people). This suggests that the censors consider the combination of these themes with generally popular topics to warrant extra resources.

³We have not ruled out other possibilities in our study, however, such as that such topics are viewed by many users and therefore more likely to be reported by regular users.

5 Topic extraction

Even though we are following a relatively modest number of Weibo authors, the volume of text we are capturing is still too much to process manually. We need automatic methods to classify the posts that we see, particularly those which are deleted.

Automatic topic extraction is the process of identifying important terms in the text that are representative of the corpus as a whole. Topic extraction was originally proposed by Luhn [19] in 1958. The basic idea is to assign weights to terms and sentences based on their frequency and some other statistical information.

However, when it comes to microblog text, standard language processing tools become inapplicable [18, 40]. Microblogs typically contain short sentences and casual language [7]. Unknown words, such as named entities and neologisms often cause problems with these term-based models. It can be especially challenging to extract topics from Asian languages such as Chinese, Korean, and Japanese, which have no spaces between words.

We applied the Pointillism approach [27] and TF*IDF to extract hot topics. In the Pointillism model, a corpus is divided into n-grams; words and phrases are reconstructed from grams using external information (specifically, temporal correlations in the appearance of grams), giving the context necessary to manage informal uses of the language such as neologisms. Salton’s TF*IDF [10] assigns weights to the terms of a document based on the terms’ relative importance to that document compared to the entire corpus.

We next explain how these techniques work together.

5.1 Algorithm

TF*IDF is a common method to determine the importance of words to a document in a corpus. The TF*IDF value in our case is calculated as:

$$f(t, d_{day}) \times \log \frac{\text{Total number of posts for the month}}{f(t, d_{month})}$$

Here, $f(t, d)$ means the frequency of the term t in document d . We use trigrams as t , and documents d are sets of posts over a certain period of time. d_{day} is the deleted posts we caught on day day . We use the posts of July, 2012 in the public timeline as IDF. $f(t, d_{month})$ is the frequency of term t in the public timeline in July, 2012.

First we calculate TF*IDF scores for all trigrams that have more than 20 occurrences in a day. The top 1000 trigrams with the highest TF*IDF score will be fed to our trigram connection algorithm, hereafter “Connector.” We call these top 1000 trigrams the *1000-TFIDF list*.

To connect trigrams back into longer phrases, Connector finds two trigrams which have two overlapping characters. For instance, if there are ABC and BCD, Connector will connect them to become ABCD. Sometimes there is more than one choice for connecting trigrams, e.g., there could also be BCE and BCF. Sometimes trigrams can even form a loop. To solve these problems, we first build directed graphs for the trigrams with a high TF*IDF score. Each node is a trigram, and edges indicate the overlap information between two trigrams. For example, if ABC and BCD can be connected to make ABCD, then there is an edge from ‘ABC’ to ‘BCD’. After all trigrams are selected, we use DFT (Depth First Traversal) to output the nodes. During the DFT we check to see if a node has been traversed already. If so we do not traverse it again. After the graphs have been traversed, we obtain a set of phrases.

For example, the Connector output of the third most popular topic on 4 August 2012 is:

1. 头骨进京鸣冤。河北广平县上坡村76岁的农民冯虎，其子在19
skull go Beijing to redress an injustice. The son of a 76 year old farmer Fenghu, from Shangpo village, Guangping city, Hebei province, was ... at 19
2. 头骨进京鸣冤。冯出示的头骨赴京鸣...
skull go Beijing to redress an injustice. The skull shown by Feng go Beijing to redress an injustice...
3. 头骨进京鸣冤。冯出示的头骨前额有一大窟窿，他...
skull go Beijing to redress an injustice. There is a big hole on the skull shown by Feng, he...
4. 头骨进京鸣冤。冯出示的头骨前额有一个无罪的公民...
skull go Beijing to redress an injustice. There is a innocent citizen on the skull shown by Feng, he...
5. 头骨进京鸣冤。冯出示的头骨进...
skull go Beijing to redress an injustice. The skull shown by Feng enter...
6. 头骨进京鸣冤。冯出示的头等舱
skull go Beijing to redress an injustice. The first class seat shown by Feng...
7. 【華聯社電】上访15年 老父携儿头骨...
Chinese Community report: petition 15 years, old father bring the skull of his son...

In this example, the 7 outputs of Connector are translated in English, which is written in the next line after the original Chinese phrase. Outputs 4 and 6 are incorrectly connected. This is because the same trigrams are shared by different stories that have high TF*IDF scores on the same day. This problem can be solved by examining the

cosine similarity of the frequency of occurrence of the first and the last trigram for each result.

Cosine similarity is used to judge whether two trigrams have correlated trends.

$$\text{cos.Sim} = \frac{\langle A_i, B_i \rangle}{\sqrt{\sum_{i=1}^n A_i^2} \times \sqrt{\sum_{i=1}^n B_i^2}}$$

where \langle , \rangle denotes an inner product between two vectors. For details, please refer to Song *et al.* [27].

From the connected sentences, listed above, we can begin to understand the general events that are driving major sensitive topics of discussion on Weibo. Table 3 lists the top topics of the deleted posts from 20 July 2012 to 18 August 2012. (A computer failure prevented us from collecting data on 6 August 2012.) Note that we just translated the posts from each topical cluster, we have not confirmed the veracity of any of the claims of the Weibo users’ posts that we translated.

Interestingly, besides named entities, we also extracted three neologisms. They are 李W阳 (Li Wangyang, from 李旺阳), 六圖四 (June Fourth, from 六四), 胡O涛 (Hu Jintao, from 胡锦涛, replacing the middle character with open- and close-parentheses), and 启\东, 启\东 and 启/东 (Qidong, from 启东, inserting punctuation between the two characters). These neologisms became popular enough that they stood out in our TF*IDF analysis.

5.2 Hot sensitive topics

Table 3 tells us the top topic for each day in terms of having the highest TF*IDF scores—however, it does not tell us which topics among these have been discussed for the longest period of time by our users. Also, are there some common themes behind those separate topics?

Here are the top 50 words which have appeared in the *1000-TFIDF list* most frequently from 20 July 2012 to 20 August 2013, manually translated to English:

Beijing City, Liu Futang, secretary, Lujiang County, Guo Jinlong, Qian Yunhui, City Government, Zhou Ke-hua, Red Cross, Diaoyu Island, subprefect, water drain, ordinary people, taxpayer, Fangshan district, Hagens, local police station, office, Beijing, Qidong, government, China, Japan, citizen, county’s head commissioner, reporter, mayor, corrupt official, freedom, country, restrain, keyhole report, wrist watch, police, national, recommend, American, repression, patriotic, democratic, corpses, people, donation, cancel, opinion, reeducation through labor, abolition, truck⁴

We used Independent Component Analysis (ICA) to extract “independent signals” from those most important

⁴For clarity, we have elided close variants on *China*, *Japan*, and *Beijing* from this list.

terms shown above. ICA [14] is a method to separate a linearly mixed signal, x , into mutually independent components, s .

Let $X = [x_1, x_2, \dots, x_m]^T$ be the observation mixture matrix, consisting of m observed signals x_i . Since X is the linear composition of the independent components, s , X can be modeled as:

$$X = AS = \sum_{i=1}^m a_i s_i$$

A , the mixing matrix, gives the coefficients for linear combinations of the independent signals, the rows of S .

Here, each word is represented by a row vector of length 864 (36×24), which contains the 36 days worth of hourly frequency from 22 July 2012 to 2 September 2012. The 50×864 matrix X is fed to an ICA program [25]. The number of independent components number is set to 5, which retains almost 100% of the eigenvalues.

There are six words that appear in almost every independent signal: Beijing, government, China, country, policeman, and people. This means that the sensitive user group in our user timeline has these general themes that cut across the many individual topics that they discuss, which may explain why their posts are often subject to censorship.

6 Discussion

Weibo appears to have a variety of other mechanisms that do not fit neatly into our hypotheses, but which are interesting to discuss. We first consider other aspects of Weibo’s filtering, then we look at diurnal (time-of-day) censorship behaviors, and finally we try to synthesize some of our observations.

6.1 Weibo’s filtering mechanisms

Sina Weibo has a complex variety of censorship mechanisms, including both proactive and retroactive mechanisms. Here we summarize the mechanisms Weibo may apply. Proactive mechanisms, as we discussed in Hypothesis 1, may include: explicit filtering, implicit filtering, and camouflaged posts. Retroactive mechanisms for removing content that has already been released may include:

- **Backwards reposts search:** In our deleted posts dataset over 82% of reposted posts have a standard deviation of less than 5 minutes for deletion time, meaning that a sensitive post is detected and then most of the other posts in a chain of reposts are then deleted (Hypothesis 3).

- **Backwards keyword search:** We also observed that Weibo sometimes removes posts retroactively in a way that causes spikes in the deletion rate of a particular keyword within a short amount of time.

Here, we give two examples (天朝 and 37人), out of many that we witnessed, with a strong spike in the deletion of posts containing that keyword.

We first consider 天朝, Tian Chao, a neologism for “Celestial Empire” where 天 is an alternate form for 天; the substitute character is visually similar to the original and also appears to be constructed from the two distinct characters 王 八, meaning “bastard.”). The frequency of 天朝 in deleted posts, day by day, is the sequence (6,3,0,0,2,2,0,3,0,2,3,3,2,1,2,0,0,1,0,0,0,5,4,4,2,14,3,6,4) respectively from 28 July 2012 to 25 August 2012. There is a concentrated deletion (14 censorship events) of posts with this word within several minutes on 22 August 2012, impacting posts that were several weeks old at the time. It is likely that a censor discovered this new phrase and ordered it globally expunged.

Another example is the keyword 37人 (37 people). There are 44 posts containing this keyword, which were created from 2 days to 5 days before the censorship event, all removed together within 5 minutes (03:25 to 03:30 27 July 2012). Those 44 posts are from different users, have no common parent posts, and have no common pictures. The only plausible explanation for this concentrated deletion would appear to be a keyword-based deletion. The deletion time at 3:25am Beijing time also strongly suggests that there are moderators working in the early morning. To understand this workforce and its distributed nature, we perform further analysis in Section 6.2.

- **Monitoring specific users:** Hypothesis 2 shows a clear preference for Weibo’s censors to pay more attention to users who seemingly like to discuss censored topics.
- **Account closures:** Weibo also closes users’ accounts. There were over 300 user accounts closed by the system from our sensitive user group (out of over 3,500 users) during the roughly two month period while when we collected data for their user timelines.
- **Search filtering:** To prevent users from finding sensitive information on weibo.com, Weibo also has a frequently updated list of words [4] which cannot be searched.
- **Public timeline filtering:** We believe that sensitive topics are filtered out of the public timeline. This

filtering appears to be limited to only general topics that have been known to be sensitive for a relatively long time. In this paper all major results are based on the user timeline, we only use the public timeline for general results about major trending topics in Weibo.

- **User credit points:** In May 2012, Sina Weibo announced a “user credit” points system [22] through which users can report sensitive or rumor-based posts to the administrators. We do not know the extent to which the point system interacts with the censorship mechanisms that we have already described. It is possible that these reports “bubble up” and help Weibo tune its automated filters, but we have no way to observe this.

6.2 Time-of-day behavior

In our data, the time at which the censors are working and deleting posts correlates more with the usage patterns of regular users than with a typical day-time work schedule (*e.g.*, 8am to 5pm Beijing time). Figure 4 shows the total hourly deletions for different kinds of posts (on a log scale) from 20 July to 8 September 2012. Both “general deletions” and “system deletions” happen even very late at night.

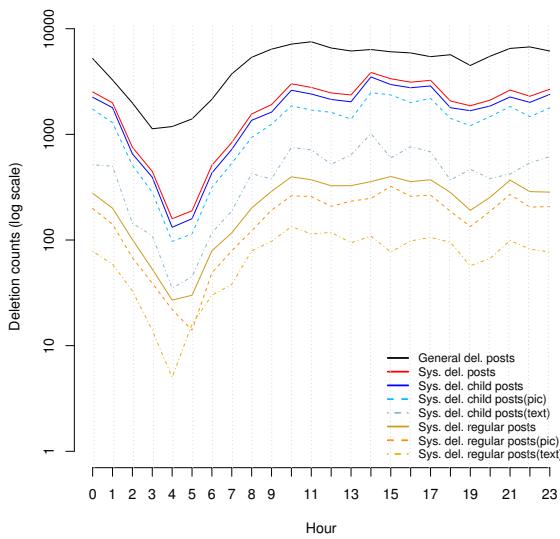


Figure 4: Post deletion amounts over 24 hours.

So do the censors respond as quickly during the night as during day hours? We plotted the median lifetime of the posts as a function of their deletion time in Figure 5. The morning-hour spike suggests that the censors are behind in the morning, both catching up on overnight posts

and dealing with a fresh influx of posts from morning risers. They catch up by late morning or early afternoon.

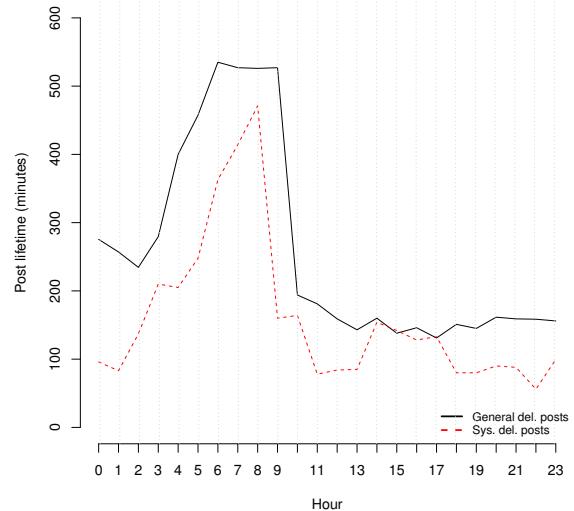


Figure 5: Post lifetime vs. deletion time of the day.

From Figure 4 and Figure 5 it is clear that, while a significant fraction of the censors seem to work during regular work hours, many do not.

6.3 Synthesis

Based on everything we have seen and observed, we can begin to understand how Weibo censorship works. Clearly, they are using a strong degree of automation to help them delete posts that have been declared sensitive. It is also clear that this process is relatively “loose,” in the sense that there are few sharp rules that define what gets deleted *vs.* what is allowed to remain. Given the long-tailed distribution that we observe in post lifetimes prior to censorship, it is clear that some posts are not considered a high priority for censorship, such as if two friends start conversing with each other using a new neologism, euphemism, or other coinage that would otherwise be censorship-worthy. However, when those new terms spread and grow, they are censored both proactively and retroactively.

This suggests that Weibo is trying to strike a balance between satisfying the legal requirements within which it operates and the costs of running a fine-grained instrument of political censorship. Weibo must conduct *just enough* censorship to satisfy government regulations without being so intrusive as to discourage users from using their service. Among other issues, they must surely be deeply concerned with false positives. If truly innocu-

ous posts disappeared with any regularity, Weibo’s users might defect to a competing service.

It is unclear the extent to which Weibo is using natural language processing (NLP) algorithms to aid in their work, versus having a stable of censors watching for things to go viral and then using search tools to stamp them out. Certainly, our use of fairly simple NLP techniques helped reduce the workload of analyzing trending topics, so comparable techniques may well be in use by Weibo. NLP techniques in a censor’s hands can be thought of as a “force multiplier,” but it is unclear whether they fundamentally change the game. Consider, with English-language spam emails, the degree to which spammers will try to evade automated spam classification systems. These techniques and more could well be applied to automated or manual rewriting of postings, with the intent of avoiding automated censorship. The results might not be as easy to read, but humans will likely have an advantage at reading jumbled text, at least until NLP algorithms are extended to deal with them. Conversely, NLP techniques can cluster together related terms, assisting censors to overcome such techniques. At least so far, we have not seen evidence of any sort of arms race between increasingly sophisticated ways to avoid censorship and increasingly powerful censorship techniques.

In many ways, Internet censorship is related to intrusion detection. When our results in this paper are compared to related work (see Section 1), including both IP-layer filtering and application-level censorship, a picture of Internet censorship in China emerges where “defense-in-depth” is taken to a new level. Intrusion detection research has long focused on issues such as false positive *vs.* false negative tradeoffs, viral spreading patterns, polymorphic content, and the distinction between different layers of abstraction (such as IP packets *vs.* application-layer data). The so-called “Great Firewall of China” and the accompanying application-layer censorship that China’s domestic web services, such as Weibo, carry out afford us an opportunity to study a real, national scale intrusion detection system.

6.4 Major caveats

The most important caveat to keep in mind when interpreting our results is that we collected posts from a very specific core set of users, built up from a “seed” group of users who post about sensitive topics, which we call the “user timeline.” Unless otherwise noted, such as when results are from the public timeline, all results in this paper are from the user timeline and therefore might be biased by the differences between this core set of users and the average Weibo user. All deletion rates, deletion times, *etc.* must be interpreted in this light. In other

words, our sample users should not be considered to be representative of the general population of Weibo.

Another important caveat is that our system does not detect post deletions in the user timeline if the post deleted is not one of the 50 most recent posts by the user (see Section 3). This may affect our results about the distribution of post deletions over time in Section 4.

7 Conclusion

Our research found that deletions happen most heavily in the first hour after a post has been made (see Figure 1). Especially for original posts that are not reposts, most deletions occur within 30 minutes, accounting for 30% of the total deletions of such posts. Nearly 90% of the deletions of such posts happen within the first 24 hours of the post.

With respect to the hypotheses enumerated in Section 4, we make the following conclusions:

- Hypothesis 1: The Weibo system keeps more than one keyword list, where each list triggers a different kind of censorship behavior.
- Hypothesis 2: The clear downward trend in Figure 2 could be evidence that certain users are flagged for closer scrutiny, but we have not ruled out other causes in this paper.
- Hypothesis 3: Figure 3 shows that over 82% of reposted posts have a standard deviation of less than 5 minutes deletion time, meaning that a sensitive post is detected and then most of the other posts in a chain of reposts are then deleted.
- Hypothesis 4: As described in Section 4, using the methods described in Section 5 we find that topics that were trends in the user timeline and were also, according to the public timeline, hot topics in public discussion as a whole about events that happened during our month of data collection (Qidong, Qian Yunhui, Beijing Rainstorms, Diaoyu islands, and a group sex scandal) had very short lifetimes. Recall that the deleted posts in the user timeline included themes related to state power (Beijing, government, China, country, policeman, and people). This suggests that such broadly discussed topics are targeted with more censorship resources to limit certain kinds of discussion about the events.

Future work may reveal many mechanisms beyond those we described here, and many different strategies that Weibo uses to prioritize what content to delete. Our

results suggest that Weibo employs a distributed, heterogeneous strategy for censorship that has a great amount of “defense-in-depth.”

One aspect of censorship that is not considered in our analysis, but would be an interesting topic for future work, is the interactions between social media and traditional media. Leskovec *et al.* [17] gives an interesting analysis of the interplay between blogs and traditional media during the 2008 U.S. Presidential election. Traditional media relevant to Weibo may include the state-run media that is heavily censored, or off-shore news outlets that are uncensored but limited in availability and sometimes offset from China’s news cycles by timezone differences.

8 Acknowledgments

We would like to thank the anonymous reviewers and our shepherd, Nikita Borisov, for helpful feedback. We owe our deepest gratitude to Professors Stephanie Forrest, Christopher Bronk, and George Luger for their feedback and comments, and for encouraging us to go forward. We are also grateful to Ben Edwards for insightful discussions about potential future work. This material is based upon work supported by the National Science Foundation under Grant Nos. #0844880, #0905177, #1017602. Jed Crandall is also supported by the Defense Advanced Research Projects Agency CRASH program under grant #P-1070-113237.

References

- [1] APACHE SOFTWARE FOUNDATION. *Apache HBase*. <http://hbase.apache.org/>.
- [2] BAMMAN, D., O’CONNOR, B., AND SMITH, N. Censorship and deletion practices in Chinese social media. *First Monday* 17, 3-5 (March 2012).
- [3] CAO, B. Sina’s Weibo outlook buoys Internet stock gains: China overnight. Bloomberg, 28 February 2012. <http://www.bloomberg.com/news/2012-02-28/sina-s-weibo-outlook-buoys-internet-stock-gains-in-n-y-china-overnight.html>.
- [4] CHINA DIGITAL TIMES. 新浪微博搜索敏感词列表. <http://chinadigitaltimes.net/space/%E6%96%B0%E6%B5%AA%E5%BE%AE%E5%8D%9A%E6%90%9C%E7%B4%A2%E6%95%8F%E6%84%9F%E8%AF%8D>.
- [5] CLAYTON, R., MURDOCH, S. J., AND WATSON, R. N. M. Ignoring the Great Firewall of China. In *6th Workshop on Privacy Enhancing Technologies* (Cambridge, United Kingdom, June 2006).
- [6] CRANDALL, J. R., ZINN, D., BYRD, M., BARR, E., AND EAST, R. ConceptDoppler: a weather tracker for internet censorship. In *Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS 2007)* (Alexandria, Virginia, Oct. 2007), ACM, pp. 352–365.
- [7] ELLEN, J. All about microtext - a working definition and a survey of current microtext research within artificial intelligence and natural language processing. In *3rd International Conference on Agents and Artificial Intelligence, ICAART 2011* (Jan. 2011).
- [8] ESPINOZA, A. M., AND CRANDALL, J. R. Work-in-progress: Automated named entity extraction for tracking censorship of current events. In *The Proceedings of the USENIX Workshop on Free and Open Communications on the Internet. (FOCI 2011)* (Aug. 2011).
- [9] FU, K.-W., CHAN, C.-H., AND CHAU, M. Assessing censorship on microblogs in China: Discriminatory keyword analysis and the real-name registration policy. *IEEE Internet Computing* (2013), 42–50.
- [10] GERARD, S., AND CHRISTOPHER, B. Term-weighting approaches in automatic text retrieval. *Inf. Process. Manage.* 24, 5 (Aug. 1988), 513–523.
- [11] THE GLOBAL INTERNET FREEDOM CONSORTIUM. *The Great Firewall Revealed*, Dec. 2002. <http://www.internetfreedom.org/files/WhitePaper/ChinaGreatFirewallRevealed.pdf>.
- [12] GOH, C.-L. *Unknown Word Identification for Chinese: Morphological Analysis*. PhD thesis, Nara Institute of Science and Technology, 2006.
- [13] HUMAN RIGHTS WATCH. *Race to the Bottom: Corporate Complicity in Chinese Internet Censorship*, Aug. 2006. <http://www.unhcr.org/refworld/docid/45cb138f2.html>.
- [14] HYVARINEN, A., AND OJA, E. Independent component analysis: algorithms and applications. *Neural Networks* 13 (2000), 411–430.
- [15] KING, G., PAN, J., AND ROBERTS, M. E. How censorship in China allows government criticism but silences collective expression. *American Political Science Review* 107 (2013), 1–18.
- [16] KNOCKEL, J., CRANDALL, J. R., , AND SAIA, J. Three researchers, five conjectures: An empirical analysis of TOM-Skype censorship and surveillance, Aug. 2011.
- [17] LESKOVEC, J., BACKSTROM, L., AND KLEINBERG, J. Memetracking and the dynamics of the news cycle. In *Proceedings of the 15th ACM International Conference on Knowledge Discovery and Data Mining (KDD ’99)* (Paris, France, 2009), pp. 497–506.
- [18] LI, J., LIU, Z., FU, Y., AND SHE, L. Chinese hot topic extraction based on web log. In *Proceedings of the 2009 International Conference on Web Information Systems and Mining (WISM ’09)* (Nov. 2009), pp. 103–107.
- [19] LUHN, H. P. The automatic creation of literature abstracts. *IBM J. Res. Dev.* 2, 2 (Apr. 1958), 159–165.
- [20] MACKINNON, R. China’s censorship 2.0: How companies censor bloggers. *First Monday* 14, 2 (Feb. 2009).
- [21] MAGISTAD, M. K. How weibo is changing china. YALEGlobal Online, Aug. 2012. <http://yaleglobal.yale.edu/content/how-weibo-changing-china>.
- [22] MERIWETHER, A. A user contract for chinese microbloggers. *Herdict Blog* (June 2012).
- [23] PARK, J. C., AND CRANDALL, J. R. Empirical study of a national-scale distributed intrusion detection system: Backbone-level filtering of HTML responses in China. In *Proceedings of the 30th International Conference on Distributed Computing Systems (ICDCS 2010)* (Genoa, Italy, June 2010).
- [24] ROGER, D., NICK, M., AND PAUL, S. Tor: the second-generation onion router. In *USENIX Security Symposium* (San Diego, CA, 2004).
- [25] SHEN, H., HÜPER, K., AND SEGHOUANE, A.-K. Geometric optimisation and FastICA algorithms. In *Proceedings of the 17th International Symposium of Mathematical Theory of Networks and Systems (MTNS 2006)* (Kyoto, Japan, 2006), pp. 1412–1418.
- [26] SINA. *Sina Weibo API*. <http://open.weibo.com/wiki/API%E6%96%87%E6%A1%A3/en>.

- [27] SONG, P., SHU, A., ZHOU, A., WALLACH, D. S., AND CRANDALL, J. R. A pointillism approach for natural language processing of social media. In *IEEE International Conference on Natural Language Processing and Knowledge Engineering* (2012).
- [28] TAO. *China: Journey to the heart of Internet Censorship*. Reporters Without Borders and Chinese Human Rights Defenders, Oct. 2007. http://www.rsf.org/IMG/pdf/Voyage_au_coeur_de_la_censure_GB.pdf.
- [29] VILLENEUVE, N. *BREACHING TRUST: An analysis of surveillance and security practices on China's TOM-Skype platform*. Citizen Lab, Munk Centre for International Studies, University of Toronto, Oct. 2008. <http://www.nartv.org/mirror/breachingtrust.pdf>.
- [30] WANG, N. Control of Internet search engines in China – a study on Google and Baidu. Master’s thesis, Unitec New Zealand, Aug. 2008. <http://unitec.researchbank.ac.nz/bitstream/handle/10652/1272/fulltext.pdf?sequence=1>.
- [31] WIKIPEDIA. Chen Guangcheng. http://en.wikipedia.org/wiki/Chen_Guangcheng.
- [32] WIKIPEDIA. Deng Yujiao incident. http://en.wikipedia.org/wiki/Deng_Yujiao_incident.
- [33] WIKIPEDIA. Protests of Wukan. http://en.wikipedia.org/wiki/Protests_of_Wukan.
- [34] WIKIPEDIA. Sina Weibo. en.wikipedia.org/wiki/Sina_Weibo.
- [35] WIKIPEDIA. Yao Jiaxin murder case. http://en.wikipedia.org/wiki/Yao_Jiaxin_murder_case.
- [36] WIKIPEDIA. Shifang protest, 2012. http://en.wikipedia.org/wiki/Shifang_protest.
- [37] WOLFGARTEN, S. *Investigating large-scale Internet content filtering*. Dublin City University, Ireland, Aug. 2006. <http://www.security-science.com/mastering-internet-security/internet-security-ebooks-and-documents/item/investigating-large-scale-internet-content-filtering>.
- [38] YANG, L. Weibo’s impact on China’s society. Journalism Research Paper, May 2011. <http://eportfolios.ithaca.edu/lyang1/docs/jresearch/weibo/>.
- [39] YE, S. Sina Weibo controls the “holy shit idea of a generation,” launches new URL Weibo.com. <http://techrice.com/2011/04/07/sina-weibo-controls-the-holy-shit-idea-of-a-generation-launches-new-url-weibo-com/>.
- [40] ZHAO, X., JIN, P., AND YUE, L. Analysis of long queries in a large scale search log. In *Future Generation Communication and Networking Symposia (FGCNS ’08)* (Dec. 2008), pp. 39–42.
- [41] ZHU, T., BRONK, C., AND WALLACH, D. S. An analysis of Chinese search engine filtering. *CoRR abs/1107.3794* (2011). <http://arxiv.org/abs/1107.3794>.
- [42] ZHU, T., PHIPPS, D., PRIDGEN, A., CRANDALL, J. R., AND WALLACH, D. S. Tracking and quantifying censorship on a Chinese microblogging site. *CoRR abs/1211.6166* (2012). <http://arxiv.org/abs/1211.6166>.
- [43] ZITTRAIN, J., AND EDELMAN, B. Internet filtering in China. *IEEE Internet Computing* 7 (Mar. 2003), 70–77.

You are How You Click: Clickstream Analysis for Sybil Detection

Gang Wang, Tristan Konolige, Christo Wilson[†], Xiao Wang[‡],

Haitao Zheng and Ben Y. Zhao

UC Santa Barbara

[†]*Northeastern University*

[‡]*Renren Inc.*

{gangw, tkonolige, htzheng, ravenben}@cs.ucsb.edu, cbw@ccs.neu.edu, xiao.wang@renren-inc.com

Abstract

Fake identities and Sybil accounts are pervasive in today’s online communities. They are responsible for a growing number of threats, including fake product reviews, malware and spam on social networks, and astroturf political campaigns. Unfortunately, studies show that existing tools such as CAPTCHAs and graph-based Sybil detectors have not proven to be effective defenses.

In this paper, we describe our work on building a practical system for detecting fake identities using server-side clickstream models. We develop a detection approach that groups “similar” user clickstreams into behavioral clusters, by partitioning a similarity graph that captures distances between clickstream sequences. We validate our clickstream models using ground-truth traces of 16,000 real and Sybil users from Renren, a large Chinese social network with 220M users. We propose a practical detection system based on these models, and show that it provides very high detection accuracy on our clickstream traces. Finally, we worked with collaborators at Renren and LinkedIn to test our prototype on their server-side data. Following positive results, both companies have expressed strong interest in further experimentation and possible internal deployment.

1 Introduction

It is easier than ever to create fake identities and user accounts in today’s online communities. Despite increasing efforts from providers, existing services cannot prevent malicious entities from creating large numbers of fake accounts or Sybils [9]. Current defense mechanisms are largely ineffective. Online Turing tests such as CAPTCHAs are routinely solved by dedicated workers for pennies per request [22], and even complex human-based tasks can be overcome by a growing community of malicious crowdsourcing services [23, 39]. The result of this trend is a dramatic rise in forged and malicious

online content such as fake reviews on Yelp [35], malware and spam on social networks [10, 11, 32], and large, Sybil-based political lobbying efforts [27].

Recent work has explored a number of potential solutions to this problem. Most proposals focus on detecting Sybils in social networks by leveraging the assumption that Sybils will find it difficult to befriend real users. This forces Sybils to connect to each other and form strongly connected subgraphs [36] that can be detected using graph theoretic approaches [8, 34, 45, 46]. However, the efficacy of these approaches in practice is unclear. While some Sybil communities have been located in the Spanish Tuenti network [7], another study on the Chinese Renren network shows the large majority of Sybils actively and successfully integrating themselves into real user communities [43].

In this paper, we describe a new approach to Sybil detection rooted in the fundamental behavioral patterns that separate real and Sybil users. Specifically, we propose the use of *clickstream models* as a tool to detect fake identities in online services such as social networks. Clickstreams are traces of click-through events generated by online users during each web browsing “session,” and have been used in the past to model web traffic and user browsing patterns [12, 20, 24, 28]. Intuitively, Sybils and real users have very different goals in their usage of online services: where real users likely partake of numerous features in the system, Sybils focus on specific actions (*i.e.* acquiring friends and disseminating spam) while trying to maximize utility per time spent. We hypothesize that these differences will manifest as significantly different (and distinctive) patterns in clickstreams, making them effective tools for “profiling” user behavior. In our context, we use these profiles to distinguish between real and Sybil users.

Our work focuses on building a practical model for accurate detection of Sybils in social networks. We develop several models that encode distinct event sequences and inter-event gaps in clickstreams. We build weighted

graphs of these sequences that capture pairwise “similarity distance” between clickstreams, and apply clustering to identify groups of user behavior patterns. We validate our models using ground-truth clickstream traces from 16,000 real and Sybil users from Renren, a large Chinese social network with 220M users. Using our methodology, we build a detection system that requires little or no knowledge of ground-truth. Finally, we validate the usability of our system by running initial prototypes on internal datasets at Renren and LinkedIn.

The key contributions of this paper are as follows:

- To the best of our knowledge, we are the first to analyze click patterns of Sybils and real users on social networks. By analyzing detailed clickstream logs from a large social network provider, we gain new insights on activity patterns of Sybils and normal users.
- We propose and evaluate several clickstream models to characterize user clicks patterns. Specially, we map clickstreams to a similarity graph, where clickstreams (vertices) are connected using weighted edges that capture pairwise similarity. We apply graph partitioning to identify clusters that represent specific click patterns. Experiments show that our model can efficiently distinguish between clickstreams of Sybil and normal users.
- We develop a practical Sybil detection system based on our clickstream model, requiring minimal input from the service provider. Experiments using ground-truth data show that our system generates <1% false positives and <4% false negatives.
- Working closely with industrial collaborators, we have deployed prototypes of our system at Renren and LinkedIn. Security teams at both companies have run our system on real user data and received very positive results. While corporate privacy policies limit the feedback visible to us, both companies have expressed strong interest in further experimentation and possible deployment of our system.

To the best of our knowledge, we are the first to study clickstream models as a way to detect fake accounts in online social networks. Moving forward, we believe clickstream models are a valuable tool that can complement existing techniques, by not only detecting well-disguised Sybil accounts, but also reducing the activity level of any remaining Sybils to that of normal users.

Roadmap. We begin in Section 2 by describing the problem context and our ground-truth dataset, followed by preliminary analysis results in Section 3. Next, in Section 4 we propose our clickstream models to effectively distinguish Sybil with normal users. Then in Section 5, we develop an incremental Sybil detector that can scale with today’s large social networks. We then extend this detector in Section 6 by proposing an unsupervised Sybil

Dataset	Users	Clicks	Date (2011)	Sessions
Sybil	9,994	1,008,031	Feb.28-Apr.30	113,595
Normal	5,998	5,856,941	Mar.31-Apr.30	467,179

Table 1: Clickstream dataset.

detector, where only a minimal (and fixed) amount of ground-truth is needed. Finally, in Section 7, we describe experimental experience of testing our prototype code in real-world social networks (Renren and LinkedIn). We then discuss related work in Section 8 and conclude in Section 9.

2 Background

In this section, we provide background for our study. First, we briefly introduce the Renren social network and the malicious Sybils that attack it. Second, we describe the key concepts of user clickstreams, as well as the ground-truth dataset we use in our study.

Renren. Renren is the oldest and largest Online Social Network (OSN) in China, with more than 220 million users [17]. Renren offers similar features and functionalities as Facebook: users maintain personal profiles and establish social connections with their friends. Renren users can update their status, write blogs, upload photos and video, and share URLs to content on and off Renren. When a user logs-in to Renren, the first page they see is a “news-feed” of their friends’ recent activities.

Sybils. Like other popular OSNs, Renren is targeted by malicious parties looking to distribute spam and steal personal information. As in prior work, we refer to the fake accounts involved in these attacks as Sybils [43]. Our goal is to detect and deter these malicious Sybils; our goal is not to identify benign fakes, *e.g.* pseudonymous accounts used by people to preserve their privacy.

Prior studies show that attackers try to friend normal users using Sybil accounts [43]. On Renren, Sybils usually have complete, realistic profiles and use attractive profile pictures to entice normal users. It is challenging to identify these Sybils using existing techniques because their profiles are well maintained, and they integrate seamlessly into the social graph structure.

Clickstream Data. In this paper, we investigate the feasibility of using *clickstreams* to detect Sybils. A clickstream is the sequence of HTTP requests made by a user to a website. Most requests correspond to a user explicitly fetching a page by clicking a link, although some requests may be programmatically generated (*e.g.* XMLHttpRequest). In our work, we assume that a clickstream can be unambiguously attributed to a specific user account, *e.g.* by examining the HTTP request cookies.

Our study is based on detailed clickstreams for 9994

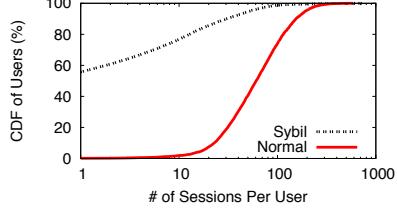


Figure 1: # of sessions per user.

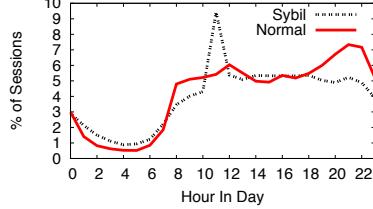


Figure 2: Sessions through the day.

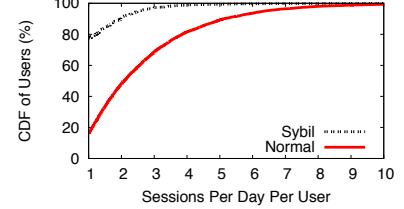


Figure 3: Sessions per day per user.

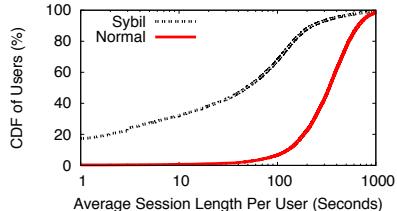


Figure 4: Average session length per user.

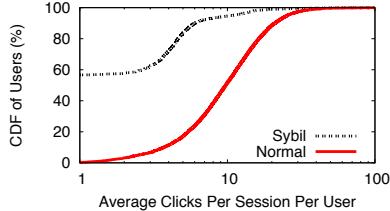


Figure 5: Average # of clicks per session per user.

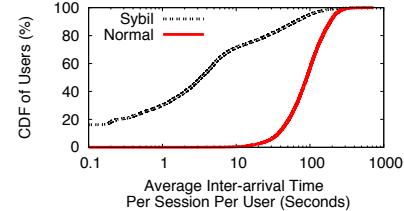


Figure 6: Average time interval between clicks per session per user.

Sybils and 5998 normal users on Renren. Sybil clickstreams were selected at random from the population of malicious accounts that were banned by Renren in March and April 2011. Accounts could be banned for abusive behaviors such as spamming, harvesting user data or sending massive numbers of friend requests. Normal user clickstreams were selected uniformly at random from Renren user population in April 2011, and were manually verified by Renren’s security team.

The dataset summary is shown in Table 1. In total, our dataset includes 1,008,031 and 5,856,941 clicks for Sybils and normal users, respectively. Each click is characterized by a timestamp, an anonymized userID, and an *activity*. The activity is derived from the request URL, and describes the action the user is undertaking. For example, the “friend request” activity corresponds to a user sending a friend request to another user. We discuss the different *categories* of activities in detail in Section 3.2.

Each user’s clickstream can be divided into *sessions*, where a session represents the sequence of a user’s clicks during a single visit to Renren. Unfortunately, users do not always explicitly end their session by logging out of Renren. As in prior work, we assume that a user’s session is over if they do not make any requests for 20 minutes [6]. Session duration is calculated as the time interval between the first and last click within a session. Overall, our traces contain 113,595 sessions for Sybils and 467,179 sessions for normal users.

3 Preliminary Clickstream Analysis

We begin the analysis of our data by looking at the high-level characteristics of Sybil and normal users on Ren-

ren. Our goals are to provide an overview of the dataset, and to motivate the use of clickstreams as a rich data source for uncovering malicious behavior. Towards these ends, we analyze our data in four ways: *first*, we examine session-level characteristics. *Second*, we analyze the activities users engage in during each session. *Third*, we construct a state-based Markov Chain model to characterize the transitions between clicks during sessions. Finally, we use a Support Vector Machine (SVM) approach to learn the important features that distinguish Sybil and normal user clickstreams.

3.1 Session-level Characteristics

In this section, we seek to determine the session-level differences between normal and Sybil accounts in our dataset. First, we examine the total number of sessions from each user. As shown in Figure 1, >50% of Sybils have only a single session; far fewer than normal users. It is likely that these Sybils sent spam during this single session and were banned shortly thereafter. A small portion of Sybils are very active and have >100 sessions.

Next, we examine when Sybils and normal users are active each day. Figure 2 shows that all users exhibit a clear diurnal pattern, with most sessions beginning during daytime. This indicates that at least a significant portion of Sybils in our dataset could be controlled by real people exhibiting normal behavioral patterns.

Next, we investigate the number of sessions per user per day. Figure 3 shows that 80% of Sybils only login to Renren once per day or less, versus 20% of normal users. The duration of Sybil sessions is also much shorter, as shown in Figure 4: 70% of Sybil session are <100 seconds long, versus 10% of normal sessions. The vast ma-

jority of normal sessions last several minutes.

Figure 5 shows the number of clicks per session per user. Almost 60% of Sybil sessions only contain one click, whereas 60% of normal user sessions have ≥ 10 clicks. Not only do Sybil sessions tend to be shorter, but Sybils also click much faster than normal users. As shown in Figure 6, the average inter-arrival time between Sybil clicks is an order of magnitude shorter than for normal clicks. This indicates that Sybils do not linger on pages, and some of their activities may be automated.

The observed session-level Sybil characteristics are driven by attacker’s attempts to circumvent Renren’s security features. Renren limits the number of actions each account can take, *e.g.* 50 friend requests per day, and 100 profiles browsed per hour. Thus, in order to maximize efficiency, attackers create many Sybils, quickly login to each one and perform malicious activities (*e.g.* sending unsolicited friend requests and spam), then logout and move to the next Sybil. As shown in Table 2, Sybils spend a great deal of clicks sending friend requests and browsing profiles, despite Renren’s security restrictions.

3.2 Clicks and Activities

Having characterized the session-level characteristics of our data, we now analyze the type and frequency clicks within each session. As shown in Table 2, we organize clicks into *categories* that correspond to high-level OSN features. Within each category there are *activities* that map to particular Renren features. In total, we observe 55 activities that can be grouped into 8 primary categories. These categories are:

- **Friending:** Includes sending friend requests, accepting or denying those requests, and un-friending.
- **Photo:** Includes uploading photos, organizing albums, tagging friends, browsing friend’s photos, and writing comments on photos.
- **Profile:** This category encompasses browsing user profiles. Like Facebook, profiles on Renren can be browsed by anyone, but the information that is displayed is restricted by the owner’s privacy settings.
- **Share:** Refers to users posting hyperlinks on their wall. Common examples include links to videos and news stories on external websites, or links to blog posts and photo albums on Renren.
- **Message:** Includes status updates, wall posts, and real-time instant-messages (IM).
- **Blog:** Encompasses writing blogs, browsing blog articles, and leaving comments on blogs.
- **Notification:** Refers to clicks on Renren’s notification mechanism that alerts users to comments or likes on their content.

Category	Description	Sybil Clks # (K)	Sybil Clks %	Nrml Clks # (K)	Nrml Clks %
Friending	Send request	417	41	16	0
	Accept invitation	20	2	13	0
	Invite from guide	16	2	0	0
Photo	Visit photo	242	24	4,432	76
	Visit album	25	2	330	6
Profile	Visit profiles	160	16	214	4
Share	Share content	27	3	258	4
Message	Send IM	20	2	99	2
Blog	Visit/reply blog	12	1	103	2
Notification	Check notification	8	1	136	2

Table 2: Clicks from normal users and Sybils on different Renren activities. # of clicks are presented in thousands. Activities with <1% of clicks are omitted for brevity.

- **Like:** Corresponds to the user liking (or unliking) content on Renren.

Table 2 displays the most popular activities on Renren. The number of clicks on each activity is shown (in thousands), as well as the percent of clicks. Percentages are calculated for Sybils and normal users separately, *i.e.* each “%” column sums to 100%. For the sake of brevity, only activities with $\geq 1\%$ of clicks for either Sybils or normal users are shown. The “Like” category has no activity with $\geq 1\%$ of clicks, and is omitted from the table.

Table 2 reveals contrasting behavior between Sybils and normal users. Unsurprisingly, normal users’ clicks are heavily skewed toward viewing photos (76%), albums (6%), and sharing (4%). In contrast, Sybils expend most of their clicks sending friend requests (41%), viewing photos (24%), and browsing profiles (16%). The photo browsing and profile viewing behavior are related: these Sybils crawl Renren and download users’ personal information, including profile photos.

Sybils’ clicks are heavily skewed toward friending (41% for Sybils, 0.3% for normal users). This behavior supports one particular attack strategy on Renren: friending normal users and then spamming them. However, given that other attacks are possible (*e.g.* manipulating trending topics [16], passively collecting friends [32]), we cannot rely on this feature alone to identify Sybils.

Normal users and Sybils share content (4% and 3%, respectively) as well as send messages (2% and 2%) at similar rates. This is an important observation, because sharing and messaging are the primary channels for spam dissemination on Renren. The similar rates of legitimate and illegitimate sharing/messaging indicate that spam detection systems cannot simply leverage numeric thresholds to detect spam content.

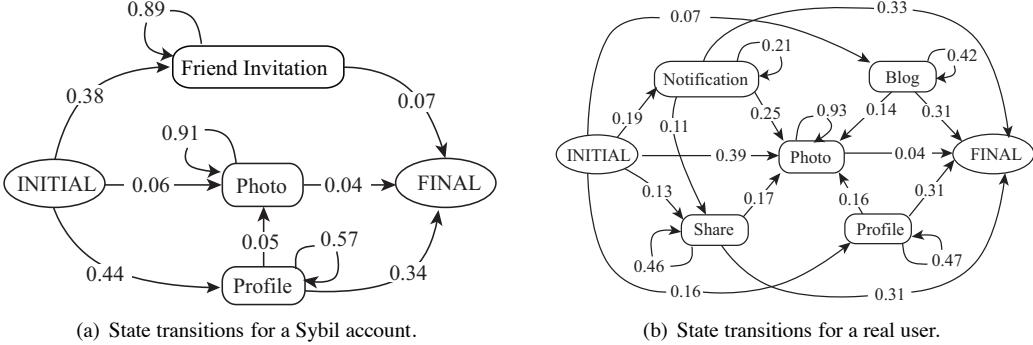


Figure 7: Categories and transition probabilities in the clickstream models of Sybils and normal users.

3.3 Click Transitions

Sections 3.1 and 3.2 highlight some of the differences between Sybils and normal users. Next, we examine differences in click ordering, *i.e.* how likely is it for a user to transition from activity A to activity B during a single session?

We use a Markov Chain model to analyze click transitions. In this model, each state is a click category, and edges represent transitions between categories. We add two abstract states, initial and final, that mark the beginning and end of each click session. Figure 7 shows the category transition probabilities for both Sybils and normal users. The sum of all outgoing transitions from each category is 1.0. To reduce the complexity of the Figure, edges with probability <5% have been pruned (except for transitions to the final state). Categories with no incoming edges after this pruning process are also omitted.

Figure 7(a) demonstrates that Sybils follow a very regimented set of behaviors. After logging-in Sybils immediately begin one of three malicious activities: friend invitation spamming, spamming photos, or profile browsing. The profile browsing path represents crawling behavior: the Sybil repeatedly views user profiles until their daily allotment of views is exhausted.

Compared to Sybils, normal users (Figure 7(b)) engage in a wider range of activities, and the transitions between states are more diverse. The highest centrality category is photos, and it is also the most probable state after login. Intuitively, users start from their newsfeed, where they are likely to see and click on friends' recent photos. The second most probable state after login is checking recent notifications. Sharing and messaging are both low probability states. This makes sense, given that studies of interactions on OSNs have shown that users generate new content less than once per day [41, 17].

It is clear from Figure 7 that currently, Sybils on Renren are not trying to precisely mimic the behavior of normal users. However, we do not feel that this type of modeling represents a viable Sybil detection approach.

Simply put, it would be trivial for Sybils to modify their behavior in order to appear more like normal users. If Sybils obfuscated their behavior by decreasing their transition probability to friending and profile browsing while increasing their transition probability to photos and blogs, then distinguishing between the two models would be extremely difficult.

3.4 SVM Classification

The above analysis shows that Sybil sessions have very different characteristics compared to normal user sessions. Based on these results, we explore the possibility of distinguishing normal and Sybil sessions using a Support Vector Machine (SVM) [26]. For our SVM experiments, we extract 4 features from session-level information and 8 features from click activities:

- **Session Features:** We leverage 4 features extracted from user sessions: average clicks per session, average session length, average inter-arrival time between clicks, and average sessions per day.
- **Click Features:** As mentioned in Section 3.2, there are 8 categories of click activities on Renren. For each user, we use the percentage of clicks in each category as a feature.

We computed values for all 12 features for all users in our dataset, input the data to an SVM, and ran 10 fold cross-validation. The resulting classification accuracy was 98.9%, with 0.8% false positives (*i.e.* classify normal users as Sybils) and 0.13% false negatives (*i.e.* classify Sybils as normal users). Table 3 shows the weights assigned to the top 5 features. Features with positive weight values are more indicative of Sybils, while features with negative weights indicate they are more likely in normal users. Overall, higher absolute value of the weights corresponds to features that more strongly indicate either Sybils or normal users. These features agree with our ad-hoc observations in previous sections.

Feature	Weight
% of clicks under <i>Friending</i>	+5.65
% of clicks under <i>Notification</i>	-3.68
Time interval of clicks (TBC)	-3.73
Session length (SL)	+1.34
% of clicks under <i>Photo</i>	+0.93

Table 3: Weight of features generated by SVM.

While our SVM results are quite good, an SVM-based approach is still a supervised learning tool. In practice, we would like to avoid using any ground truth datasets to train detection models, since they can introduce unknown biases. Later, we will describe our unsupervised detection techniques in detail.

3.5 Discussion

In summary, we analyze the Renren clickstream data to characterize user behavior from three angles: sessions, click activities, and click transitions. SVM analysis of these basic features demonstrates that clickstreams are useful for identifying Sybils on social networks.

However, these basic tools (session distributions, Markov Chain models, SVM) are of limited use in practice: they require training on large samples of ground-truth data. For a practical Sybil detection system, we must develop clickstream analysis techniques that leverage unsupervised learning on real-time data samples, *i.e.* require zero or little ground-truth. In the next section, we will focus on developing clickstreams models for real-time, unsupervised Sybil detection.

4 Clickstream Modeling and Clustering

In Section 3, we showed that clickstream data for Sybils and normal users captured the differences in their behavior. In this section, we build models of user activity patterns that can effectively distinguish Sybils from normal users. Our goal is to cluster similar clickstreams together to form general user “profiles” that capture specific activity patterns. We then leverage these clusters (or profiles) to build a Sybil detection system.

We begin by defining three models to represent a user’s clickstream. For each model, we describe similarity metrics that allow us to cluster similar clickstreams together. Finally, we use our ground-truth data to evaluate the efficacy of each model in distinguishing Sybils from normal users. We build upon these results later to develop practical Sybil detection systems based on clickstream analysis.

4.1 Clickstream Models

We define three models to capture a user’s clickstream.

Click Sequence Model. We start with the most straightforward model, which only considers click events. As shown in Section 3, Sybils and normal users exhibit different click transition patterns and focus their energy on different activities. The Click Sequence (CS) Model treats each user’s clickstream as a sequence of click events, sorted by order of arrival.

Time-based Model. As shown in Figure 6, Sybils and normal users generate click events at different speeds. The Time-based Model focuses on the distribution of gaps between events: each user’s clickstream is represented by a list of inter-arrival times $[t_1, t_2, t_3, \dots, t_n]$ where n is the number of clicks in a user’s clickstream.

Hybrid Model. The Hybrid Model combines click types and click inter-arrival times. Each user’s clickstream is represented as an in-order sequence of clicks along with inter-event gaps between clicks. For example: $a(t_1)c(t_2)a(t_3)d(t_4)b$ where a, b, c, d are click types, and t_i is the time interval between the i^{th} and $(i+1)^{th}$ event.

Click Types. Both the Click Sequence Model and the Hybrid Model represent each event in the sequence by its click event type. We note that we can control how granular the event types are in our sequence representation. One approach is to encode clicks based on their specific *activity*. Renren’s logs define 55 unique activities. Another option is to encode click events using their broader *category*. In our dataset, our 55 activities fall under 8 click categories (see Section 3.2). Our experimental analysis evaluates both representations to understand the impact of granularity on model accuracy.

4.2 Computing Sequence Similarity

Having defined three models of clickstream sequences, we now move on to investigating methods to quantify the similarity between clickstreams. In other words, we want to compute the *distance* between pairs of clickstreams. First, we discuss general approaches to computing the distance between sequences. Then we discuss how to apply each approach to our three clickstream models.

4.2.1 Defining Distance Functions

Common Subsequences. The first distance metric involves locating the common subsequences of varying lengths between two clickstreams. We formalize a clickstream as a sequence $S = (s_1 s_2 \dots s_i \dots s_n)$, where s_i is the i^{th} element in the sequence. We then define T_k as the set of all possible k -grams (k consecu-

tive elements) in sequence S : $T_k(S) = \{k\text{-gram} | k\text{-gram} = (s_i s_{i+1} \dots s_{i+k-1}), i \in [1, n+1-k]\}$. Simply put, each k -gram in $T_k(S)$ is a subsequence of S . Finally, the distance between two sequences can then be computed based on the number of common subsequences shared by the two sequences. Inspired by the *Jaccard Coefficient* [19], we define the distance between sequences S_1 and S_2 as:

$$D_k(S_1, S_2) = 1 - \frac{|T_k(S_1) \cap T_k(S_2)|}{|T_k(S_1) \cup T_k(S_2)|} \quad (1)$$

We will discuss the choice of k in Section 4.2.2.

Common Subsequences With Counts. The common subsequence metric defined above only measures distinct common subsequences, *i.e.* it does not consider the frequency of common subsequences. We propose a second distance metric that rectifies this by taking the *count* of common subsequences into consideration. For sequences S_1, S_2 and a chosen k , we first compute the set of all possible subsequences from both sequences as $T = T_k(S_1) \cup T_k(S_2)$. Next, we count the frequency of each subsequence within each sequence i ($i = 1, 2$) as array $[c_{i1}, c_{i2}, \dots, c_{in}]$ where $n = |T|$. Finally, the distance between S_1 and S_2 can be computed as the normalized *Euclidean Distance* between the two arrays:

$$D(S_1, S_2) = \frac{1}{\sqrt{2}} \sqrt{\sum_{j=1}^n (c_{1j} - c_{2j})^2} \quad (2)$$

Distribution-based Method. Unfortunately, the prior metrics cannot be applied to sequences of continuous values (*i.e.* the Time-based Model). Instead, for continuous value sequences S_1 and S_2 , we compute the distance by comparing their value distribution using a two-sample KolmogorovSmirnov test (*K-S* test). A two-sample *K-S* test is a general nonparametric method for comparing two empirical samples. It is sensitive to differences in location and shape of the empirical Cumulative Distribution Functions (CDF) of the two samples. We define the distance function using K-S statistics:

$$D(S_1, S_2) = \sup_t |F_{n,1}(t) - F_{n',2}(t)| \quad (3)$$

where $F_{n,i}(t)$ is the CDF of values in sequence S_i .

4.2.2 Applying Distances Functions to Clickstreams

Having defined three distance functions for computing sequence similarity, we now apply these metrics to our three clickstream models. Table 4 summarizes the distance metrics we apply to each of our models. The Time-based Model is the simplest case, because it only has one corresponding distance metric (K-S Test). For the Click Sequence and Hybrid Models, we use several different parameterizations of our distance metrics.

Model	Distance Metrics
Click Sequence Model	<i>unigram</i> , <i>unigram+count</i> , <i>10gram</i> , <i>10gram+count</i>
Time-based Model	<i>K-S</i> test
Hybrid Model	<i>5gram</i> , <i>5gram+count</i>

Table 4: Summary of distance functions.

Click Sequence Model. We use the common subsequence and common subsequence with counts metrics to compute distances in the CS model. However, these two metrics require that we choose k , the length of k -gram subsequences to consider. We choose two values for k : 1 and 10, which we refer to as *unigram* and *10gram*. *Unigram* represents the trivial case of comparing common click events in two clickstreams, while ignoring the ordering of clicks. *10gram* includes all unigrams, as well as bigrams, trigrams, *etc.* As shown in Table 4, we also instantiate *unigram+count* and *10gram+count*, which include the frequency counts of each unique subsequence.

Although values of $k > 10$ are possible, we limit our experiments to $k = 10$ for two reasons. First, when $k = n$ (where n is the length of a clickstream), the computational complexity becomes $O(n^2)$. This overhead is significant when you consider that $O(n^2)$ subsequences will be computed for every user in a clickstream dataset. Second, long subsequences have diminishing utility, because they are likely to be unique for a particular user. In our tests, we found $k = 10$ to be a good limit on computational overhead and subsequence over-specificity.

Hybrid Model. Like the Click Sequence Model, distances between sequences in the Hybrid Model can also be computed using the common subsequence and common subsequence plus count metrics. The only change between the Click Sequence and Hybrid Models is that we must discretize the inter-arrival times between clicks so they can be encoded into the sequence. We do this by placing inter-arrival times into log-scale buckets (in seconds): $[0, 1], [1, 10], [10, 100], [100, 1000], [1000, \infty]$. Based on Figure 6, the inter-arrival time distribution is highly skewed, so log-scale buckets are better suited than linear buckets to evenly encode the times.

After we discretize the inter-arrival times and insert them into the clickstream, we use $k = 5$ as the parameter for configuring the two distance metrics. Further increasing k offers little improvement in the model but introduces extra computation overhead. As shown in Table 4, we refer to these metrics as *5gram* and *5gram+count*. Thus, each 5gram contains three consecutive click events along with two tokens representing inter-arrival time gaps between them.

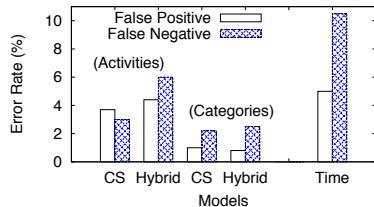


Figure 8: Error rate of three models.

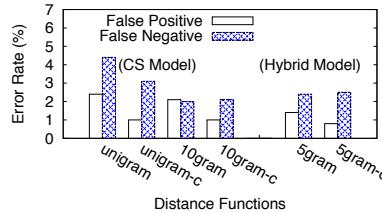


Figure 9: Error rate using different distance functions.

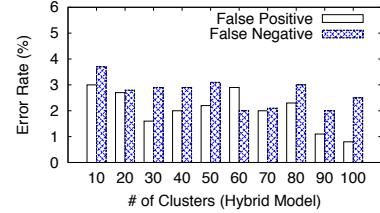


Figure 10: Impact of number of clusters (K).

4.3 Sequence Clustering

At this point we have defined models of clickstreams as well as metrics for computing the distance between them. Our next step is to cluster users with similar clickstreams together. As shown in Section 3, Sybil and normal users exhibit very different behaviors, and should naturally form distinctive clusters.

To achieve our goal, we build and then partition a *sequence similarity graph*. Each user’s clickstream is represented by a single node. The sequence similarity graph is complete, *i.e.* every pair of nodes is connected by a weighted edge, where the weight is the similarity distance between the sequences. Partitioning this graph means producing the desired clusters while minimizing the total weight of cut edges: users with similar activities (high weights between them) will be placed in the same cluster, while users with dissimilar activities will be placed in different clusters. Thus the clustering process separates Sybil and normal users. Note that not all Sybils and normal users exhibit homogeneous behavior; thus, we expect there to be multiple, distinct clusters of Sybils and normal users.

Graph Clustering. To cluster sequence similarity graphs, we use METIS [18], a widely used multilevel k-way partitioning algorithm. The objective of METIS is to minimize the weight of edges that cross partitions. In the sequence similarity graph, longer distances (*i.e.* dissimilar sequences) have lower weights. Thus, METIS is likely to place dissimilar sequences in different partitions. METIS requires a parameter K that specifies the number of partitions desired. We will assess the impact of K on our system performance in Section 4.4.

Cluster Quality. A key question when evaluating our methodology is assessing the quality of clusters produced by METIS. In Section 4.4, we leverage our ground-truth data to evaluate false positives and negatives after clustering the sequence similarity graph. We label each cluster as “Sybil” or “normal” based on whether the majority of nodes in the cluster are Sybils or normal users. Normal users that get placed into Sybil clusters are false positives, while Sybils placed in normal

clusters are false negatives. We use these criteria to evaluate different clickstream models and distance functions.

4.4 Model Evaluation

We now evaluate our clickstream models and distance functions to determine which can best distinguish Sybil activity patterns from those of normal users. We examine four different variables: 1) choice of clickstream model, 2) choice of distance function for each model, 3) what representation of clicks to use (specific activities or categories), and 4) K , the number of desired partitions for METIS.

Experiment Setup. The experimental dataset consists of 4000 normal users and 4000 Sybils randomly selected from our dataset. In each scenario, we build click sequences for each user (based on a given clickstream model and click representation), compute all distances between each pair of sequences, and then cluster the resulting sequence similarity graph for a given value of K . Finally, each experimental run is evaluated based on the false positive and negative error rates.

Model Analysis. First, we examine the error rates of different clickstream models and click representations in Figure 8. For the CS and Hybrid models, we encode clicks based on activities as well as categories. In the Time model, all clicks are encoded as inter-arrival times. In this experiment, we use 10gram+count, Sgram+count, and K -S as the distance function for CS, Hybrid, and Time, respectively. We fix $K = 100$. We investigate the impact of distance functions and K in subsequent experiments.

Two conclusions can be drawn from Figure 8. First, the CS and Hybrid models significantly outperform the Time-based model, especially in false negatives. This demonstrates that click inter-arrival times alone are insufficient to disambiguate Sybils from normal users. Manual inspection of false negative Sybils from the Time experiment reveals that these Sybils click at the same rate as normal users. Thus these Sybils are either operated by real people, or the software that controls them has been intentionally rate limited.

The second conclusion from Figure 8 is that encoding clicks based on category outperforms encoding by activity. This result confirms findings from the existing literatures on web usage mining [3]: representing clicks using high-level categories (or *concepts*) instead of raw click types better exposes the browsing patterns of users. A possible explanation is that high-level categories have better tolerance for noise in the clickstream log. In the rest of our paper, we use categories to encode clicks.

Next, we examine the error rate of different distance functions for the CS and Hybrid models. As shown in Figure 9, we evaluate the CS model using the *unigram* and *10gram* functions, as well as counting versions of those functions. We evaluate the Hybrid model using the *5gram* and *5gram+count* functions.

Several conclusions can be drawn from Figure 9. First, the *unigram* functions have the highest false negative rates. This indicates that looking at clicks in isolation (*i.e.* without click transitions) is insufficient to discover many Sybils. Second, the counting versions of all three distance functions produce low false positive rates. This demonstrates that the repeat frequency of sequences is important for identifying normal users. Finally, we observe that CS *10gram+count* and Hybrid have similar accuracy. This shows that click inter-arrival times are not necessary to achieve low error rates.

Finally, we examine the impact of the number of clusters K on detection accuracy. Figure 10 shows the error rate of Hybrid *5gram+count* as we vary K . The overall trend is that larger K produces lower error rates. This is because larger K grants METIS more opportunities to partition weakly connected sequences. This observation is somewhat trivial: if $K = N$, where N is the number of sequences in the graph, then the error rate would be zero given our evaluation methodology. In Section 6, we discuss practical reasons why K must be kept ≈ 100 .

Summary. Our evaluation shows that the Click Sequence and Hybrid models perform best at disambiguating Sybils and normal users. *10gram+count* and *5gram+count* are the best distance functions for each model, respectively. We find that accuracy is highest when clicks are encoded based on categories, and when the number of partitions K is large. In the following sections, we will use these parameters when building our Sybil detection system.

5 Incremental Sybil Detection

Our results in Section 4 showed that our models can effectively distinguish between Sybil clickstreams and normal user clickstreams. In this section, we leverage this methodology to build a real-time, incremental Sybil detector. This system works in two phases: first, we create clusters of Sybil and normal users based on ground-

truth data, as we did in Section 4. Second, we compute the position of unclassified clickstreams in our sequence similarity graph. If an unclassified clickstream falls into a cluster representing clickstreams from ground-truth Sybils, we conclude the new clickstream is a Sybil. Otherwise, it is benign.

5.1 Incremental Detection

To classify a new clickstream given an existing clustered sequence similarity graph, we must determine how to “re-cluster” new sequences into the existing graph. We investigate three algorithms.

The first is *K Nearest Neighbor* (KNN). For a given unclassified sequence, we find the top-K nearest sequences in the ground-truth data. If the majority of these sequences are located in Sybil clusters, then the new sequence is classified as a Sybil sequence.

The second algorithm is *Nearest Cluster* (NC). We compute the average distance from an unclassified sequence to all sequences in each cluster. The unclassified sequence is then added to the cluster with the closest average distance. The new sequence is classified as Sybil or normal based on the cluster it is placed in.

The third algorithm is a less computationally-intensive version of Nearest Cluster that we refer to as *Nearest Cluster-Center* (NCC). NC and KNN require computing the distance from an unclassified sequence to all sequences in the ground-truth clusters. We can streamline NC’s classification process by precomputing *centers* for each cluster. In NCC, we only need to compute the distance from an unclassified sequence to the center of each existing cluster.

For each existing cluster, the center is chosen by *closeness centrality*. Intuitively, the center sequence is the one that has the shortest distance to all the other sequences in the same cluster. To be more robust, we precompute three centers for each cluster, that is, the three sequences with highest closeness centrality.

5.2 System Evaluation

In this section, we evaluate our incremental Sybil detection system using our ground-truth clickstream dataset. We start by evaluating the basic accuracy of the system at classifying unknown sequences. Next, we evaluate how quickly the system can identify Sybils, in terms of number of clicks in their clickstream. Finally, we explore how long the system can remain effective before it needs to be retrained using updated ground-truth data.

Detection Accuracy. We start with a basic evaluation of system accuracy using our ground-truth dataset. We split the dataset into training data and testing data. Both datasets include 3000 Sybils and 3000 normal users. We build sequence similarity graphs from the training data

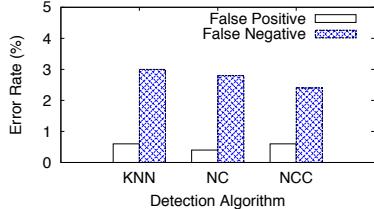


Figure 11: Error rate of three reclustering algorithms.

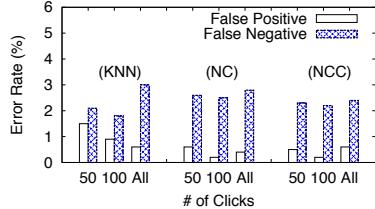


Figure 12: Error rate vs. maximum # of clicks in each sequence.

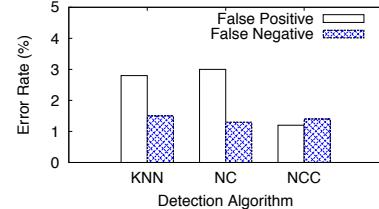


Figure 13: Detection accuracy when training data is two weeks old.

using Hybrid Model with *5gram+count* as distance function. The number of clusters $K = 100$. In each sequence similarity graph, we label the Sybil and normal clusters.

Next, we examine the error rates of the incremental detector when unclassified users (3000 Sybils and 3000 normal users) are added to the sequence similarity graph. We perform this experiment three times, once for each of the proposed reclustering algorithms (KNN, NC and NCC). As shown in Figure 11, the error rates for all three reclustering algorithms are very similar, and all three have <1% false positives. NC has slightly fewer false positives, while NCC has the fewest false negatives.

Detection Speed. The next question we want to address is: *what is the minimum number of clicks necessary to accurately classify clickstreams?* Another way to frame this question is in terms of detection speed: *how quickly (in terms of clicks) can our system accurately classify clickstreams?* To identify and respond to Sybils quickly, we must detect Sybils using the minimal number of click events.

Figure 12 shows the results of our evaluation when the maximum number of clicks in all sequences are capped. The “All” results refer to a cap of infinity, *i.e.* all clicks in our dataset are considered. Note that not all sequences in our dataset have 50 or 100 clicks: some Sybils were banned before they produced this many clicks. Hence, the caps are upper bounds on sequence length.

Surprisingly, the “All” results are not the most accurate overall. As shown in Figure 12, using all clicks results in more false negatives. This occurs due to overfitting: given a large number of very long clickstreams from normal users, it is likely that they will occasionally exhibit unusual, Sybil-like behavior. However, this problem is mitigated if the sequence length is capped, since this naturally excludes these infrequent, aberrant clickstreams.

In contrast to the “All” results, the results from the ≤ 50 click experiments produce the most false positives. This demonstrates that there is a minimum sequence length necessary to perform accurate classification of clickstreams. We repeated these experiments using *CS/10gram+count* and received similar result, which we omit for brevity.

There are two additional, practical take-aways from Figure 12. First, the NCC algorithm performs equally well versus NC and KNN. This is a positive result, since the computational complexity of NCC is dramatically lower than NC and KNN. Second, we observe that our system can produce accurate results (false positives <1%, false negatives <3%) when only considering short sequences. This means that the system can make classifications quickly, without needing to store very long clickstreams in memory.

Accuracy Over Time. In order for our incremental detection system to be practically useful, its accuracy should remain high for long periods of time. Put another way, sequence similarity graphs trained with old data should be able to detect fresh Sybil clickstreams. To evaluate the accuracy of our system over time, we split our dataset based on date. We train our detector using the early data, and then apply the detector to the later data. We restrict our analysis to data from April 2011; although we have Sybil data from March 2011, we do not have corresponding data on normal users for this month.

Figure 13 shows the accuracy of the detector when it is trained on data from March 31-April 15, then applied to data from April 16-30. As the results show, the detector remains highly accurate for at least two weeks after it has been trained using the NCC reclustering algorithm. Unfortunately, the limited duration of our dataset prevents us from examining accuracy at longer time intervals.

We repeated this experiment using only one week of training data, but the false negative rate of the detector increased to $\approx 10\%$. This shows that the detector needs to be trained on sufficient data to provide accurate results.

6 Unsupervised Sybil Detection

Our incremental Sybil detection system from Section 5 has a serious shortcoming: it must be trained using large samples of ground-truth data. In this section, we develop an unsupervised Sybil detection system that requires only a small, constant amount of ground-truth. The key idea is to build a clustered sequence similarity graph as before. But instead of using full ground-truth

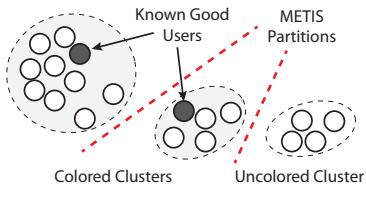


Figure 14: Unsupervised clustering with coloring.

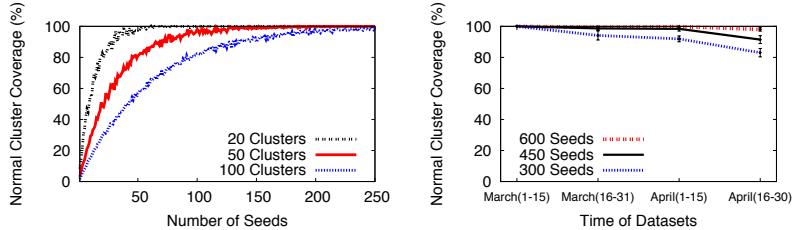


Figure 15: # of seeds vs. % of correctly colored normal user clusters.

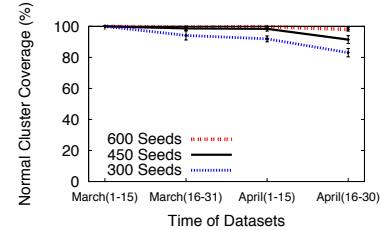


Figure 16: Consistency over time of normal seeds for coloring.

of all clickstreams to mark a cluster as Sybil or normal, we only need a small number of clickstreams of known real users as “seeds” that color the clusters they reside in. These seeds can be manually verified as needed. We *color* all clusters that include a *seed* sequence as “normal,” while uncolored clusters are assumed to be “Sybil.” Since normal users are likely to fall under a small number of behavioral profiles (clusters in the graph), we expect a small fixed number of seeds will be sufficient to color all clusters of normal user clickstreams.

Figure 14 depicts our unsupervised approach, showing how METIS partitions nodes into clusters which are then colored if they contain seed users. Once the system is trained in this manner, it can be used incrementally to detect more Sybils over time, as described in Section 5.

In this section, we discuss the design of our unsupervised system and evaluate its performance. We begin by analyzing the number and composition of seeds that are necessary to ensure high accuracy of the system. Next, we evaluate the performance of the system by comparing its accuracy to our ground-truth data. Finally, we examine how the ratio of Sybils to normal users in the unclassified data impacts system accuracy.

6.1 Seed Selection and Composition

Number of Seeds. The most important parameter in our unsupervised Sybil detection system is the number of seeds. On one hand, the number of seeds needs to be large and diverse enough to color all “normal” clusters. Normal clusters that remain uncolored are potential false positives. On the other hand, the seed set needs to be small enough to be practical. If the size of the seed set is large, it is equivalent to having ground-truth about the dataset, which is the situation we are trying to avoid.

We now conduct experiments to determine how many seeds are necessary to color the clusters. We choose 3000 Sybils and 3000 normal users at random from our dataset to be the unclassified dataset. We also randomly choose some number of additional normal users to be the seeds. As in Section 5, we use the Hybrid Model with the *5gram+count* distance function. We also conducted

experiments with *CS/10gram+count*, but the results are very similar and we omit them for brevity.

Figure 15 depicts the percentage of normal of clusters that are correctly colored for different values of K (number of METIS partitions) as the number of seeds is varied. As expected, fewer seeds are necessary when K is small because there are fewer clusters (and thus each cluster includes more sequences). When $K = 100$, 250 seeds (or 4% of all normal users in the experiment) are able to color 99% of normal clusters.

Seed Consistency Over Time. Next, we examine whether a set of seeds chosen at an early date are equally effective at coloring clusters based on later data. In other words, we want to know if the seeds are consistent over time. If this is not the case, it would represent additional overhead on the deployment of our system.

To test seed consistency over time, we divide our two months of Sybil clickstream data into four, two-week long datasets. We add an equal number of randomly selected normal users to each of the four datasets. Finally, we select an additional x random normal users to act as seeds. We verify (for each value of x) that these seeds color 100% of the normal clusters in the first temporal dataset. We then evaluate what percentage of normal clusters are colored in the subsequent three temporal datasets. In all experiments, we set $K = 100$, *i.e.* the worst case scenario for our graph coloring approach.

The results of the temporal consistency experiments are shown in Figure 16. In general, even though the Sybil and normal clickstreams change over time, the vast majority of normal clusters are successfully colored. Given 600 seeds, 99% of normal clusters are colored after 4 weeks, although the percentage drops to 83% with 300 seeds. These results demonstrate that the seed set does not need to be drastically altered over time.

6.2 Coloring Evaluation

We now evaluate the overall effectiveness of our Sybil detection system when it leverages unsupervised training. In these experiments, we use our entire clickstream dataset. We choose x random normal users as seeds,

build and cluster the sequence similarity graph using Hybrid/5gram+count, and then color the clusters that contain the seeds. Finally, we calculate the false positive and negative rates using the same methodology as in Section 5, *i.e.* by comparing the composition of the colored clusters to ground-truth.

The results are shown in Figure 17. As the number of seeds increases, the false positive rate decreases. This is because more seeds mean more normal clusters are correctly colored. With just 400 seeds, the false positive rate drops to <1%. Unfortunately, relying on unsupervised training does increase the false negative rate of our system by 2% versus training with ground-truth data. However, in cases where ground-truth data is unavailable, we believe that this is a reasonable trade-off. Note that we also repeated these experiments with CMS/10gram+count, and it produced slightly higher false positive rates, although they were still <1%.

Unbalanced Training Dataset. Next, we evaluate the impact of having an unbalanced training dataset (*e.g.* more normal users than Sybils) on the accuracy of our system. Thus far, all of our experiments have assumed a roughly equal percentage of Sybils and normal users in the data. However, in practice it is likely that normal users will outnumber Sybils when unsupervised learning is used. For example, Facebook suspects that 8.7% of its user base is illegitimate, out of >1 billion total users [1].

We now evaluate how detection accuracy changes when we decrease the percentage of Sybils in the training data. In these experiments, we construct training sets of 6000 total users with different normal-to-Sybil ratios. We then run unsupervised training with 500 seeds. Finally, we incrementally add an additional 3000 Sybils and 3000 normal users to the colored similarity graph using the NCC algorithm (see Section 5.1). We ran additional tests using the NC and KNN algorithms, but the results were very similar and we omit them for brevity.

Figure 18 shows the final error rate of the system (*i.e.* after 6000 users have been incrementally added) for varying normal-to-Sybil ratios. The false positive rate remains $\leq 1.2\%$ regardless of the normal-to-Sybil ratio. This is a very good result: even with highly skewed training data, the system is unlikely to penalize normal users. Unfortunately, the false negative rate does rise as the number of Sybils in the training data falls. This result is to be expected: the system cannot adequately classify Sybil clickstreams if it is trained on insufficient data.

Handling False Positives. The above analysis demonstrates that our system achieves high accuracy with a false positive rate of 1% or less. Through manual inspection, we find that “false positives” generated by our detector exhibit behaviors generally attributed to Sybils, including aggressively sending friend requests or

browsing profiles. In real-world OSNs, suspicious users identified by our system could be further verified via existing complementary systems that examines other aspects of users. For example, this might include systems that classify user profiles [32, 43], systems that verify user real-world identity [2], or even Sybil detection systems using crowdsourced human inspection [38]. These efforts could further protect benign users from misclassification.

7 Practical Sybil Detection

In this section, we examine the practical performance of our proposed Sybil detection system. First, we shipped our code to the security teams at Renren and LinkedIn, where it was evaluated on fresh data in a production environment. Both test results are very positive, and we report them here. Second, we discuss the fundamental limits of our approach, by looking at our impact on Sybil accounts that can perfectly mimic the clickstream patterns of normal users.

7.1 Real-world Sybil Detection

With the help of supportive collaborators at both Renren and LinkedIn, we were able to ship prototype code to the security teams at both companies for internal testing on fresh data. We configured our system to use unsupervised learning to color clusters. Sequence similarity graphs are constructed using the Hybrid Model and the 5gram+count distance function, and the number of METIS partitions K is 100.

Renren. Renren’s security team trained our system using clickstreams from 10K users, of which 8K were randomly selected, and 2K were previously identified as suspicious by the security team. These clickstreams were collected between January 17–27, 2013. 500 honest users that have been manually verified by Renren’s security team were used as seeds. Once trained, our system was fed clickstreams from 1 million random users (collected in early February, 2013) for classification as normal or suspicious. In total, our system identified 22K potential Sybil accounts. These accounts are now being investigated by the security team.

While corporate privacy policies prevented Renren from sharing detailed results with us, their feedback was very positive. They also indicated that our system identified a new attack performed by a large cluster of users whose clickstream behavior focused heavily on photo sharing. Manual inspection revealed that these photos used embedded text to spread spam for brands of clothes and shoes. Traditional text analysis-based spam detectors and URL blacklists were unable to catch this new attack, but our system identified it immediately.

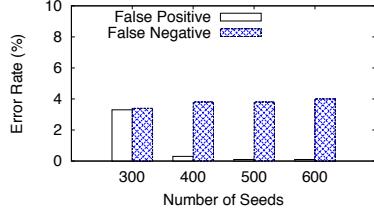


Figure 17: Detection accuracy versus number of seeds.

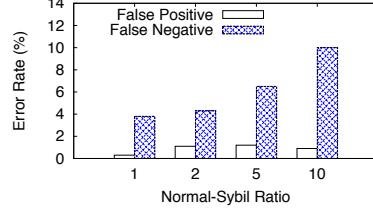


Figure 18: Detection accuracy versus Normal-Sybil ratio.

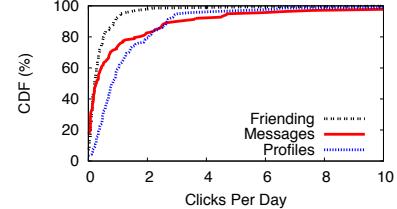


Figure 19: Clicks per day by outlier normal users.

LinkedIn. LinkedIn’s security team used our software to analyze the clickstreams of 40K users, of which 36K were randomly sampled, and 4K were previously identified as suspicious by the security team. These clickstreams were gathered in February, 2013. Again, our feedback was very positive, but did not include precise statistics. However, we were told that our system confirmed that ≈ 1700 of the 4000 suspicious accounts are likely to be Sybils. Our system also detected an additional 200 previously unknown Sybils.

A closer look at the data shows that many of the accounts not detected by our system were borderline accounts with specific flags popping up in their profiles. For example, some accounts had unusual names or occupational specialties, while others had suspicious URLs in their profiles. These results remind us that a behavior model is clearly only a part of the equation, and should be used in conjunction with existing profile analysis tools and spam detectors [5, 10, 37, 38, 44].

Ongoing Collaboration. In summary, the security teams at both Renren and LinkedIn were very pleased with the initial results of our system. We plan to continue collaborating with both groups to improve our system and implement it in production.

7.2 Limits of Sybil Detection

Finally, we wish to discuss the worst case scenario for our system, *i.e.* a scenario where attackers have full knowledge of the clickstream patterns for real users, and are able to instrument the behavior of their Sybils to mimic them precisely. In this attack model, the attacker’s goal is to have Sybils carry out malicious actions (*e.g.* sending spam) without being detected. However, to evade detection, these Sybils must limit themselves to behavior consistent with that of normal users.

We can thus bound the capabilities of Sybils that avoid detection in this attack model. First, the Sybil’s clickstream must remain inside the “normal” clusters produced by our detector. Second, the most aberrant behavior within a given “normal” cluster is exhibited by real users within the cluster who are farthest from the center.

The activities performed by these *outliers* serve as effective bounds on Sybil behavior. Sybil clickstreams cannot deviate from the center of the cluster more than these outliers, otherwise they will be excluded from the cluster and risk detection. Thus, we can estimate the maximum amount of malicious activity a Sybil could perform (without getting caught) by studying these outliers.

We now examine the behavior of outliers. We calibrate our system to produce clusters with false positive rate $< 1\%$ using Hybrid/5gram+count, and $K = 100$. In this configuration, the detector outputs 40 Sybil and 60 normal clusters when run on our full dataset. Next, we identify the two farthest outliers in each normal cluster. Finally, we plot the clicks per day in three activities from the 120 outliers in Figure 19. We focus on clicks for sending friend requests, posting status updates/wall messages, and viewing user profiles. These activities correspond to the three most common attacks we observe in our ground-truth data, *i.e.* sending friend request spam, status/wall spam, and profile crawling.

As shown in Figure 19, 99% of outliers generate ≤ 10 clicks per day in the target activities. In the vast majority of cases, even the outliers generate < 2 clicks per day. These results show that the effective bound on Sybil behavior is very tight, *i.e.* to avoid detection, Sybils can barely generate any clicks each day. These bounds significantly increase the cost for attackers, since they will need many more Sybils to maintain the same level of spam generation capacity.

8 Related Work

Sybil Detection on OSNs. Studies have shown that Sybils are responsible for large amounts of spam on Facebook [10], Twitter [11, 32], and Renren [43]. Various systems have been proposed by the research community to detect and mitigate these Sybils. One body of work leverages social graphs to detect Sybils. These systems detect tight-knit Sybil communities that have a small quotient-cut from the honest region of the graph [46, 45, 34, 36, 8, 7]. However, recent studies have demonstrated the limitations of this approach. Yang *et al.*

show that Sybils on Renren blend into the social graph rather than forming tight communities [43]. Mohaisen *et al.* show that many social graphs are not fast-mixing, which is a necessary precondition for community-based Sybil detectors to be effective [21].

A second body of work has used machine learning to detect Sybil behavior on Twitter [44, 5, 37] and Facebook [31]. However, relying on specific features makes these systems vulnerable to Sybils with different attack strategies. Finally, one study proposes using crowdsourcing to identify Sybils [38].

Web Usage Mining. Researchers have studied the usage patterns of web services for the last decade [30]. Several studies focus on session-level analysis to learn user’s browsing habits [14, 13, 24]. Others develop session clustering techniques [4, 42, 40, 33, 25], Markov Chain models [20, 28], and tree-based models [12] to characterize user browsing patterns. We also leverage a Markov Chain model and clustering in our work. Two studies have focused specifically on characterizing clickstreams from OSNs [6, 29].

The vast majority of the web usage mining literature focuses on characterizing the behavior of normal users. To the best of our knowledge, there are only two studies that leverage clickstreams for anomaly detection [15, 28]. Both of these studies use session-level features to identify crawlers, one focusing on e-commerce and the other on search engines. Their techniques (*e.g.* session distributions, Markov Chain models) require training on large samples of ground-truth data, and cannot scale to today’s large social networks.

9 Conclusion

To the best of our knowledge, this is the first work to leverage clickstream models for detecting malicious users in OSNs. Our results show that we can build an accurate Sybil detector by identifying and coloring clusters of “similar” clickstreams. Our system has been validated on ground-truth data, and a prototype has already detected new types of image-spam attacks on Renren.

We believe clickstream models can be a powerful technique for user profiling in contexts outside of OSNs. In our ongoing work, we are studying ways to extend clickstream models to detect malicious crowdsourcing workers and forged online product and travel reviews.

IRB Protocol

This work was carried out under an approved IRB protocol. All data was anonymized by Renren prior to our use. The clickstreams are old enough that the events they describe are no longer accessible via the current website. All experiments run on recent user data were conducted

on-site at Renren and LinkedIn respectively, and all results remain on-site.

Acknowledgments

We would like to thank the anonymous reviewers for their feedback, and Yanjie Liang (Renren) and David Freeman (LinkedIn) for their assistance in experiments. This work is supported in part by NSF grants CNS-1224100 and IIS-0916307 and DARPA GRAPHs (BAA-12-01). Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the funding agencies.

References

- [1] Facebook has more than 83 million illegitimate accounts. BBC News, August 2012.
- [2] Verify facebook account. <https://www.facebook.com/help/398085743567023/>, 2013.
- [3] BANERJEE, A., AND GHOSH, J. Concept-based clustering of clickstream data. In *Proc. of ICIT* (2000).
- [4] BANERJEE, A., AND GHOSH, J. Clickstream clustering using weighted longest common subsequences. In *Proc. of the Web Mining Workshop, SIAM Conference on Data Mining* (2001).
- [5] BENEVENUTO, F., MAGNO, G., RODRIGUES, T., AND ALMEIDA, V. Detecting spammers on twitter. In *Proc. of CEAS* (2010).
- [6] BENEVENUTO, F., RODRIGUES, T., CHA, M., AND ALMEIDA, V. Characterizing user behavior in online social networks. In *Proc. of IMC* (2009).
- [7] CAO, Q., SIRIVIANOS, M., YANG, X., AND PREGUEIRO, T. Aiding the detection of fake accounts in large scale social online services. In *Proc. of NSDI* (2012).
- [8] DANEZIS, G., AND MITTAL, P. Sybilinfer: Detecting sybil nodes using social networks. In *Proc. of NDSS* (2009).
- [9] DOUCEUR, J. R. The Sybil attack. In *Proc. of IPTPS* (2002).
- [10] GAO, H., HU, J., WILSON, C., LI, Z., CHEN, Y., AND ZHAO, B. Y. Detecting and characterizing social spam campaigns. In *Proc. of IMC* (2010).
- [11] GRIER, C., THOMAS, K., PAXSON, V., AND ZHANG, M. @spam: the underground on 140 characters or less. In *Proc. of CCS* (2010).
- [12] GÜNDÜZ, C., AND ÖZSU, M. T. A web page prediction model based on click-stream tree representation of user behavior. In *Proc. of SIGKDD* (2003).

- [13] HEER, J., AND CHI, E. H. Mining the structure of user activity using cluster stability. In *Proc. of the Workshop on Web Analytics, SIAM Conference on Data Mining* (2002).
- [14] HEER, J., AND CHI, E. H. Separating the swarm: categorization methods for user sessions on the web. In *Proc. of CHI* (2002).
- [15] HOFGESANG, P. I., AND KOWALCZYK, W. Analysing clickstream data: From anomaly detection to visitor profiling. In *Proc. of ECML/PKDD Discovery Challenge* (2005).
- [16] IRANI, D., BALDUZZI, M., BALZAROTTI, D., KIRDA, E., AND PU, C. Reverse social engineering attacks in online social networks. In *Proc. of DIMVA* (2011).
- [17] JIANG, J., WILSON, C., WANG, X., HUANG, P., SHA, W., DAI, Y., AND ZHAO, B. Y. Understanding latent interactions in online social networks. In *Proc. of IMC* (2010).
- [18] KARYPIS, G., KUMAR, V., AND KUMAR, V. Multilevel k-way partitioning scheme for irregular graphs. *Journal of Parallel and Distributed Computing* 48 (1998), 96–129.
- [19] LEVANDOWSKY, M., AND WINTER, D. Distance between sets. *Nature* 234 (1971), 34–35.
- [20] LU, L., DUNHAM, M., AND MENG, Y. Mining significant usage patterns from clickstream data. In *Proc. of WebKDD* (2005).
- [21] MOHAISEN, A., YUN, A., AND KIM, Y. Measuring the Mixing Time of Social Graphs. In *Proc. of IMC* (2010).
- [22] MOTOYAMA, M., LEVCHENKO, K., KANICH, C., MCCOY, D., VOELKER, G. M., AND SAVAGE, S. Re: Captchas – understanding captcha-solving from an economic context. In *Proc. of USENIX Security* (2010).
- [23] MOTOYAMA, M., MCCOY, D., LEVCHENKO, K., SAVAGE, S., AND VOELKER, G. M. Dirty jobs: The role of freelance labor in web service abuse. In *Proc. of Usenix Security* (2011).
- [24] OBENDORF, H., WEINREICH, H., HERDER, E., AND MAYER, M. Web page revisitaton revisited: implications of a long-term click-stream study of browser usage. In *Proc. of CHI* (2007).
- [25] PETRIDOU, S. G., KOUTSONIKOLA, V. A., VAKALI, A. I., AND PAPADIMITRIOU, G. I. Time-aware web users’ clustering. *IEEE Trans. on Knowl. and Data Eng.* (2008), 653–667.
- [26] PLATT, J. C. Advances in kernel methods. MIT Press, 1999, ch. Fast training of support vector machines using sequential minimal optimization, pp. 185–208.
- [27] Russian twitter political protests ‘swamped by spam’. BBC News, December 2011.
- [28] SADAGOPAN, N., AND LI, J. Characterizing typical and atypical user sessions in clickstreams. In *Proc. of WWW* (2008).
- [29] SCHNEIDER, F., FELDMANN, A., KRISHNAMURTHY, B., AND WILLINGER, W. Understanding online social network usage from a network perspective. In *Proc. of IMC* (2009).
- [30] SRIVASTAVA, J., COOLEY, R., DESHPANDE, M., AND TAN, P. N. Web usage mining: discovery and applications of usage patterns from Web data. *SIGKDD Explor. Newsl.* 1, 2 (2000), 12–23.
- [31] STRINGHINI, G., KRUEGEL, C., AND VIGNA, G. Detecting spammers on social networks. In *Proc. of ACSAC* (2010).
- [32] THOMAS, K., ET AL. Suspended accounts in retrospect: An analysis of twitter spam. In *Proc. of IMC* (2011).
- [33] TING, I.-H., KIMBLE, C., AND KUDENKO, D. Ubb mining: Finding unexpected browsing behaviour in clickstream data to improve a web site’s design. In *Proc. of International Conference on Web Intelligence* (2005).
- [34] TRAN, N., MIN, B., LI, J., AND SUBRAMANIAN, L. Sybil-resilient online content voting. In *Proc. of NSDI* (2009).
- [35] VEGA, C. Yelp outs companies that pay for positive reviews. ABC News, November 2012. <http://abcnews.go.com/blogs/business/2012/11/yelp-outs-companies-that-pay-for-positive-reviews>.
- [36] VISWANATH, B., POST, A., GUMMADI, K. P., AND MISLOVE, A. An analysis of social network-based sybil defenses. In *Proc. of SIGCOMM* (2010).
- [37] WANG, A. H. Don’t follow me: Spam detection on twitter. In *Proc. of SECRYPT* (2010).
- [38] WANG, G., MOHANLAL, M., WILSON, C., WANG, X., METZGER, M., ZHENG, H., AND ZHAO, B. Y. Social turing tests: Crowdsourcing sybil detection. In *Proc. of NDSS* (2013).
- [39] WANG, G., WILSON, C., ZHAO, X., ZHU, Y., MOHANLAL, M., ZHENG, H., AND ZHAO, B. Y. Serf and turf: crowdturfing for fun and profit. In *Proc. of WWW* (2012).
- [40] WANG, W., AND ZAIANE, O. R. Clustering web sessions by sequence alignment. In *Proc. of DEXA* (2002).
- [41] WILSON, C., BOE, B., SALA, A., PUTTASWAMY, K. P. N., AND ZHAO, B. Y. User interactions in social networks and their implications. In *Proc. of EuroSys* (2009).
- [42] XIAO, J., AND ZHANG, Y. Clustering of web users using session-based similarity measures. In *Proc. of ICCNMC* (2001).
- [43] YANG, Z., WILSON, C., WANG, X., GAO, T., ZHAO, B. Y., AND DAI, Y. Uncovering social network sybils in the wild. In *Proc. of IMC* (2011).
- [44] YARDI, S., ROMERO, D., SCHOENEBECK, G., AND BOYD, D. Detecting spam in a twitter network. *First Monday* 15, 1 (2010).
- [45] YU, H., GIBBONS, P. B., KAMINSKY, M., AND XIAO, F. Sybillimit: A near-optimal social network defense against sybil attacks. In *Proc. of IEEE S&P* (2008).
- [46] YU, H., KAMINSKY, M., GIBBONS, P. B., AND FLAXMAN, A. Sybilguard: defending against sybil attacks via social networks. In *Proc. of SIGCOMM* (2006).

Alice in Warningland: A Large-Scale Field Study of Browser Security Warning Effectiveness

Devdatta Akhawe

*University of California, Berkeley**
devdatta@cs.berkeley.edu

Adrienne Porter Felt

Google, Inc.
felt@google.com

Abstract

We empirically assess whether browser security warnings are as ineffective as suggested by popular opinion and previous literature. We used Mozilla Firefox and Google Chrome’s in-browser telemetry to observe over 25 million warning impressions *in situ*. During our field study, users continued through a tenth of Mozilla Firefox’s malware and phishing warnings, a quarter of Google Chrome’s malware and phishing warnings, and a third of Mozilla Firefox’s SSL warnings. This demonstrates that security warnings can be effective in practice; security experts and system architects should not dismiss the goal of communicating security information to end users. We also find that user behavior varies across warnings. In contrast to the other warnings, users continued through 70.2% of Google Chrome’s SSL warnings. This indicates that the user experience of a warning can have a significant impact on user behavior. Based on our findings, we make recommendations for warning designers and researchers.

1 Introduction

An oft-repeated maxim in the security community is the futility of relying on end users to make security decisions. Felten and McGraw famously wrote, “Given a choice between dancing pigs and security, the user will pick dancing pigs every time [21].” Herley elaborates [17],

Not only do users take no precautions against elaborate attacks, they appear to neglect even basic ones. For example, a growing body of measurement studies make clear that ...[users] are oblivious to security cues [27], ignore certificate error warnings [31] and cannot tell legitimate web-sites from phishing imitations [11].¹

*The Mozilla Firefox experiments were implemented while the author was an intern at Mozilla Corporation.

¹Citations updated to match our bibliography.

The security community’s perception of the “oblivious” user evolved from the results of a number of laboratory studies on browser security indicators [5, 11, 13, 15, 27, 31, 35]. However, these studies are not necessarily representative of the current state of browser warnings in 2013. Most of the studies evaluated warnings that have since been deprecated or significantly modified, often in response to criticisms in the aforementioned studies. Our goal is to investigate whether modern browser security warnings protect users in practice.

We performed a large-scale field study of user decisions after seeing browser security warnings. Our study encompassed 25,405,944 warning impressions in Google Chrome and Mozilla Firefox in May and June 2013. We collected the data using the browsers’ telemetry frameworks, which are a mechanism for browser vendors to collect pseudonymous data from end users. Telemetry allowed us to unobtrusively measure user behavior during normal browsing activities. This design provides realism: our data reflects users’ actual behavior when presented with security warnings.

In this paper, we present the rates at which users click through (i.e., bypass) malware, phishing, and SSL warnings. Low clickthrough rates are desirable because they indicate that users notice and heed the warnings. Click-through rates for the two browsers’ malware and phishing warnings ranged from 9% to 23%, and users clicked through 33.0% of Mozilla Firefox’s SSL warnings. This demonstrates that browser security warnings can effectively protect most users in practice.

Unfortunately, users clicked through Google Chrome’s SSL warning 70.2% of the time. This implies that the user experience of a warning can have a significant impact on user behavior. We discuss several factors that might contribute to this warning’s higher clickthrough rates. Our positive findings for the other five warnings suggest that the clickthrough rate for Google Chrome’s SSL warning can be improved.

We also consider user behaviors that are indicative of attention to warnings. We find that Google Chrome's SSL clickthrough rates vary by the specific type of error. In Mozilla Firefox, a fifth of users who choose to click through an SSL warning remove a default option, showing they are making cognitive choices while bypassing the warning. Together, these results contradict the stereotype of the wholly oblivious user with no interest in security.

We conclude that users can demonstrate agency when confronted with browser security warnings. Users do not always ignore security warnings in favor of their desired content. Consequently, security experts and platform designers should not dismiss the role of the user. We find that the user experience of warnings can have an enormous impact on user behavior, justifying efforts to build usable warnings.

Contributions. We make the following contributions:

- To our knowledge, we present the first in-depth, large-scale field study of browser security warnings.
- We survey prior laboratory studies of browser security warnings and discuss why our field study data differs from prior research.
- We analyze how demographics (operating system and browser channel), warning frequency, and warning complexity affect users' decisions. Notably, we find evidence suggesting that technically skilled users ignore warnings more often, and warning frequency is inversely correlated with user attention.
- We provide suggestions for browser warning designers and make recommendations for future studies.

2 Background

Web browsers show warnings to users when an attack might be occurring. If the browser is certain that an attack is occurring, it will show an error page that the user cannot bypass. If there is a chance that the perceived attack is a false positive, the browser will show a bypassable warning that discourages the user from continuing. We study only bypassable warnings because we focus on user decisions.

A user *clicks through* a warning to dismiss it and proceed with her original task. A user *leaves* the warning when she navigates away and does not continue with her original task. A *clickthrough rate* describes the proportion of users who clicked through a warning type. When a user clicks through a warning, the user has (1) ignored the warning because she did not read or understand it or (2) made an informed decision to proceed because she believes that the warning is a false positive or her computer is safe against these attacks (e.g., due to an antivirus).

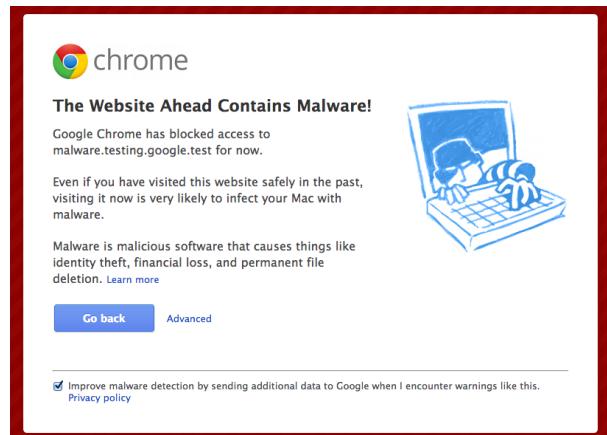


Figure 1: Malware warning for Google Chrome

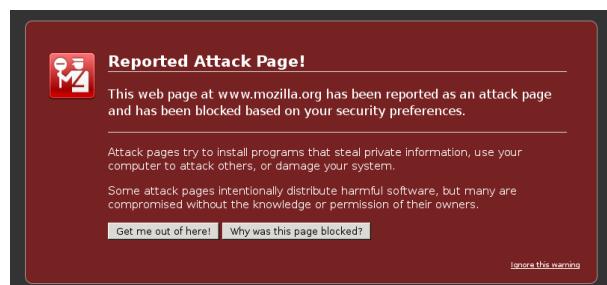


Figure 2: Malware warning for Mozilla Firefox

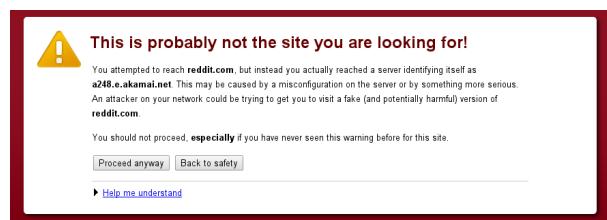


Figure 3: SSL warning for Google Chrome. The first paragraph changes depending on the specific SSL error.

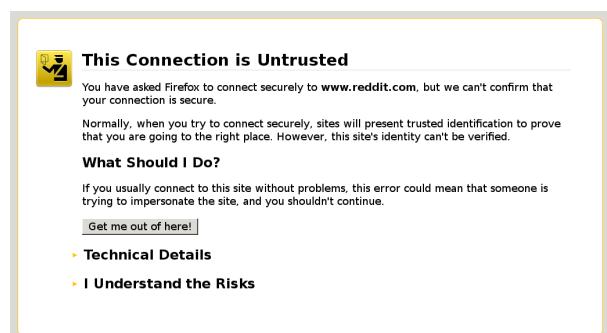


Figure 4: SSL warning for Mozilla Firefox



Figure 5: SSL Add Exception Dialog for Mozilla Firefox

We focus on three types of browser security warnings: malware, phishing, and SSL warnings. At present, all three types of warnings are full-page, interstitial warnings that discourage the user from proceeding.

2.1 Malware and Phishing Warnings

Malware and phishing warnings aim to prevent users from visiting websites that serve malicious executables or try to trick users. Google Chrome and Mozilla Firefox rely on the Google Safe Browsing list [26] to identify malware and phishing websites. The browsers warn users away from the sites instead of blocking them because the Safe Browsing service occasionally has false positives, although the false positive rate is very low [26].

Clickthrough Rate. If a malware or phishing warning is a true positive, clicking through exposes the user to a dangerous situation. Nearly all Safe Browsing warnings are true positives; the false positive rate is low enough to be negligible. The ideal clickthrough rate for malware and phishing warnings is therefore close to 0%.

Warning Mechanisms. The browsers routinely fetch a list of suspicious (i.e., malware or phishing) sites from Safe Browsing servers. If a user tries to visit a site that is on the locally cached list, the browser checks with the Safe Browsing service that the URL is still on the malware or phishing list. If the site is still on one of the lists, the browser presents a warning.

The two browsers behave differently if a page loads a third-party resource (e.g., a script) from a URL on the Safe Browsing list. Google Chrome stops the page load and replaces the page with a warning. Mozilla Firefox blocks the third-party resource with no warning. As a result, Mozilla Firefox users can see fewer warnings than Google Chrome users, despite both browsers using the same Safe Browsing list.

Warning Design. Figures 1 and 2 show the Google Chrome and Mozilla Firefox warnings. Their phishing warnings are similar to their respective malware warnings. When a browser presents the user with a malware or phishing warning, she has three options: leave the page via the warning’s escape button, leave the page by closing the window or typing a new URL, or click through the warning and proceed to the page. The warnings also allow the user to seek more information about the error.

Click Count. Mozilla Firefox users who want to bypass the warning need to click one button: the “Ignore this warning” link at the bottom right. On the other hand, Chrome users who want to bypass the warning need to click twice: first on the “Advanced” link, and then on “Proceed at your own risk.”

2.2 SSL Warnings

The Secure Sockets Layer (SSL/TLS) protocol provides secure channels between browsers and web servers, making it fundamental to user security and privacy on the web. As a critical step, the browser verifies a server’s identity by validating its public-key certificate against a set of trusted root authorities. This validation will fail in the presence of a man-in-the-middle (MITM) attack.

Authentication failures can also occur in a wide variety of benign scenarios, such as server misconfigurations. Browsers usually cannot distinguish these benign scenarios from real MITM attacks. Instead, browsers present users with a warning; users have the option to bypass the warning, in case the warning is a false positive.

Clickthrough Rate. We hope for a 0% clickthrough rate for SSL warnings shown during MITM attacks. However, many SSL warnings may be false positives (e.g., server misconfigurations). There are two competing views regarding SSL false positives. In the first, warning text should discourage users from clicking through both true and false positives, in order to incentivize developers to get valid SSL certificates. In the other, warning text should provide users with enough information to correctly identify and dismiss false positives. The desired clickthrough rates for false-positive warnings would be 0% and 100%, respectively. In either case, false positives are undesirable for the user experience because we do not want to annoy users with invalid warnings. Our goal is therefore a 0% clickthrough rate for all SSL warnings: users should heed all valid warnings, and the browser should minimize the number of false positives.

Warning Design. Figures 3 and 4 present Google Chrome and Mozilla Firefox’s SSL warnings. The user can leave via the warning’s escape button, manually navigate away, or click through the warning. In Mozilla Firefox, the user must also click through a second dialog (Figure 5) to bypass the warning.

The browsers differ in their presentation of the technical details of the error. Google Chrome places information about the specific error in the main warning (Figure 3, first paragraph), whereas Firefox puts the error information in the hidden “Technical Details” section and the second “Add Exception” dialog (Figure 5).

Click Count. Mozilla Firefox’s SSL warning requires more clicks to bypass. Google Chrome users click through a single warning button to proceed. On the other hand, Mozilla Firefox’s warning requires three clicks: (1) click on “I Understand The Risks,” (2) click on the “Add Exception” button, which raises a second dialog, (3) click on “Confirm Security Exception” in the second dialog. By default, Firefox permanently remembers the exception and will not show the warning again if the user reencounters the same certificate for that website. In contrast, Chrome presents the warning every time and does not remember the user’s past choices.

2.3 Browser Release Channels

Mozilla and Google both follow rapid release cycles. They release official versions of their browsers every six or seven weeks, and both browsers update automatically. The official, default version of a browser is referred to as “stable” (Google Chrome) or “release” (Mozilla Firefox).

If users are interested in testing pre-release browser versions, they can switch to a different *channel*. The stable/release channel is the recommended channel for end users, but a minority of users choose to use earlier channels to test cutting-edge features. The “Beta” channel is several weeks ahead of the stable/release channel. The “developer” (Google Chrome) or “Aurora” (Mozilla Firefox) channel is delivered even earlier. Both browsers also offer a “nightly” (Mozilla Firefox) or “Canary” (Google Chrome) release channel, which updates every day and closely follows the development repository.

The pre-release channels are intended for advanced users who want to experience the latest-and-greatest features and improvements. They give website, extension, and add-on developers time to test their code on upcoming versions before they are deployed to end users. The early channels are not recommended for typical end users because they can have stability issues, due to being under active development. The rest of this paper assumes a positive correlation between pre-release channels use and technical ability. While this matches the intention of browser developers, we did not carry out any study to validate this assumption.

3 Prior Laboratory Studies

We survey prior laboratory studies of SSL and phishing warnings. The body of literature paints a grim picture of browser security warnings, but most of the warnings have since been deprecated or modified. In some cases, warnings were changed in response to these studies.

Only two studies evaluated warnings that are similar to the modern (June 2013) browser warnings that we study in this paper. Sunshine et al. and Sotirakopoulos et al. reported clickthrough rates of 55% to 80% for the Firefox 3 and 3.5 SSL warnings, which are similar but not identical to the current Firefox SSL warning [30, 31]. However, Sotirakopoulos et al. concluded that laboratory biases had inflated both studies’ clickthrough rates [30].

3.1 SSL Warnings

SSL warnings are the most studied type of browser warning. Usability researchers have evaluated SSL warnings in both SSL-specific studies and phishing studies because SSL warnings and passive indicators were once viewed as a way to identify phishing attacks.²

Dhamija et al. performed the first laboratory study of SSL warnings in 2006. They challenged 22 study participants to differentiate between phishing and legitimate websites in Mozilla Firefox 1.0.1 [11]. In this version, the warning was a modal dialog that allowed the user to permanently accept, temporarily accept, or reject the certificate. When viewing the last test website, participants encountered an SSL warning. The researchers reported that 15 of their 22 subjects (68%) quickly clicked through the warning without reading it. Only one user was later able to tell the researchers what the warning had said. The authors considered the clickthrough rate of 68% a conservative lower bound because participants knew that they should be looking for security indicators.

In 2007, Schechter et al. studied user reactions to Internet Explorer 7’s SSL warning, which is the same one-click interstitial that is present in all subsequent versions of Internet Explorer [27]. Participants encountered the warning while logging into a bank website to look up information. The researchers were aware of ecological validity concerns with laboratory studies and split their participants into three groups: participants who entered their own credentials, a role-playing group that entered fake passwords, and a security-primed role-playing group that entered fake passwords. Overall, 53% of the total 57 participants clicked through. However, only 36% of the non-role-playing group clicked through. The difference between the role-playing participants and non-role-playing partic-

²There is evidence that modern phishing sites can have valid SSL certificates [24].

ipants was statistically significant, illustrating one challenge of experiments in artificial environments.

Sunshine et al. performed multiple studies of SSL warnings in 2009 [31]. First, they conducted an online survey. They asked 409 people about Firefox 2, Firefox 3, and Internet Explorer 7 warnings. Firefox 2 had a modal dialog like Firefox 1.0.1, and Firefox 3’s warning is similar but not identical to the current Firefox warning. Less than half of respondents said they would continue to the website after seeing the warning. As a follow-up, Sunshine et al. also conducted a laboratory study that exposed 100 participants to SSL warnings while completing information lookup tasks. The clickthrough rates were 90%, 55%, and 90% when participants tried to access their bank websites in Firefox 2, Firefox 3, and Internet Explorer 7, respectively. The clickthrough rates increased to 95%, 60%, and 100% when participants saw an SSL warning while trying to visit the university library website.

Sotirakopoulos et al. replicated Sunshine’s laboratory SSL study with a more representative population sample [30]. In their study, 80% of participants using Firefox 3.5 and 72% of participants using Internet Explorer 7 clicked through an SSL warning on their bank website. More than 40% of their participants said that the laboratory environment had influenced them to click through the warnings, either because they felt safe in the study environment or were trying to complete the experimental task. Sotirakopoulos et al. concluded that the laboratory environment biased their results, and they suspect that these biases are also present in similar laboratory studies.

Bravo-Lillo et al. interviewed people about an SSL warning from an unspecified browser [5]. They asked 20 participants about the purpose of the warning, what would happen if a friend were to click through, and whether a friend should click through the warning. Participants were separated into “advanced” and “novice” browser users. “Advanced” participants said they would not click through an SSL warning on a bank website, but “novice” participants said they would.

Passive Indicators. Some studies focused on passive SSL indicators, which non-interruptively show the status of the HTTP(S) connection in the browser UI. Although browsers still have passive SSL indicators, interruptive SSL and phishing warnings are now the primary tool for communicating security information to users.

Friedman et al. asked participants whether screenshots of websites depicted secure connections; many participants could not reliably determine whether a connection was secure [15]. Whalen and Inkpen used eye-tracking software to determine that none of their 16 participants looked at the lock or key icon in the URL bar, HTTP(S) status in the URL bar, or the SSL certificate when asked to browse websites “normally” [34]. Some browsers modify

the lock icon or color of the URL bar to tell the user when a website has an Extended Validation (EV) SSL certificate. Jackson et al. asked 27 study subjects to classify 12 websites as either phishing or legitimate sites, but the EV certificates did not help subjects identify the phishing sites [19]. In a follow-up study, Sobey et al. found that none of their 28 subjects clicked on the EV indicators, and the presence of EV indicators did not affect decision-making [29]. Similarly, Biddle et al. found that study participants did not understand Internet Explorer’s certificate summaries [3].

In 2012, a Google Chrome engineer mentioned high clickthrough rates for SSL warnings on his blog [20]. We expand on this with a more accurate and detailed view of SSL clickthrough rates in Google Chrome.

3.2 Phishing Warnings

Phishing warnings in contemporary browsers are active, interstitial warnings; in the past, they have been passive indicators in toolbars. Researchers have studied whether they are effective at preventing people from entering their credentials into phishing websites.

Wu et al. studied both interstitial and passive phishing warnings [35]. Neither of the warnings that they evaluated are currently in use in browsers. First, they launched phishing attacks on 30 participants. The participants role-played during the experiment while using security toolbars that display passive phishing warnings. Despite the toolbars, at least one attack fooled 20 out of 30 participants. In their next experiment, they asked 10 study participants to perform tasks on PayPal and a shopping wish list website; they injected modal phishing warnings into the websites. None of the subjects entered the credentials into the PayPal site, but the attack on the wish list site fooled 4 subjects. The authors do not report the warning clickthrough rates.

Egelman et al. subjected 60 people to simulated phishing attacks in Internet Explorer 7 or Mozilla Firefox 2.0 [13]. Firefox 2.0 had a modal phishing dialog that is not comparable to the current Mozilla Firefox phishing dialog, and Internet Explorer had both passive and active warnings. Participants believed that they were taking part in a laboratory study about shopping. The researchers asked participants to check their e-mail, which contained both legitimate shopping confirmation e-mails and similar spear phishing e-mails sent by the researchers. Users who clicked on the links in the phishing e-mails saw a phishing warning. Participants who saw Mozilla Firefox’s active warning, Internet Explorer’s active warning, or Internet Explorer’s passive warning were phished 0%, 45%, and 90% of the time, respectively. The clickthrough rates were an unspecified superset of the rates at which people fell for the phishing attacks.

3.3 Malware Download Warnings

Google Chrome and Microsoft Internet Explorer also display non-blocking warning dialogs when users attempt to download malicious executables. In a blog post, a Microsoft employee stated that the clickthrough rate for Internet Explorer’s SmartScreen warning was under 5% [16]. We did not study this warning for Google Chrome, and Mozilla Firefox does not have this warning.

4 Methodology

We rely on the telemetry features implemented in Mozilla Firefox and Google Chrome to measure clickthrough rates *in situ*. Telemetry is a mechanism for browser vendors to collect pseudonymous data from end users who opt in to statistics reporting. Google Chrome and Mozilla Firefox use similar telemetry platforms.

4.1 Measuring Clickthrough Rates

We implemented metrics in both browsers to count the number of times that a user sees, clicks through, or leaves a malware, phishing, or SSL warning. Based on this data, we can calculate clickthrough rates for each warning type. As discussed in Section 2, we report only the clickthrough rates for warnings that the user can bypass. We measured the prevalence of non-bypassable warnings separately. To supplement the clickthrough rates, we recorded whether users clicked on links like “Help me understand,” “View,” or “Technical Details.”

Bypassing some warnings takes multiple clicks, and our clickthrough rates for these warnings represent the number of users who completed all of the steps to proceed to the page. For Mozilla Firefox’s SSL warning (which takes three clicks to proceed), we recorded how often users perform two intermediate clicks (on “Add Exception” or “Confirm Security Exception”) as well as the overall clickthrough rate.

We also measured how often users encounter and click through specific SSL errors. In addition to the overall clickthrough rates for the warnings, we collected clickthrough data for each type of Mozilla Firefox SSL error and the three most common Google Chrome SSL errors.

Our Mozilla Firefox data set does not allow us to track specific telemetry participants. In Google Chrome, we can correlate warning impressions with pseudonymous browser client IDs; however, the sample size for most individual users is too small to draw conclusions. We therefore report the results of measurements aggregated across all users unless otherwise specified. The telemetry frameworks do not provide us with any personal or demographic information except for the operating system and browser version for each warning impression.

4.2 Measuring Time Spent on Warnings

We also used the Google Chrome telemetry framework to observe how much time Google Chrome users spent on SSL warnings. Timing began as soon as an SSL warning came to the foreground in a tab. In particular,

- We recorded the time spent on a warning and associated it with the outcome (click through or leave).
- We recorded the time spent on a warning and associated it with the error type, if it was one of the three most common error types (untrusted authority, name mismatch, and expired certificate).

Together, these correspond to five timing measurements (two for outcome and three for error type). For scalability, the telemetry mechanism in Google Chrome only allows timing measurements in discrete buckets. As a result, our analysis also treats time as a discrete, ordinal variable.

We used log-scaled bucket sizes (e.g., the first bucket size is 45ms but the last is 90,279ms) with 50 buckets, ranging from 0ms to 1,200,000ms, for the two outcome histograms. The three error type histograms had 75 buckets each, ranging from 0ms to 900,000ms. We used more buckets for the error histograms because we anticipated that they would be more similar to each other.

4.3 Ethics

We collected data from users who participate in their browsers’ broad, unpaid user metrics programs. At first run of a browser, the browser asks the user to share usage data. If the user consents, the browser collects data on performance, features, and stability. In some pre-release developer channels, data collection is enabled by default. The browser periodically sends this pseudonymous data over SSL to the central Mozilla or Google servers for analysis. The servers see the IP addresses of clients by necessity, but they are not attached to telemetry data. All telemetry data is subject to strict privacy policies and participants can opt out by changing their settings [7, 23]. Multiple Google Chrome committers and Mozilla Firefox contributors reviewed the data collection code to ensure that the metrics did not collect any private data.

This work is not considered human subjects research by UC Berkeley because the student did not have access to database identifiers or personally identifying information.

4.4 Data Collection

Collection Period. Google Chrome’s malware and phishing measurement code was in place in Chrome 24 prior to our work, and our SSL measurement code was added to Google Chrome 25. The Google Chrome data in this

paper was collected April 28 - May 31, 2013. Our Mozilla Firefox measurement code was added to Firefox 17, and a bug in the SSL measurement code was fixed in Firefox 23. The data on the Firefox malware warning, phishing warning, and SSL “Add Exception” dialog was collected May 1-31, 2013. The data on Firefox SSL warnings was collected June 1 - July 5, 2013, as the Firefox 23 fix progressed through the various release channels.

Sample Sizes. In Google Chrome, we recorded 6,040,082 malware warning impressions, 386,350 phishing warning impressions, and 16,704,666 SSL warning impressions. In Mozilla Firefox, we recorded 2,163,866 malware warning impressions, 100,004 phishing warning impressions, and 10,976 SSL warning impressions. Appendix A further breaks down these sample sizes by OS and channel.

Number of Users. For Mozilla Firefox, we recorded warning impressions from the approximately 1% of Firefox users who opt in to share data with Mozilla via telemetry. In Google Chrome, we observed malware, phishing, and SSL warning impressions on 2,148,026; 204,462; and 4,491,767 clients (i.e., browser installs), respectively.

4.5 Method Limitations

Private Data. Due to privacy constraints, we could not collect information about users’ personal demographics or browsing habits. Consequently, we cannot measure whether user behavior differs based on personal characteristics, the target site, or the source of the link to the site. We also cannot identify SSL false positives due to captive portals, network proxies, or server misconfigurations.

Sampling Bias. The participants in our field study are not a random population sample. Our study only represents users who opt in to browser telemetry programs. This might present a bias. The users who volunteered might be more likely to click through dialogs and less concerned about privacy. Thus, the clickthrough rates we measure could be higher than population-wide rates. Given that most of our observed rates are low, this bias augments our claim that clickthrough rates are lower than anticipated.

Overrepresentation. We present clickthrough rates across all warnings shown to all users. A subset of users could potentially be overrepresented in our analysis. Within the Google Chrome data set, we identified and removed a small number of overrepresented clients who we believe are either crawlers or malware researchers. We were unable to remove individual clients from the Mozilla Firefox set, but we do not believe this represents a bias because we know that the overrepresented clients in Chrome still contributed fewer than 1% of warning impressions. Some clients experienced multiple types of warning impressions; we investigated this in Chrome

and found that the clickthrough rates do not differ if we remove non-independent clients. Our large sample sizes and small critical value ($\alpha = 0.001$) should further ameliorate these concerns.

Frames. Our original measurement for Mozilla Firefox did not differentiate between warnings shown in top-level frames (i.e., warnings that fill the whole tab) and warnings shown in iframes. In contrast, Google Chrome always shows malware and phishing warnings in the top-level frame and does not render any warning type in iframes. Since users might not notice warnings in iframes, the two metrics are not necessarily directly comparable.

Upon discovering this issue, we modified our Firefox measurement implementation to take frame level into account. Our new implementation is not available to all Firefox users yet, but we have data for recent pre-release channels. For malware and phishing warning impressions collected from the beta channel, the clickthrough rate for the top-level frame is within two percentage points of the overall clickthrough rate. This is due to the relative infrequency of malware and phishing warnings in iframes and the low overall clickthrough rate. Since the frame level does not make a notable difference for malware and phishing warnings, we present the overall rates (including both top-level frames and iframes) for the full sample sizes in Section 5.1. The difference is more important for SSL warnings: the clickthrough rate for top-level frames is 28.7 percentage points higher than the overall clickthrough rate of 4.3%. Consequently, Section 5.2 presents only the top-level frame rate for SSL warnings, although it limits our sample to pre-release users.

5 Clickthrough Rates

We present the clickthrough data from our measurement study. Section 5.1 discusses malware and phishing warnings together because they share a visual appearance. We then present rates for SSL warnings in Section 5.2.

5.1 Malware and Phishing Warnings

The clickthrough rates for malware warnings were 7.2% and 23.2% in stable versions of Mozilla Firefox and Google Chrome, respectively. For phishing warnings, we found clickthrough rates of 9.1% and 18.0%. In this section, we discuss the effects of warning type, demographics, and browser on the clickthrough rates.

5.1.1 Malware Rates by Date

The malware warning clickthrough rates for Google Chrome vary widely by date. We have observed clickthrough rates ranging from 11.2% to 24.9%, depending

Operating System	Malware		Phishing	
	Firefox	Chrome	Firefox	Chrome
Windows	7.1%	23.5%	8.9%	17.9%
MacOS	11.2%	16.6%	12.5%	17.0%
Linux	18.2%	13.9%	34.8%	31.0%

Table 1: User operating system vs. clickthrough rates for malware and phishing warnings. The data comes from stable (i.e., release) versions.

Channel	Malware		Phishing	
	Firefox	Chrome	Firefox	Chrome
Stable	7.2%	23.2%	9.1%	18.0%
Beta	8.7%	22.0%	11.2%	28.1%
Dev	9.4%	28.1%	11.6%	22.0%
Nightly	7.1%	54.8%	25.9%	20.4%

Table 2: Release channel vs. clickthrough rates for malware and phishing warnings, for all operating systems.

on the week, since the current version of the warning was released in August 2012. In contrast, the Mozilla Firefox malware warning clickthrough rate across weeks stays within one percentage point of the month-long average. We did not observe similar temporal variations for phishing or SSL warnings.

Recall from Section 2.1 that Google Chrome and Mozilla Firefox’s malware warnings differ with respect to secondary resources: Google Chrome shows an interstitial malware warning if a website includes secondary resources from a domain on the Safe Browsing list, whereas Mozilla Firefox silently blocks the resource. We believe that this makes Google Chrome’s malware clickthrough rates more sensitive to the contents of the Safe Browsing list. For example, consider the case where a well-known website accidentally loads an advertisement from a malicious domain. Google Chrome would show a warning, which users might not believe because they trust the website. Mozilla Firefox users would not see any warning. Furthermore, Chrome phishing warnings are less likely to be due to secondary resources, and that warning’s clickthrough rates do not vary much by time.

5.1.2 Malware/Phishing Rates by Warning Type

In Mozilla Firefox, we find a significantly higher clickthrough rate for phishing warnings than malware warnings (χ^2 test: $p(1) < 0.0001$). This behavior is rational: a malware website can infect the user’s computer without any action on the user’s part, but a phishing website can only cause harm by tricking the user at a later point in time. Mozilla Firefox makes this priority ordering explicit by choosing to display the malware warning if a website

is listed as both malware and phishing.³ However, the practical difference is small: 7.2% vs. 9.1%.

In Google Chrome, the average malware clickthrough rate is higher than the phishing clickthrough rate. However, the malware clickthrough rate fluctuates widely (Section 5.1.1); the malware clickthrough rate is sometimes lower than the phishing clickthrough rate.

5.1.3 Malware/Phishing Rates by Demographics

We consider whether users of different operating systems and browser release channels react differently to warnings. As Table 1 depicts, Linux users have significantly higher clickthrough rates than Mac and Windows users combined for the Firefox malware warning, Firefox phishing warning, and Chrome phishing warning (χ^2 tests: $p(1) < 0.0001$). While the low prevalence of malware for Linux could explain the higher clickthrough rates for the Firefox malware warning, use of Linux does not provide any additional protection against phishing attacks. The Chrome malware warning does not follow the same pattern: Windows users have a significantly higher clickthrough rate (χ^2 tests: $p(1) < 0.0001$).

We also see differences between software release channels (Table 2). Nightly users click through Google Chrome malware and Firefox phishing warnings at much higher rates than stable users, although they click through Firefox malware and Google Chrome phishing warnings at approximately the same rates.

In several cases, Linux users and early adopters click through malware and phishing warnings at higher rates. One possible explanation is that a greater degree of technical skill – as indicated by use of Linux or early-adopter versions of browsers – corresponds to reduced risk aversion and an increased willingness to click through warnings. This does not hold true for all categories and warnings (e.g., nightly and stable users click through the Firefox malware warning at the same rate), suggesting the need for further study.

5.1.4 Malware/Phishing Rates by Browser

Google Chrome stable users click through phishing warnings more often than Mozilla Firefox stable users. This holds true even when we account for differences in how the browsers treat iframes (Section 4.5). Mozilla Firefox’s beta channel users still click through warnings at a lower rate when we exclude iframes: 9.6% for malware warnings, and 10.8% for phishing warnings.

One possibility is that Mozilla Firefox’s warnings are more frightening or more convincing. Another possi-

³Google Chrome will display both warnings. To preserve independence, our measurement does not include any warnings with both phishing and malware error messages. Dual messages are infrequent.

bility is that the browsers have different demographics with different levels of risk tolerance, which is reflected in their clickthrough rates. There might be differences in technical education, gender, socioeconomic status, or other factors that we cannot account for in this study. In support of this theory, we find that differences between the browsers do not hold steady across operating systems or channels. The gap between the browsers narrows or reverses for some categories of users, such as Linux users and nightly release users.

5.2 SSL Warnings

The clickthrough rates for SSL warnings were 33.0% and 70.2% for Mozilla Firefox (beta channel) and Google Chrome (stable channel), respectively.

5.2.1 SSL Rates by Demographic

In Section 5.1, we observed that malware and phishing clickthrough rates differed across operating systems and channels. For SSL, the differences are less pronounced.

As with the malware and phishing warnings, nightly users click through SSL warnings at a higher rate for both Firefox and Chrome (χ^2 tests: $p < 0.0001$).

The effect of users’ operating systems on SSL clickthrough rates differs for the two browsers. In Firefox, Linux users are much more likely to click through SSL warnings than Windows and Mac users combined (χ^2 test: $p < 0.0001$), although it is worth noting that the Firefox Linux sample size is quite small (58). In Chrome, Windows users are very slightly more likely to click through SSL warnings than Linux and Mac users combined (χ^2 test: $p < 0.0001$).

5.2.2 SSL Rates by Browser

We find a large difference between the Mozilla Firefox and Google Chrome clickthrough rates: Google Chrome users are 2.1 times more likely to click through an SSL warning than Mozilla Firefox users. We explore five possible causes.

Number of Clicks. Google Chrome users click one button to dismiss an SSL warning, but Mozilla Firefox users need to click three buttons. It is possible that the additional clicks deter people from clicking through. However, we do not believe this is the cause of the rate gap.

First, the number of clicks does not appear to affect the clickthrough rates for malware and phishing warnings. Mozilla Firefox’s malware and phishing warnings require one click to proceed, whereas Google Chrome’s malware and phishing warnings require two. The Google Chrome malware and phishing warnings with two clicks do not have lower clickthrough rates than the Mozilla Firefox warnings with one click. Second, as we discuss in Section 5.2.3, 84% of users who perform the first two

Operating System	SSL Warnings	
	Firefox	Chrome
Windows	32.5%	71.1%
MacOS	39.3%	68.8%
Linux	58.7%	64.2%
Android	NC	64.6%

Table 3: User operating system vs. clickthrough rates for SSL warnings. The Google Chrome data is from the stable channel, and the Mozilla Firefox data is from the beta channel.

Channel	SSL Warnings	
	Firefox	Chrome
Release	NC	70.2%
Beta	32.2%	73.3%
Dev	35.0%	75.9%
Nightly	43.0%	74.0%

Table 4: Channel vs. clickthrough rates for SSL warnings.

clicks in Mozilla Firefox also perform the third. This indicates that the extra click is not a determining decision point. Unfortunately, we do not have data on the difference between the first and second clicks.

Warning Appearance. The two warnings differ in several ways. Mozilla Firefox’s warning includes an image of a policeman and uses the word “untrusted” in the title. These differences likely contribute to the rate gap. However, we do not think warning appearance is the sole or primary factor; the browsers’ malware and phishing warnings also differ, but there is only about a 10% difference between browsers for these warnings.

Certificate Pinning. Google Chrome ships with a list of “pinned” certificates and preloaded HTTP Strict Transport Security (HSTS) sites. Users cannot click through SSL warnings on sites protected by these features. Certificate pinning and HSTS cover some websites with important private data such as Google, PayPal, and Twitter [8]. In contrast, Mozilla Firefox does not come with many preloaded “pinned” certificates or any pre-specified HSTS sites. As a result, Chrome shows more non-bypassable warnings: our field study found that 20% of all Google Chrome SSL warning impressions are non-bypassable, as compared to 1% for Mozilla Firefox.

Based on this, we know that Mozilla Firefox users see more warnings for several critical websites. If we assume that users are less likely to click through SSL warnings on these critical websites, then it follows that Mozilla Firefox’s clickthrough rate will be lower. This potential bias could account for up to 15 points of the 37-point gap between the two clickthrough rates, if we were to assume that Google Chrome users would never click through SSL errors on critical websites if given the chance.

Remembering Exceptions. Due to the “permanently store this exception” feature in Mozilla Firefox, Mozilla Firefox users see SSL warnings only for websites without saved exceptions. This means that Mozilla Firefox users might ultimately interact with websites with SSL errors at the same rate as Google Chrome users despite having lower clickthrough rates. For example, imagine a user that encounters two websites with erroneous SSL configuration: she leaves the first after seeing a warning, but visits the second website nine times despite the warning. This user would have a 50% clickthrough rate in Mozilla Firefox but a 90% clickthrough rate in Google Chrome, despite visiting the second website at the same rate.

We did not measure how often people revisit websites with SSL errors. However, we suspect that people do repeatedly visit sites with warnings (e.g., a favorite site with a self-signed certificate). If future work were to confirm this, there could be two implications. First, if users are repeatedly visiting the same websites with errors, the errors are likely false positives; this would mean that the lack of an exception-storing mechanism noticeably raises the false positive rate in Google Chrome. Second, warning fatigue could be a factor. If Google Chrome users are exposed to more SSL warnings because they cannot save exceptions, they might pay less attention to each warning that they encounter.

Demographics. It’s possible that the browsers have different demographics with different levels of risk tolerance. However, this factor likely only accounts for a few percentage points because the same demographic effect applies to malware and phishing warnings, and the difference between browsers for malware and phishing warnings is much smaller.

5.2.3 SSL Rates by Certificate Error Type

To gain insight into the factors that drive clickthrough rates, we study whether the particular certificate error affects user behavior.

Google Chrome. Google Chrome’s SSL warning includes a short explanation of the particular error, and clicking on “Help me understand” will open a more-detailed explanation. In case a certificate has multiple errors, Google Chrome only shows the first error out of untrusted issuer error, name mismatch error, and certificate expiration error, respectively.

Table 5 presents the clickthrough rates by error types for Google Chrome. If Google Chrome users are paying attention to and understanding the warnings, one would expect different clickthrough rates based on the warning types. We find a 24.4-point difference between the clickthrough rates for untrusted issuer errors and expired certificate errors. One explanation could be that untrusted issuer

Certificate Error	Percentage of Total	Clickthrough Rate
Untrusted Issuer	56.0%	81.8%
Name Mismatch	25.0%	62.8%
Expired	17.6%	57.4%
Other Error	1.4%	—
All Error Types	100.0%	70.2%

Table 5: Prevalence and clickthrough rates of error types for the Google Chrome SSL warning. Google Chrome only displays the most critical warning; we list the error types in order, with untrusted issuer errors as the most critical. Data is for the stable channel across all operating systems.

errors appear on unimportant sites, leading to higher clickthrough rates without user attention or comprehension; however, the Mozilla Firefox data suggests otherwise. An alternative explanation could be that expired certificates, which often occur for websites with previously valid certificates [1], surprise the user. In contrast, untrusted certificate errors always occur for a website and conform with expectations.

Mozilla Firefox. Mozilla Firefox’s SSL warning does not inform the user about the particular SSL error by default.⁴ Instead, the secondary “Add Exception” dialog presents *all* errors in the SSL certificate. The user must confirm this dialog to proceed.

Table 6 presents the rates at which users confirm the “Add Exception” dialog in Mozilla Firefox. The error types do not greatly influence the exception confirmation rate. This indicates that the “Add Exception” dialog does not do an adequate job of explaining particular error categories and their meaning to the users. Thus, users ignore the categories and click through errors at the same rate. This finding also suggests that the differences in clickthrough rates across error types in Google Chrome cannot be attributed to untrusted issuer errors corresponding to unimportant websites; if that were the case, we would expect to see the same phenomenon in Firefox.

Error Prevalence. The frequency of error types encountered by users in our field study also indicates the base rate of SSL errors on the web. Our Google Chrome data contradicts a previous network telemetry study, which suggested that untrusted issuer errors correspond to 80% of certificate errors seen on the wire [18]. Also, Google Chrome users see fewer untrusted issuer errors than Mozilla Firefox users; this may be because Mozilla Firefox users are more likely to click on the “Add Exception” dialog for untrusted issuer errors. Recall that we collect the Mozilla Firefox error type statistics only after a user clicks on the “Add Exception” button.

⁴This information is available under the “Technical details” link, but our measurements indicate that it is rarely opened (Section 5.2.4).

Certificate Error	Percentage of Total	Confirmation Rate
Untrusted Issuer	38%	87.1%
Untrusted and Name Mismatch	26.4%	87.9%
Name Mismatch	15.7%	80.3%
Expired	10.2%	80.7%
Expired, Untrusted and Name Mismatch	4.7%	87.6%
Expired and Untrusted	4.1%	83.6%
Expired and Name Mismatch	0.7%	85.2%
None of the above	<0.1%	77.9%
All error types	100.0%	85.4%

Table 6: Prevalence and confirmation rates of error types for the Mozilla Firefox “Add Exception” dialog. The confirmation rate measures the percentage of users who click on “Confirm Security Exception” (Figure 5). The Mozilla Firefox dialog lists all the errors that occur for a certificate. Data is for the release channel across all operating systems; we did not need to limit it to the beta channel because frame level issues do not affect clickthrough rates inside the “Add Exception” dialog.

The high frequency of untrusted issuer errors highlights the usability benefits of “network view” SSL certificate verification systems like Perspectives and Convergence [10,33], which do not need certificates from trusted authorities. All of the untrusted certificate warnings—between 38% and 56% of the total—would disappear. Warnings with other errors in addition to an untrusted certificate error would remain. Nonetheless, our study also shows that these mechanisms are not a panacea: name mismatch errors constitute a large fraction of errors, and new systems like Perspectives and Convergence still perform this check.⁵

5.2.4 Additional SSL Metrics

We collected several additional metrics to complement the overall clickthrough rates.

More Information. Google Chrome and Mozilla Firefox both place additional information about the warning behind links. However, very few users took the opportunity to view this extra information. The “Help me understand” button was clicked during 1.6% of Google Chrome SSL warning impressions. For Mozilla Firefox warnings, 0 users clicked on “Technical Details,” and 3% of viewers of the “Add Exception” dialog clicked on “View Certificate.” This additional content therefore has no meaningful impact on the overall clickthrough rates.

Add Exception Cancellation. Not all Mozilla Firefox

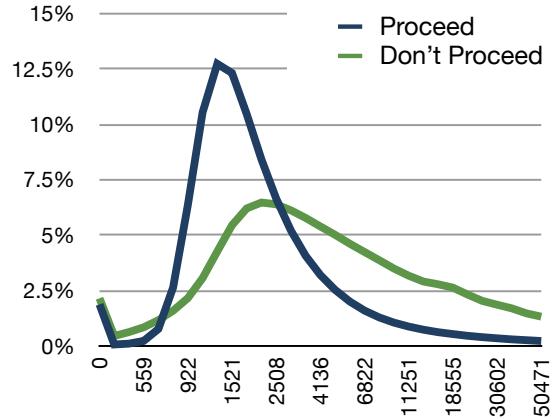


Figure 6: Google Chrome SSL clickthrough times (ms), by outcome. The graph shows the percent of warning impressions that fall in each timing bucket. The *x*-axis increases logarithmically, and we cut off the distribution at 90% due to the long tail.

users proceed to the page after opening the “Add Exception” dialog: 14.6% of the time that a dialog is opened, the user cancels the exception. These occurrences indicate that at least a minority of users consider the text in the dialog before confirming the exception.

Remember Exception. By default, the “Remember Exception” checkbox is checked in the Mozilla Firefox “Add Exception” dialog. Our measurements found that 21.3% of the time that the dialog is opened, the user un-ticks the checkbox. We hypothesize that these users are still wary of the website even if they choose to proceed.

6 Time Spent On SSL Warnings

In addition to MITM attacks, SSL warnings can occur due to server misconfigurations. Previous work found that 20% of the thousand most popular SSL sites triggered a false warning due to such misconfigurations [31]. Consequently, it may be safe and rational to click through such false warnings. The prevalence of a large number of such false warnings can potentially train users to consider *all* SSL warnings false alarms and click through them without considering the context.

In order to determine whether users examine SSL warnings before making a decision, we measured how much time people spent on SSL warning pages. In this section, we compare the click times by outcome (clickthrough or leave) and error type to gain insight into user attention. Our timing data is for all operating systems and channels.

6.1 Time by Outcome

Figure 6 presents the click times for different outcomes. Users who leave spend more time on the warning than

⁵Convergence does not check the certificate issuer, relying on network views instead. However, it performs name checks [10].

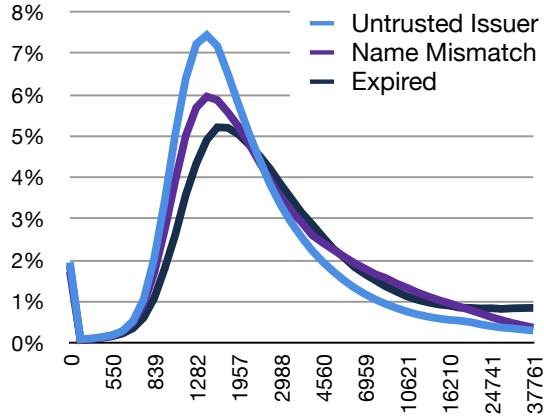


Figure 7: Google Chrome SSL clickthrough times (ms), by error type. The graph shows the percent of warning impressions that fall in each timing bucket. The *x*-axis increases logarithmically, and we cut off the distribution at 90% due to the long tail.

users who click through and proceed to the page. 47% of users who clicked through the warning made the decision within 1.5s, whereas 47% of users who left the page did so within 3.5s. We interpret this to mean that users who click through the warning often do so after less consideration.

6.2 Time by Error Type

Figure 7 depicts the click times for three error types (untrusted authority, name mismatch, and expired certificate errors). Users clicked through 49% of untrusted issuer warning impressions within 1.7s, but clicked through 50% of name and date errors within 2.2s and 2.7s, respectively. We believe that this data is indicative of warning fatigue: users click through more-frequent errors more quickly. The frequency and clickthrough rate of each error type (as reported in Section 5.2) are inversely correlated with that error type’s timing variance and mode (Figure 7).

7 Implications

Our primary finding is that browser security warnings *can* be effective security mechanisms in practice, but their effectiveness varies widely. This should motivate more attention to improving security warnings. In this section, we summarize our findings and their implications, present suggestions for warning designers, and make recommendations for future warning studies.

7.1 Warning Effectiveness

7.1.1 Clickthrough Rates

Popular opinion holds that browser security warnings are ineffective. However, our study demonstrates that browser

security warnings can be highly effective at preventing users from visiting websites: as few as a tenth of users click through Firefox’s malware and phishing warnings. We consider these warnings very successful.

We found clickthrough rates of 18.0% and 23.2% for Google Chrome’s phishing and malware warnings, respectively, and 31.6% for Firefox’s SSL warning. These warnings prevent 70% (or more) of attempted visits to potentially dangerous websites. Although these warnings could be improved, we likewise consider these warnings successful at persuading and protecting users.

Google Chrome’s SSL warning had a clickthrough rate of 70.2%. Such a high clickthrough rate is undesirable: either users are not heeding valid warnings, or the browser is annoying users with invalid warnings and possibly causing warning fatigue. Our positive findings for the other warnings demonstrate that this warning has the potential for improvement. We hope that this study motivates further studies to determine and address the cause of its higher clickthrough rate. We plan to test an exception-remembering feature to investigate the influence of repeat exposures to warnings. At Google, we have also begun a series of A/B tests in the field to measure the impact of a number of improvements.

7.1.2 User Attention

Although we did not directly study user attention, two results of our study suggest that at least a minority of users pay attention to browser security warnings.

- There is a 24.4-point difference between the click-through rates for untrusted issuer errors (81.8%) and expired certificate errors (57.4%) in Google Chrome.
- 21.3% of the time that Mozilla Firefox users viewed the “Add Exception” dialog, they un-checked the default “Permanently store this exception” option.

These results contradict the stereotype of wholly oblivious users with no interest in security.

7.2 Comparison with Prior Research

As Bravo-Lillo et al. wrote [5]:

Evidence from experimental studies indicates that most people don’t read computer warnings, don’t understand them, or simply don’t heed them, even when the situation is clearly hazardous.

In contrast, a majority of users heeded five of the six types of browser warnings that we studied. This section explores why our results differ from prior research.

Browser Changes. Most prior browser research was conducted between 2002 and 2009. Browsers were rapidly

changing during this time period; some changes were directly motivated by published user studies. Notably, passive indicators are no longer considered primary security tools, and phishing toolbars have been replaced with browser-provided, full-page interstitial warnings. As a result, studies of passive indicators and phishing toolbars no longer represent the state of modern browser technology.

Two studies tested an older version of the Mozilla Firefox SSL warning, in which the warning was a modal (instead of full-page) dialog. Dhamija et al. observed a 68% clickthrough rate, and Sunshine et al. recorded clickthrough rates of 90%-95% depending on the type of page [11, 31]. The change in warning design could be responsible for our lower observed clickthrough rates.

Ecological Invalidity. Sunshine et al. and Sotirakopoulos et al. recorded 55%-60% and 80% clickthrough rates, respectively, for a slightly outdated version of the Mozilla Firefox SSL warning [30, 31]. They evaluated the Firefox 3 and 3.5 warnings, which had the same layout and appearance as the current (Firefox 4+) warning but with different wording. It's possible that changes in wording caused clickthrough rates to drop from 55%-80% to 33.0%. However, during an exit survey, 46% of Sotirakopoulos's subjects said they clicked through the warning because they either felt safe in the laboratory environment or wanted to complete the task [30]. Since their study methodology was intentionally similar to the Sunshine study, Sotirakopoulos et al. concluded that both studies suffered from biases that raised their clickthrough rates [30]. We therefore attribute some of the discrepancy between our field study data and these two laboratory studies to the difficulty of establishing ecological validity in a laboratory environment.

In light of this, we recommend a renewed emphasis on field techniques for running and confirming user studies of warnings. Although we used in-browser telemetry, there are other ways of obtaining field data. For example, experience sampling is a field study methodology that asks participants to periodically answer questions about a topic [2, 6, 9, 28]. Researchers could install a browser extension on participants' computers to observe their responses to normally occurring warnings and display a survey after each warning. This technique allows researchers to collect data about participants' emotions, comprehension, and demographics. Participants may become more cautious or attentive to warnings if the purpose of the study is apparent, so researchers could obscure the purpose by surveying subjects about other browser topics. Network-based field measurements also provide an alternative methodology with high ecological validity. A network monitor could maintain its own copy of the Safe Browsing list and identify users who click through warnings. If the monitor can associate network flows

with specific demographics (e.g., students), it can help understand the impact of these factors on user behavior. Similar studies could help understand SSL clickthrough rates; recent work addressed how to reproduce certificate validation at the network monitor [1].

7.3 Demographics

We found that clickthrough rates differ by operating system and browser channel. Our findings suggest that higher technical skill (as indicated by use of Linux and pre-release channels) may predispose users to click through some types of warnings. We recommend further investigation of user demographics and their impact on user behavior. Large-scale demographic studies might uncover additional demographic factors that we were unable to study with our methodology. If so, can warning design address and overcome those demographic differences?

Technically advanced users might feel more confident in the security of their computers, be more curious about blocked websites, or feel patronized by warnings. Studies of these users could help improve their warning responses.

7.4 Number of Clicks

Our data suggests that the amount of effort (i.e., number of clicks) required to bypass a warning does not always have a large impact on user behavior. To bypass Google Chrome’s malware and phishing warnings, the user must click twice: once on a small “Advanced” link, and then again to “proceed.” Despite the hidden button, users click through Google Chrome’s malware/phishing warning at a higher rate than Mozilla Firefox’s simpler warning. Furthermore, 84% of users who open Mozilla Firefox’s “Add Exception” dialog proceed through it.

We find this result surprising. Common wisdom in e-commerce holds that extra clicks decrease clickthrough rates (hence, one-click shopping) [12, 32]. Google Chrome’s warning designers introduced the extra step in the malware/phishing warning because they expected it to serve as a strong deterrent. One possible explanation is that users make a single cognitive decision when faced with a warning. The decision might be based on the URL, warning appearance, or warning message. Once the user has decided to proceed, additional clicks or information is unlikely to change his or her decision.

Our data suggests that browser-warning designers should not rely on extra clicks to deter users. However, we did not explicitly design our study to examine the effects of multiple clicks. Future studies on multi-click warnings could shed light on user decision models and impact security warning design. It is possible that extra clicks do not serve as a deterrent until they reach some threshold of difficulty.

7.5 Warning Fatigue

We observed behavior that is consistent with the theory of warning fatigue. In Google Chrome, users click through the most common SSL error faster and more frequently

than other errors. Our findings support recent literature that has modeled user attention to security warnings as a finite resource [4] and proposed warning mechanisms based on this constraint [14].

Based on this finding, we echo the recommendation that security practitioners should limit the number of warnings that users encounter. Designers of new warning mechanisms should always perform an analysis of the number of times the system is projected to raise a warning, and security practitioners should consider the effects that warning architectures have on warning fatigue.

7.6 “More Information”

Users rarely click on the explanatory links such as “More Information” or “Learn More” (Section 5.2.4). Designers who utilize such links should ensure that they do not hide a detail that is important to the decision-making process.

Mozilla Firefox places information about SSL errors under “Technical Details” and in the “Add Exception” dialog instead of the primary warning. Thus, the error type has little impact on clickthrough rates. In contrast, Google Chrome places error details in the main text of its SSL warning, and the error has a large effect on user behavior. It is possible that moving this information into Mozilla Firefox’s primary warning could reduce their clickthrough rates even further for some errors.

8 Conclusion

We performed a field study with Google Chrome and Mozilla Firefox’s telemetry platforms, allowing us to collect data on 25,405,944 warning impressions. We find that browser security warnings can be successful: users clicked through fewer than a quarter of both browser’s malware and phishing warnings and a third of Mozilla Firefox’s SSL warnings. We also find clickthrough rates as high as 70.2% for Google Chrome SSL warnings, indicating that the user experience of a warning can have a tremendous impact on user behavior. However, warning effectiveness varies between demographic groups. Our findings motivate more work on browser security warnings, with particular attention paid to demographics. At Google, we have begun experimenting with new warning designs to further improve our warnings.

Acknowledgements

We thank the participants in Google and Mozilla’s telemetry programs for providing us with valuable insight into our warnings. At Google, we would like to thank Matt Mueller for setting up the malware and phishing measurements, Adam Langley for making suggestions about how to implement SSL measurements, and many others for providing insightful feedback. At Mozilla, we would like to thank Sid Stamm for his mentorship and help collecting telemetry data, Dan Veditz for gathering data from Firefox 23, Brian Smith for providing information about the telemetry mechanisms, and the Mozilla contributors who reviewed our code and helped land this telemetry [22]. We also thank David Wagner, Vern Paxson, Serge Egelman, Stuart Schechter, and the anonymous reviewers for providing feedback on drafts of the paper.

References

- [1] AKHawe, D., Amann, B., Vallentin, M., and Sommer, R. Here’s My Cert, So Trust Me, Maybe? Understanding TLS Errors on the Web. In *Proceedings of the 2013 World Wide Web Conference* (2013).
- [2] Ben Abdesselam, F., Parris, I., and Henderson, T. Mobile Experience Sampling: Reaching the Parts of Facebook Other Methods Cannot Reach. In *Privacy and Usability Methods Pow-wow* (2010).
- [3] Biddle, R., van Oorschot, P. C., Patrick, A. S., Sobey, J., and Whalen, T. Browser interfaces and extended validation SSL certificates: an empirical study. In *Proceedings of the ACM Workshop on Cloud Computing Security* (2009).
- [4] Böhme, R., and Grossklags, J. The Security Cost of Cheap User Interaction. In *Proceedings of the New Security Paradigms Workshop (NSPW)* (2011).
- [5] Bravo-Lillo, C., Cranor, L. F., Downs, J. S., and Kemanduri, S. Bridging the Gap in Computer Security Warnings: A Mental Model Approach. In *IEEE Security and Privacy* (March 2011), vol. 9.
- [6] Christensen, T., Barrett, L., Bliss-Moreau, E., Lebo, K., and Kaschub, C. A Practical Guide to Experience-Sampling Procedures. In *Journal of Happiness Studies* (2003), vol. 4.
- [7] Google Chrome Privacy Notice. <http://www.google.com/chrome/intl/en/privacy.html>.
- [8] CHROMIUM AUTHORS. HSTS Preload and Certificate Pinning List. https://src.chromium.org/viewvc/chrome/trunk/src/net/base/transport_security_state_static.json.
- [9] Consolvo, S., and Walker, M. Using the Experience Sampling Method to Evaluate Ubicomp Applications. In *Pervasive Computing* (2003).
- [10] Convergence. <http://www.convergence.io>.
- [11] DhamiJa, R., Tygar, J. D., and Hearst, M. Why phishing works. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (2006).
- [12] Dutta, R., Jarvenpaa, S., and Tomak, K. Impact of Feedback and Usability of Online Payment Processes on Consumer Decision Making. In *Proceedings of the International Conference on Information Systems* (2003).
- [13] Egelman, S., Cranor, L. F., and Hong, J. You’ve Been Warned: An Empirical Study of the Effectiveness of Web Browser Phishing Warnings. In *Proceedings of the ACM CHI Conference on Human Factors in Computing Systems* (2008).
- [14] Felt, A. P., Egelman, S., Finifter, M., Akhawe, D., and Wagner, D. How to ask for permission. In *Proceedings of the USENIX Conference on Hot Topics in Security (HotSec)* (2012).
- [15] Friedman, B., Hurley, D., Howe, D. C., Felten, E., and Nissenbaum, H. Users’ Conceptions of Web Security: A Comparative Study. In *CHI Extended Abstracts on Human Factors in Computing Systems* (2002).
- [16] Haber, J. Smartscreen application reputation in ie9, May 2011. <http://blogs.msdn.com/b/ie/archive/2011/05/17/smartscreen-174-application-reputation-in-ie9.aspx>.
- [17] Herley, C. The plight of the targeted attacker in a world of scale. In *Proceedings of the Workshop on the Economics of Information Security (WEIS)* (2010).

- [18] HOLZ, R., BRAUN, L., KAMMENHUBER, N., AND CARLE, G. The ssl landscape: a thorough analysis of the x.509 pki using active and passive measurements. In *Proceedings of the ACM SIGCOMM Internet Measurement Conference (IMC)* (2011).
- [19] JACKSON, C., SIMON, D. R., TAN, D. S., AND BARTH, A. An evaluation of extended validation and picture-in-picture phishing attacks. In *Proceedings of the Workshop on Usable Security (USEC)* (2007).
- [20] LANGLEY, A. SSL Interstitial Bypass Rates, February 2012. <http://www.imperialviolet.org/2012/07/20/sslbyprates.html>.
- [21] McGRAW, G., FELTEN, E., AND MACMICHAEL, R. *Securing Java: getting down to business with mobile code*. Wiley Computer Pub., 1999.
- [22] MOZILLA BUGZILLA. Bug 767676: Implement Security UI Telemetry. <https://bugzilla.mozilla.org/767676>.
- [23] Mozilla firefox privacy policy. <http://www.mozilla.org/en-US/legal/privacy/firefox.html#telemetry>.
- [24] NETCRAFT. Phishing on sites using ssl certificates, August 2012. <http://news.netcraft.com/archives/2012/08/22/phishing-on-sites-using-ssl-certificates.html>.
- [25] PATERIYA, P. K., AND KUMAR, S. S. Analysis of Man in the Middle Attack on SSL. *International Journal of Computer Applications* 45, 23 (2012).
- [26] PROVOS, N. Safe Browsing - Protecting Web Users for 5 Years and Counting. Google Online Security Blog. <http://googleonlinesecurity.blogspot.com/2012/06/safe-browsing-protecting-web-users-for.html>, June 2012.
- [27] SCHECHTER, S. E., DHAMIJA, R., OZMENT, A., AND FISCHER, I. The Emperor's New Security Indicators. In *Proceedings of the IEEE Symposium on Security and Privacy* (2007).
- [28] SCOLLON, C. N., KIM-PRIETO, C., AND DIENER, E. Experience Sampling: Promises and Pitfalls, Strengths and Weaknesses. In *Journal of Happiness Studies* (2003), vol. 4.
- [29] SOBEY, J., BIDDLE, R., VAN OORSCHOT, P., AND PATRICK, A. S. Exploring user reactions to new browser cues for extended validation certificates. In *Proceedings of the European Symposium on Research in Computer Security* (2008).
- [30] SOTIRAKOPOULOS, A., HAWKEY, K., AND BEZNOSOV, K. On the Challenges in Usable Security Lab Studies: Lessons Learned from Replicating a Study on SSL Warnings. In *Proceedings of the Symposium on Usable Privacy and Security* (2011).
- [31] SUNSHINE, J., EGELMAN, S., ALMUHIMEDI, H., ATRI, N., AND CRANOR, L. F. Crying Wolf: An Empirical Study of SSL Warning Effectiveness. In *Proceedings of the USENIX Security Symposium* (2009).
- [32] TILSON, R., DONG, J., MARTIN, S., AND KIEKE, E. Factors and Principles Affecting the Usability of Four E-commerce Sites. In *Our Global Community Conference Proceedings* (1998).
- [33] WENDLANDT, D., ANDERSEN, D. G., AND PERRIG, A. Perspectives: Improving SSH-style Host Authentication with Multi-Path Probing. In *USENIX Annual Technical Conference* (2008).
- [34] WHALEN, T., AND INKPEN, K. M. Gathering evidence: Use of visual security cues in web browsers. In *Proceedings of the Graphics Interface Conference* (2005).
- [35] WU, M., MILLER, R. C., AND GARFINKEL, S. L. Do Security Toolbars Actually Prevent Phishing Attacks? In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (2006).

A Sample Sizes

	Malware	Phishing	SSL	Add Exception
Release	1,968,707	89,948	NC	1,805,928
Beta	74,782	3,058	10,976	66,694
Dev	61,588	2,759	15,560	53,001
Nightly	58,789	4,239	18,617	64,725

Table 7: Warning impression sample sizes for Mozilla Firefox warnings, by channel, for all operating systems.

	Malware	Phishing	SSL	Add Exception
Mac	71,371	3,951	534	154,129
Win	1,892,285	85,598	10384	1,634,193
Linux	1,750	112	58	17,606

Table 8: Warning impression sample sizes for Mozilla Firefox warnings, by operating system. The malware, phishing, and the “Add Exception” samples are from the release channel, whereas the SSL samples are from the beta channel. The frame issue does not affect statistics that pertain only to the “Add Exception” dialog.

	Malware	Phishing	SSL
Stable	5,946,057	381,027	16,363,048
Beta	44,742	3,525	232,676
Dev	14,022	1,186	66,922
Canary	35,261	612	42,020

Table 9: Warning impression sample sizes for Google Chrome warnings, by channel, for all operating systems.

	Malware	Phishing	SSL
Mac	598,680	20,623	947,971
Windows	9,775,104	333,522	13,399,820
Linux	15,456	577	515,319
Android	NC	NC	1,499,938

Table 10: Warning impression sample sizes for Google Chrome warnings, by operating system, for the stable channel.

In Google Chrome, we recorded 6,040,082 malware warning impressions, 386,350 phishing warning impressions, and 16,704,666 SSL warning impressions. In Mozilla Firefox, we recorded 2,163,866 malware warning impressions, 100,004 phishing warning impressions, and 45,153 SSL warning impressions. Tables 7, 8, 9, and 10 further separate the sample sizes based on OS and release channel.

An Empirical Study of Vulnerability Rewards Programs

Matthew Finifter, Devdatta Akhawe, and David Wagner

University of California, Berkeley

{finifter, devdatta, daw}@cs.berkeley.edu

Abstract

We perform an empirical study to better understand two well-known vulnerability rewards programs, or VRPs, which software vendors use to encourage community participation in finding and responsibly disclosing software vulnerabilities. The Chrome VRP has cost approximately \$580,000 over 3 years and has resulted in 501 bounties paid for the identification of security vulnerabilities. The Firefox VRP has cost approximately \$570,000 over the last 3 years and has yielded 190 bounties. 28% of Chrome’s patched vulnerabilities appearing in security advisories over this period, and 24% of Firefox’s, are the result of VRP contributions. Both programs appear economically efficient, comparing favorably to the cost of hiring full-time security researchers. The Chrome VRP features low expected payouts accompanied by high potential payouts, while the Firefox VRP features fixed payouts. Finding vulnerabilities for VRPs typically does not yield a salary comparable to a full-time job; the common case for recipients of rewards in either program is that they have received only one reward. Firefox has far more critical-severity vulnerabilities than Chrome, which we believe is attributable to an architectural difference between the two browsers.

1 Introduction

Some software vendors pay security researchers for the responsible disclosure of a security vulnerability. Programs implementing the rules for this exchange are known as vulnerability rewards programs (VRPs) or bug bounty programs. The last couple of years have seen an upsurge of interest in VRPs, with some vendors expanding their existing programs [1, 19], others introducing new programs [3, 34, 38], and some companies offering to act as an intermediary between security researchers and vendors offering VRPs [53].

VRPs offer a number of potential attractions to software vendors. Offering adequate incentives entices security researchers to look for vulnerabilities, and this increased attention improves the likelihood of finding latent vulnerabilities.¹ Second, coordinating with security researchers allows vendors to more effectively manage vulnerability disclosures, reducing the likelihood of unexpected and

¹For instance, Linus’s Law suggests “Given enough eyeballs, all bugs are shallow.” [48]

costly zero-day disclosures. Monetary rewards provide an incentive for security researchers not to sell their research results to malicious actors in the underground economy or the gray world of vulnerability markets. Third, VRPs may make it more difficult for black hats to find vulnerabilities to exploit. Patching vulnerabilities found through a VRP increases the difficulty and therefore cost for malicious actors to find zero-days because the pool of latent vulnerabilities has been diminished. Additionally, experience gained from VRPs (and exploit bounties [23, 28]) can yield improvements to mitigation techniques and help identify other related vulnerabilities and sources of bugs. Finally, VRPs often engender goodwill amongst the community of security researchers. Taken together, VRPs provide an attractive tool for increasing product security and protecting customers.

Despite their potential benefits, there is an active debate over the value and effectiveness of VRPs. A number of vendors, notably Microsoft,² Adobe, and Oracle, do not maintain a VRP, with Microsoft arguing that VRPs do not represent the best return on investment on a per-bug basis [26]. Further, it is also not clear if the bounties awarded are a sufficient attraction for security researchers motivated by money—underground economy prices for vulnerabilities are far higher than those offered by VRPs [20, 37].

Given the emergence of VRPs as a component of the secure development lifecycle and the debate over the efficacy of such programs, we use available data to better understand existing VRPs. We focus on the Google Chrome and Mozilla Firefox web browsers, both of which are widely considered to have mature VRPs, as case studies. We analyze these VRPs along several dimensions with the intention of better understanding the characteristics, metrics, and trajectory of a VRP.

We make the following contributions:

- We collect and analyze data on vulnerability rewards over the last 3 years for the Google Chrome VRP and the Mozilla Firefox VRP (Section 3).
- We assess the state of these VRPs along several dimensions, including costs, benefits, popularity, and

²On June 19, 2013, during final preparation of this manuscript, Microsoft announced a month-long VRP for the IE11 developer preview [54].

efficacy (Section 4), finding that these VRPs appear both effective and cost-effective.

- We make concrete recommendations for software vendors aiming to start or evolve their own VRP (Section 5.2).
- We generate hypotheses, which identify opportunities for future research on VRPs and secure software development.

2 Background

A *secure software development lifecycle (SDLC)* aims to address software security throughout the entire software development process, from before specifications are developed to long after software has been released [15]. A *vulnerability remediation strategy* is any systematic approach whose goal is to reduce the number of software vulnerabilities [57]. Vulnerability remediation strategies are one important part of an SDLC, complemented by things like incident response [32], operational security considerations [46], and defense in depth [16].

Potential vulnerability remediation strategies include:

- **Code reviews.** These can range from informal, as-needed requests for code review to systematized, formal processes for mandatory code inspection. Typically, SDLCs also include an extra security review for security critical features.
- **Penetration testing.** Software vendors may perform in-house penetration testing or may hire external companies who specialize in this service. Penetration testing ranges from manual to automated.
- **Use of dynamic and static analysis tools.** Specialized tools exist for catching a wide range of flaws, e.g., memory safety vulnerabilities, configuration errors, and concurrency bugs.
- **Vulnerability rewards programs.** The focus of our study, VRPs have recently received increased attention from the security community.

How such strategies are systematized and realized varies widely between software vendors. One company might require mandatory code reviews before code check-in, while another might hire outside penetration testing experts a month before product release. Vendors often combine or innovate on existing strategies.

Vulnerability rewards programs (VRPs) appear to be emerging as a viable vulnerability remediation strategy. Many companies have them, and their popularity continues to grow [6, 9]. But VRPs have gone largely unstudied. For a company considering the use of a VRP in their SDLC, guidance is limited.

By studying mature, high-profile VRPs, we aim to provide guidance on the development of new VRPs and the evolution and maturation of existing VRPs. Vendors looking to grow their VRPs can benefit from an improved understanding of those VRPs we study.

Toward this end, we measure, characterize, and discuss the Google Chrome and Mozilla Firefox VRPs. We choose these VRPs in particular because browsers are a popular target for malicious actors today. Their ubiquitous nature and their massive, complex codebase with significant legacy components make them especially vulnerable. Complex, high-performance components with a large attack surface such as JavaScript JITs also provide an alluring target for malicious actors. For the same reasons, they are also widely studied by security researchers; they therefore provide a large sample size for our study. In addition, browser vendors were among the first to offer rewards for vulnerabilities: Mozilla’s VRP started in 2004 and Google introduced the Chrome VRP in 2010, before the security community at large adopted VRPs as a vulnerability remediation strategy.

2.1 Goals

We intend to improve our understanding of the following characteristics of a mature VRP: (1) Expected cost, (2) expected benefits, (3) incentive levels effective for encouraging and sustaining community participation, and (4) volume of VRP activity (e.g., number of patches coming out of VRP reports).

We do so by studying available data coming out of two exemplars of well-known, mature VRPs, that of Google Chrome and Mozilla Firefox. Understanding these VRPs will allow these vendors to evaluate and improve their programs, and it will suggest targets for other vendors to strive toward with their VRPs. At minimum, we hope to arrive at a better understanding of the current state of VRPs and how they have evolved. At best, we aim to make concrete suggestions for the development and improvement of VRPs.

2.2 Google Chrome VRP

The Google Chrome VRP³ is widely considered an exemplar of a mature, successful VRP. When first introduced in January 2010, the Google Chrome VRP offered researchers rewards ranging from \$500 for high- and critical-severity bugs, with a special \$1337 reward for particularly critical or clever bugs. Over time, the typical payout increased to a \$1000 minimum with a maximum payout of \$3133.7 for high-impact vulnerabilities. Additionally, the Chrome team, has provided rewards of up to \$31,336 for exceptional vulnerability reports [21].

³The program is officially the Chromium VRP with prizes sponsored by Google. We refer to it as the Google Chrome VRP for ease of exposition.

Google also sponsors a separate, semi-regular exploit bounty called the “pwnium” competition [23]. This program focuses on full exploits; a mere vulnerability is not enough. In return, it awards higher bounties (as high as \$150,000) for these exploits [8]. Reliable exploits for modern browsers typically involve multiple vulnerabilities and significant engineering effort. For example, the two winning entries in a recent “pwnium” contest required six and ten vulnerabilities in addition to “impressive” engineering in order to achieve a successful exploit [7, 45]. Our focus is on programs that provide bounties for vulnerabilities; we do not consider exploit bounties in this work.

The severity of a vulnerability plays a key role in deciding reward amounts. Google Chrome uses a clear guideline for deciding severity [12]. In short, a critical vulnerability allows an attacker to run arbitrary native code on the user’s machine; for instance, web-accessible memory corruption vulnerabilities that appear in the Chrome kernel⁴ are typically critical severity. A high-severity vulnerability is one that allows an attacker to bypass the same-origin policy, e.g., via a Universal XSS vulnerability (which enables an attacker to mount an XSS attack on any web site) or a memory corruption error in the sandbox. A vulnerability is of medium severity if achieving a high/critical status requires user interaction, or if the vulnerability only allows limited disclosure of information. Finally, a low-severity vulnerability refers to all the remaining security vulnerabilities that do not give the attacker control over critical browser features. Medium-severity vulnerabilities typically receive rewards of \$500, and low-severity vulnerabilities typically do not receive rewards.

2.3 Mozilla Firefox VRP

Mozilla’s VRP is, to the best of our knowledge, one of the oldest VRPs in the industry. It was first started in 2004 and based on a similar project at Netscape in 1995 [41]. The Mozilla VRP initially awarded researchers \$500 for high- or critical-severity security bugs. Starting July 1, 2010 Mozilla expanded its program to award all high/critical vulnerabilities \$3000 [1].

Mozilla’s security ratings are similar to that of Chrome. Critical vulnerabilities allow arbitrary code execution on the user’s computer. Vulnerabilities that allow an attacker to bypass the same-origin policy or access confidential information on the user’s computer are high severity. Due to the absence of privilege separation in the Firefox browser, *all* memory corruption vulnerabilities are critical, regardless of the component affected. Mozilla is currently investigating a privilege-separated design for Firefox [17, 36, 39].

Mozilla’s VRP also qualitatively differs from the

⁴Chrome follows a privilege-separated design [4]. The Chrome kernel refers to the privileged component.

Google program. First, Mozilla awards bounties even if the researcher publicly discusses the vulnerability instead of reporting it to Mozilla.⁵ Second, Mozilla also explicitly awards vulnerabilities discovered in “nightly” (or “trunk”) versions of Firefox. In contrast, Google discourages researchers from using “canary” builds and only awards bounties in canary builds if internal testing would miss those bugs [55].

3 Methodology

For both browsers, we collect all bugs for which rewards were issued through the browser vendor’s VRP. To evaluate the impact of the VRP as a component of the SDLC, we also collected all security bugs affecting stable releases. We chose to look only at bugs affecting stable releases to ignore the impact of transient bugs and regressions caught by internal testing.

For each bug in the above two datasets, we gathered the following details: (1) severity of the bug, (2) reward amount, (3) reporter name, (4) report date. For bugs affecting stable releases, we also aimed to gather the date a release patching the bug became available for download. As we discuss below, we were able to gather this data for only a subset of all bugs.

For all bugs, we mark a bug as internally or externally reported via a simple heuristic: if a reward was issued, the reporter was external, and otherwise the reporter was internal. Because low and medium severity vulnerabilities usually do not receive bounties, we only look at critical/high vulnerabilities when comparing internal and external bug reports. While all high/critical vulnerabilities are eligible for an award, a researcher can still refuse an award, in which case, our heuristic falsely marks the bug “internal.” We are aware of a handful of such instances, but there are not enough of these in our dataset to affect our analysis.

We are also aware of some researchers who transitioned from external to internal over the course of our study period. Because our heuristic operates on a per-bug basis (as opposed to marking each person as internal or external), the same person may be (intentionally) considered internal for one bug and external for another.

In this section, we present how we gathered this dataset for Chrome and Firefox. We first discuss how we identify the list of bugs affecting stable releases and bugs awarded bounties, followed by a discussion on how we identified, for each bug, other details such as severity. Finally, we discuss threats to the validity of our measurement study.

3.1 Gathering the Google Chrome dataset

We gathered data from the official Chromium bug tracker [13] after confirming with Google employees that

⁵But Mozilla reports that this was a rare occurrence over the period of time we consider, possibly because the VRP became widely known [56].

the bug tracker contained up-to-date, authoritative data on rewards issued through their VRP. We search the bug tracker for bugs marked with the special “Reward” label to collect bugs identified through the VRP. Next, we searched the bug tracker for bugs marked with the special “Security-Impact: Stable” to collect bugs affecting stable releases. Next, we remove the special Pwnium [23] rewards from all datasets because Pwnium rewards *exploits* instead of vulnerabilities as in the regular VRP. This provides us with 501 bugs identified through the VRP and 1347 bugs affecting stable releases.

The Chromium Bug tracker provides a convenient interface to export detailed bug metadata, including severity, reporter, and report date, into a CSV file, which we use to appropriately populate our dataset. We identify the reward amounts using the “Reward” label.

Unfortunately, the Chromium bug tracker does not include the release date of bug fixes. Instead, we gather this data from the Google Chromium release blog [27]. For each stable release of the Chromium browser, Google releases a blog post listing security bugs fixed in a release. For the subset of bugs mentioned in these release notes, we extract the release date of the stable version of Chrome that patches the bug.

3.2 Gathering the Mozilla Firefox dataset

Similar to Google Chrome, we searched Bugzilla, the Firefox bug tracker, for an attachment used to tag a bug bounty.⁶ We identified 190 bugs via this search.

Unlike the Chrome bug tracker, Bugzilla does not provide a convenient label to identify bugs affecting stable releases. Instead, Mozilla releases Mozilla Foundation Security Advisories (MFSA) with every stable release of Mozilla Firefox [40]. We scraped these advisories for a list of bugs affecting stable releases. We also use the MFSA to identify the release date of a patched, stable version of Firefox. We gathered 613 unique bugs from the MFSA advisories dating back to March 22, 2010 (Firefox 3.6).

Similar to the Chromium Bug tracker, the Bugzilla website provides a convenient interface to export detailed bug data into a CSV file for further analysis. We used Bugzilla to collect, for each bug above, the bug reporter, the severity rating, and the date reported. The security severity rating for a bug is part of the Bugzilla keywords field and not Bugzilla’s severity field. We do not separately collect the amount paid because, as previously discussed, Mozilla’s expanded bounty program awards \$3,000 for all critical/high vulnerabilities.

⁶The existence of this attachment is not always visible to the public. We acknowledge the support of Mozilla contributor Dan Veditz for his help gathering this data.

Severity	Chrome		Firefox	
	Stable	Bounty	Stable	Bounty
Low	226	1	16	1
Medium	288	72	66	9
High	793	395	79	38
Critical	32	20	393	142
Unknown	8	13	59	0
Total	1347	501	613	190

Table 1: Number of observations in our dataset.

3.3 Dataset

Table 1 presents information about the final dataset we used for our analysis. We have made our dataset available online for independent analysis [33].

3.4 Threats to validity

In this section, we document potential threats to validity so readers can better understand and take into account the sources of error and bias in our study.

It is possible that our datasets are incomplete, i.e., there exist patched vulnerabilities that do not appear in our data. For example, for both Chrome and Firefox, we rely heavily on the keyword/label metadata to identify bugs; since this labeling is a manual process, it is possible that we are missing bugs. To gather the list of bugs affecting stable releases, we use the bug tracker for Chrome but use security advisories for Mozilla, which could be incomplete. Given the large number of vulnerabilities we do have in our datasets, we expect that a small number of missing observations would not materially influence the conclusions we draw.

We treat all rewards in the Firefox VRP as \$3,000 despite knowing that 8% of the rewards were for less than this amount [56]. Data on which rewards were for less money and what those amounts were is not publicly available. Any results we present regarding amounts paid out for Firefox vulnerabilities may therefore have as much as 8% error, though we expect a far lower error, if any. We do not believe this affects the conclusions of our analysis.

Parts of our analysis also compare Firefox and Chrome VRPs in terms of number of bugs found, which assumes that finding security vulnerabilities in these browsers requires comparable skills and resources. It could be the case that it is just easier to find bugs in one over the other, or one browser has a lower barrier to entry for vulnerability researchers. For example, the popular Address Sanitizer tool worked only on Google Chrome until Mozilla developers tweaked their build infrastructure to enable support for the same [31]. Another confound is the possibility that researchers target a browser based on personal factors beyond VRPs. For example, researchers could look for vulnerabilities only in the browser they personally use.

Assigning bug severity is a manual process. While

the severity assignment guidelines for both browsers are similar, it is possible that vendors diverge in their actual severity assignment practices. As a result, the bug severities could be incomparable across the two browsers.

We study only two VRPs; our results do not necessarily generalize to any other VRPs. We caution the reader to avoid generalizing to other VRPs, but instead take our results as case studies of two mature, well-known VRPs.

4 Results

We study VRPs from the perspectives of three interested parties: the software vendor, the independent security researcher, and the security researcher employed full-time by the software vendor.

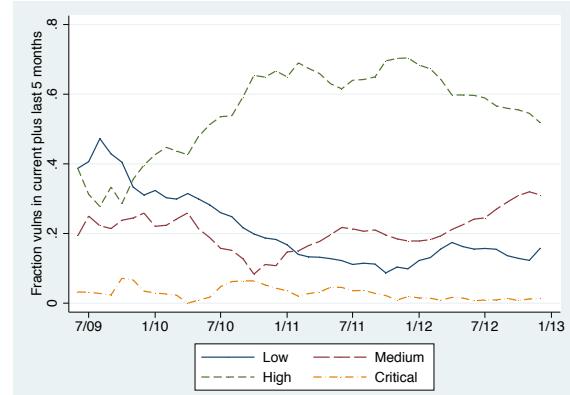
4.1 Software vendor

We model the software vendor’s goal as follows: to increase product security as much as possible while spending as little money as possible. There are many potential strategies for working toward this goal, but in this paper we consider the strategy of launching a VRP. We present data on costs and benefits for two VRPs, and generate hypotheses from this data. The software vendor’s motivation can also include publicity and engaging the security research community. We do not measure the impact of VRPs on these.

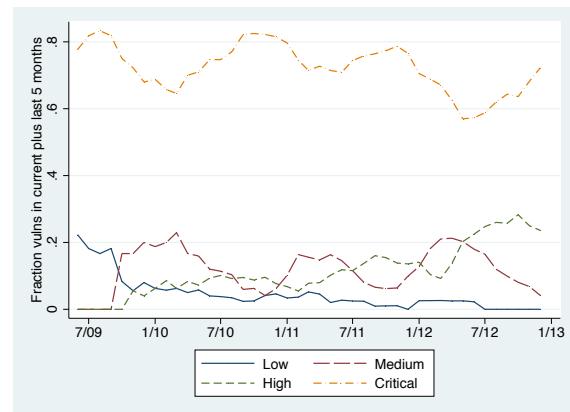
4.1.1 Number of vulnerabilities

The intended benefit of a VRP is to improve product security. A reduction in the number of latent vulnerabilities is one way of improving product security. We find that the Chrome VRP uncovers about 2.6 times as many vulnerabilities as that of Firefox (501 vs. 190), despite the fact that Chrome’s total number of security vulnerabilities in our dataset is only 2.2 times that of Firefox (Table 1). 27.5% of bugs affecting Chrome releases originate from VRP contributions (371 of 1347), and 24.1% of bugs affecting Firefox releases (148 of 613) result from VRP contributions.

Discussion Both VRPs yield a significant fraction of the total number of security advisories, which is a clear benefit. Chrome is seeing approximately 1.14 times the benefit of Firefox by our metric of fraction of advisories resulting from VRP contributions. We only study bugs affecting stable releases in this metric and caution the reader from assuming that VRPs are competitive with internal researchers. For both browsers, internal researchers find far more bugs during the months of testing that precede a typical browser release. For example, from January to May 2013, across all release channels, Google researchers found 140 high or critical vulnerabilities in Chrome, while the Chrome VRP only found 40 vulnerabilities in the same time period.



(a) Chrome



(b) Firefox

Figure 1: Moving average over the current plus 5 previous months of the percentage of vulnerabilities at each severity level (low is blue solid line, medium is red long-dashed line, high is green short-dashed line, and critical is yellow dash-dotted line). In this and subsequent line graphs, the data are aggregated by month to improve graph readability, and the x-axis represents the open date of the bug.

4.1.2 Vulnerability severity

Another measure of improvement to product security is change in vulnerability severity over time. It is a good sign, for example, if the percentage of critical-severity vulnerabilities has decreased over time.

Table 1 lists the total number of vulnerabilities by severity for Firefox and Chrome. Figure 1 plots the fraction of vulnerabilities at each severity level over the current plus 5 previous months.

Discussion Firefox has a much higher ratio of critical vulnerabilities to high vulnerabilities than Chrome. We expect that many of Firefox’s critical vulnerabilities would instead be high severity if, like Chrome, it also had a privilege-separated architecture. The lack of such an architecture means that any memory corruption vulnera-

bility in Firefox is a critical vulnerability. We therefore hypothesize that:

Hypothesis 1 *This architectural difference between Chrome and Firefox—that the former is privilege-separated and the latter is not—is the most influential factor in causing such a large difference in vulnerability severity classification.*

The fraction of critical severity bugs has remained relatively constant for Chrome. We also notice the start of a trend in Chrome—the fraction of high severity vulnerabilities is declining and the fraction of medium severity vulnerabilities is increasing.

Chrome’s privilege-separated architecture means that a critical vulnerability indicates malicious code execution in the privileged process. We see that there continue to be new bugs resulting in code execution in the privileged process. Further investigation into these bugs can help understand how and why they continue to surface.

Low-severity vulnerabilities in Google Chrome make up a significant fraction of all vulnerabilities reported. In contrast, the fraction of low- and medium-severity vulnerabilities in Firefox remains negligible.

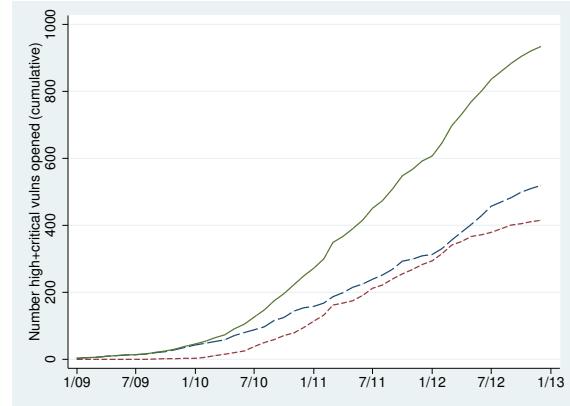
Note that our dataset does not allow us to attribute any change in vulnerability severity over time to the use or success of a VRP. However, severity over time is a metric worth tracking for a software vendor because it can indicate trends in the overall efforts to improve product security, of which a VRP may be one component.

4.1.3 Community engagement

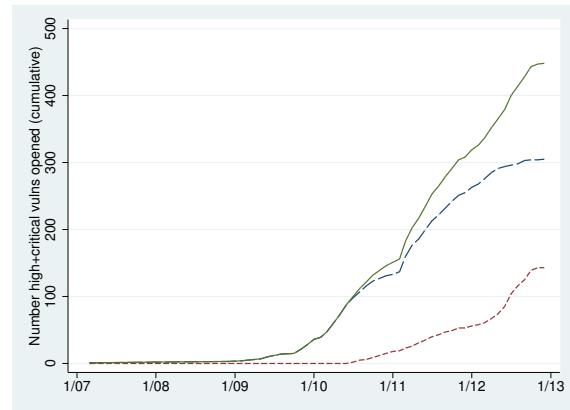
One advantage of VRPs is engagement with the broader security community. We studied this engagement along two axes: (1) the contribution of internal and external researchers towards identifying security vulnerabilities, and (2) the number of external participants in each VRP.

Figure 2 depicts the cumulative number of high- and critical-severity vulnerabilities patched and Figure 3 depicts the same, but for only critical vulnerabilities. Table 2 shows the distribution of the total number of vulnerabilities reported by each external participant in each of the two VRPs. Although a few external participants submit many bugs, there is a clear long tail of participants in both VRPs. Table 3 shows the same distribution, but for internal (i.e., employee) reporters of vulnerabilities.

Discussion For both browsers, internal contributions for high- and critical-severity vulnerabilities have consistently yielded the majority of patches. The number of external contributions to Chrome has nearly caught up with the number of internal contributions (i.e., around 4/11 and 3/12, in Figure 2a) at various times, and as of the end of our study, these two quantities are comparable. Considering only critical-severity vulnerabilities, external contributions have exceeded internal contributions as of



(a) Chrome



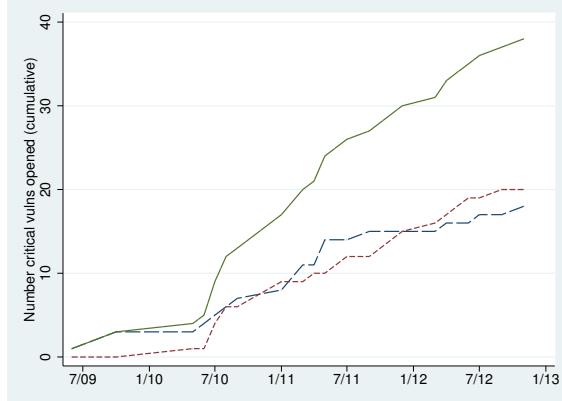
(b) Firefox

Figure 2: Number of high- plus critical-severity vulnerabilities reported over time, discovered internally (blue long-dashed line), externally (red short-dashed line), and total (green solid line).

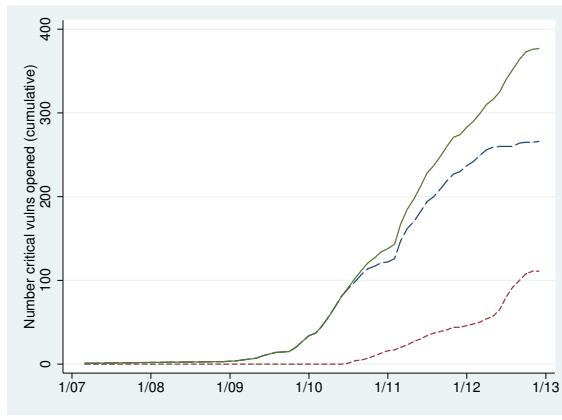
the end of our study. For Firefox, on the other hand, the number of external contributions has consistently been far lower than the number of internal contributions.

We observe an increase in the rate of external contributions to Chrome starting around July 2010, six months after the inception of the VRP. As seen in Figure 3a, this is more pronounced when considering only critical-severity vulnerabilities. We conjecture that this change corresponds to increased publicity for the Chrome VRP after Google increased reward amounts [19].

Linus’s Law, defined by Eric Raymond as “Given enough eyes, all bugs are shallow,” suggests that it is in the interests of the software vendor to encourage more people to participate in the search for bugs. The distributions of bugs found by external participants indicate that both VRPs have been successful in encouraging broad community participation. The existence of a long tail of contributors holds for internal contributors as well as external contributors.



(a) Chrome



(b) Firefox

Figure 3: Number of critical-severity vulnerabilities reported over time, discovered internally (blue long-dashed line), externally (red short-dashed line), and total (green solid line).

4.1.4 Diversity

There is the potential benefit that the wide variety of external participants may find *different* types of vulnerabilities than internal members of the security team. A few pieces of anecdotal evidence support this. Chrome has awarded bounty amounts that include \$1,337, \$2,337, \$3,133.7, and \$7,331 for bugs that they considered clever or novel [21], and our dataset contains 31 such awards. Additionally, one of PinkiePie’s Pwnium exploits led to a full review of the Chrome kernel file API, which resulted in the discovery of several additional vulnerabilities [21, 51]. The Chrome security team missed all these issues until PinkiePie discovered and exploited one such issue [14]. We therefore hypothesize that:

Hypothesis 2 *An increase in the number of researchers looking for vulnerabilities yields an increase in the diversity of vulnerabilities discovered.*

# Bugs	Freq.
1	45
3	2
4	1
6	1
10	1
12	2
13	3
16	1
17	1
22	1
24	1
27	1
35	1
48	1
92	1
Total	63

(a) Chrome

# Bugs	Freq.
1	46
2	9
3	4
5	1
6	1
9	1
10	1
12	1
14	1
47	1
Total	66

(b) Firefox

Table 2: Frequency distribution of number of high- or critical-severity vulnerabilities found by external contributors.

# Bugs	Freq.
1	67
2	10
3	10
4	2
5	2
14	1
20	2
67	1
263	1
Total	96

(a) Chrome

# Bugs	Freq.
1	43
2	10
3	7
4	3
5	2
6	2
7	1
12	1
13	1
15	1
17	2
18	1
21	1
23	1
44	1
Total	77

(b) Firefox

Table 3: Frequency distribution of number of high- or critical-severity bugs found by internal contributors.

4.1.5 Cost of rewards

Though the number of bounties suggests that VRPs provide a number of benefits, a thorough analysis necessarily includes an analysis of the costs of these programs. In this section, we examine whether VRPs provide a *cost-effective* mechanism for software vendors. We analyze one ongoing cost of the VRP: the amount of money paid to researchers as rewards for responsible disclosure. Running a VRP has additional overhead costs that our dataset does not provide any insight into.

Figure 4 displays the total cost of paying out rewards for vulnerabilities affecting stable releases. We find that over the course of three years, the costs for Chrome and Firefox are similar: approximately \$400,000.

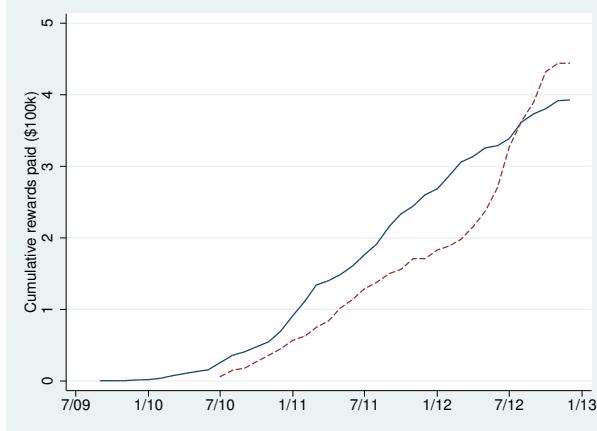


Figure 4: Cumulative rewards paid out for Chrome (blue solid line) and Firefox (red dashed line), excluding rewards for vulnerabilities not affecting stable versions.

Rewards for Development Releases Both Firefox and Chrome issue rewards for vulnerabilities that do not affect stable release versions, increasing the *total* cost of the VRP beyond the cost of rewarding vulnerabilities affecting stable releases. One potential drawback of such rewards is that the VRPs awards transient bugs that may never make their way into a user-facing build in the first place. On the other hand, such rewards could catch bugs earlier in the development cycle, reducing the likelihood of expensive out-of-cycle releases.

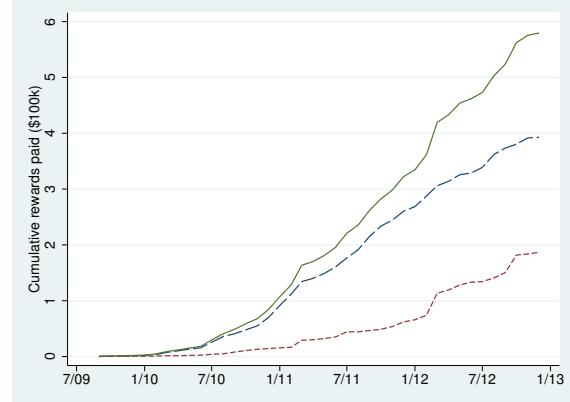
Figure 5 shows the cumulative rewards issued by each of the two VRPs for vulnerabilities affecting stable releases, vulnerabilities not affecting stable releases, and the sum of the two. We observe that the Chrome VRP has paid out \$186,839, 32% of its total cost of \$579,605 over the study period for vulnerabilities not affecting a stable release. The Firefox VRP has paid out \$126,000, 22% of its total cost of \$570,000, over the study period for such vulnerabilities.

Discussion The total cost of each of the two VRPs is remarkably similar. Both spend a significant fraction of the total cost on vulnerabilities not present in stable release versions.

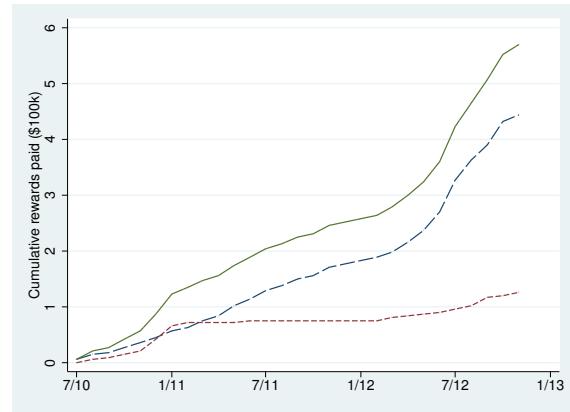
4.1.6 Average daily cost

Figure 6 plots the average daily cost to date of each VRP over time. We see that Chrome’s VRP has cost \$485 per day on average, and that of Firefox has cost \$658 per day.

Discussion If we consider that an average North American developer on a browser security team (i.e., that of Chrome or Firefox) would cost the vendor around \$500 per day (assuming a \$100,000 salary with a 50% overhead), we see that the cost of either of these VRPs is comparable to the cost of just one member of the browser security team. On the other hand, the *benefit* of a VRP



(a) Chrome



(b) Firefox

Figure 5: Cumulative rewards paid out for vulnerabilities affecting a stable release (blue long-dashed line), vulnerabilities not affecting any stable release (red short-dashed line), and the sum of the two (green solid line).

far outweighs that of a single security researcher because each of these VRPs finds many more vulnerabilities than any one researcher is likely to be able to find. For bugs affecting stable releases, the Chrome VRP has paid 371 bounties, and the most prolific internal security researcher has found 263 vulnerabilities. For Firefox, these numbers are 148 and 48, respectively. Based on this simple cost/benefit analysis, we hypothesize that:

Hypothesis 3 A VRP can be a cost-effective mechanism for finding security vulnerabilities.

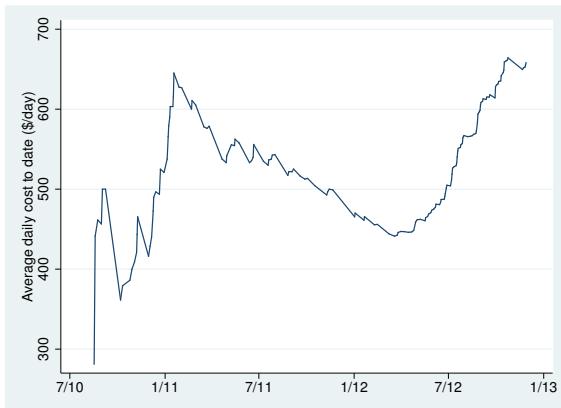
4.2 External security researcher

We model the goal of an external security researcher as follows: to make as much money as possible in as little time as possible.⁷ The researcher can contribute to

⁷Naturally, this does not reflect the reality of every security researcher’s goal.



(a) Chrome



(b) Firefox

Figure 6: Average daily cost to date since first reward.

any VRP he chooses, each of which pays out according to some rewards distribution. The researcher has some perception of security of each product, which reflects the expected amount of time the researcher will have to spend to find a vulnerability.

A rational strategy for the security researcher is to look for products perceived to be insecure that also happen to pay large bounties. This implies that a product with a higher perceived security must pay relatively higher bounties to encourage researchers to look for vulnerabilities in it as opposed to in a different product that is perceived to be less secure. We therefore hypothesize that:

Hypothesis 4 *In an efficient market with many VRPs and fluid reward structures, larger rewards reflect a higher level of perceived security by the population of researchers who contribute to VRPs.*

4.2.1 Reward amounts

Our dataset provides insight into the distributions of rewards for two products. Firefox offers a standard reward of \$3,000 for all vulnerabilities. In contrast, the Chrome

Amount (\$)	Frequency (%)
500	24.75
1,000	60.08
1,337	3.59
1,500	2.99
2,000	2.99
2,337	0.60
2,500	0.60
3,000	0.20
3,133	1.80
3,500	0.20
4,000	0.20
4,500	0.20
5,000	0.20
7,331	0.20
10,000	1.40

Table 4: Percentage of rewards given for each dollar amount in Chrome VRP.

VRP’s incentive structure provides different reward levels based on a number of subjective factors like difficulty of exploit, presence of a test case, novelty, and impact, all of which is at the discretion of Google developers.

Table 4 depicts the reward amounts paid to external researchers by the Chrome VRP. The majority of the rewards are for only \$500 or \$1,000. Large rewards, such as \$10,000 rewards, are infrequent.

Discussion We hypothesize that high maximum rewards entice researchers to participate, but low (\$500 or \$1,000) rewards are typical, and the total cost remains low. The median (mean) payout for Chrome bug bounty is \$1,000 (\$1,156.9), suggesting that a successful VRP can be inexpensive with a low expected individual payout. Much like the lottery, a large maximum payout (\$30,000 for Chrome), despite a small expected return (or even negative, as is the case of anyone who searches for bugs but never successfully finds any) appears to suffice in attracting enough participants. Bhattacharyya and Garrett [5] explain this phenomenon as follows:

Lotteries are instruments with negative expected returns. So when people buy lottery tickets, they are trading off negative expected returns for skewness. Thus, if a lottery game has a larger prize amount, then a buyer will be willing to accept a lower chance of winning that prize.

4.2.2 VRPs as employment

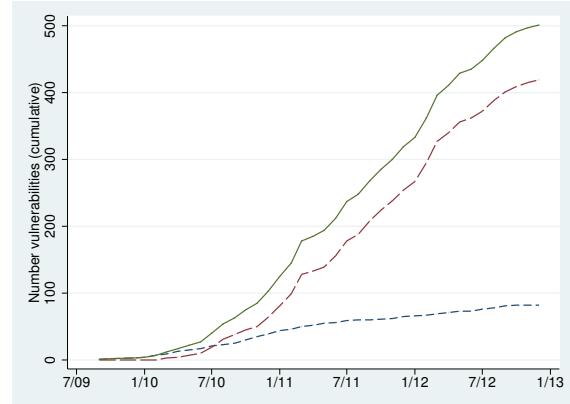
Our dataset also allows limited insight into how much money independent security researchers make. Table 5a displays the total amounts of money earned by each external contributor to the Chrome VRP. Only three external contributors (out of eighty two) have earned over \$80,000 over the lifetime of the VRP, and an additional five have earned over \$20,000.

\$ earned	Freq.
500	26
1,000	25
1,337	6
1,500	2
2,000	1
3,000	2
3,133	1
3,500	2
4,000	1
5,000	1
7,500	1
11,000	1
11,500	1
11,837	1
15,000	1
17,133	1
18,337	1
20,633	1
24,133	1
28,500	1
28,633	1
37,470	1
80,679	1
85,992	1
105,103	1
Total	82

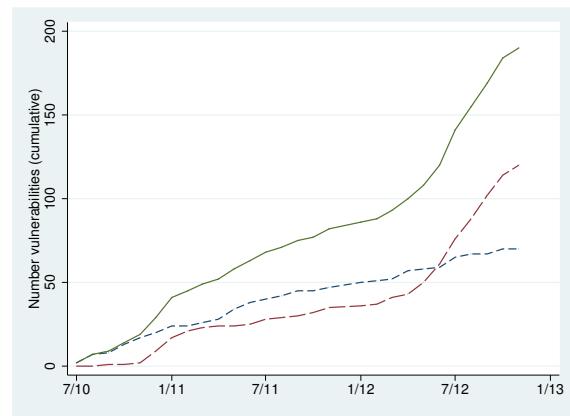
(a) Chrome

\$ earned	Freq.
3,000	46
6,000	12
9,000	4
12,000	1
15,000	1
21,000	1
27,000	1
30,000	1
36,000	1
42,000	1
141,000	1
Total	70

(b) Firefox



(a) Chrome



(b) Firefox

Table 5: Frequency distributions of total amounts earned by external VRP contributors.

In contrast to Google Chrome, we see in Table 5b that a single Firefox contributor has earned \$141,000 (\$47,000 per year) since the beginning of our study period. Ten of this individual’s rewards, representing \$30,000, were for vulnerabilities that did not impact a stable release. Six contributors have earned more than \$20,000 via the Mozilla VRP.

Discussion Based on the data from 2 VRPs, we hypothesize that:

Hypothesis 5 *Contributing to a single VRP is, in general, not a viable full-time job, though contributing to multiple VRPs may be, especially for unusually successful vulnerability researchers.*

4.2.3 Repeat contributors

Figure 7 shows the cumulative number of vulnerabilities patched due to reports from first-time VRP participants and repeat participants. For both programs, first-time participant rewards are steadily increasing, and repeat participant rewards are increasing even more quickly.

Discussion Both VRP incentive structures are evidently sufficient for both attracting new participants and continuing to entice existing participants, though we do note differences between Chrome and Firefox. Until recently, repeat participants in Firefox’s VRP represented a relatively small fraction of the number of awards issued.

Figure 7: Cumulative number of vulnerabilities rewarded, as reported by (1) first-time VRP contributors (blue short-dashed line), (2) repeat contributors (red long-dashed line), and (3) all contributors (green solid line).

Chrome, on the other hand, has seen the majority of its reports come from repeat participants for nearly the whole lifetime of its VRP.

4.3 Internal security researcher

An internal security researcher is a full-time employee of a software vendor who is paid a salary to find as many vulnerabilities as possible. Google hired at least three researchers who first came to light via the Chrome VRP [21] and Mozilla hired at least three researchers as well [56].

Discussion A software vendor may choose to hire an unusually successful independent security researcher. The researcher’s past performance indicates how many vulnerabilities the vendor can expect them to find, and the vendor may prefer to pay a fixed salary instead of a per-vulnerability reward. The researcher may also prefer this; the researcher trades a potentially higher amount of cash for less compensation, but more benefits and job security.

Severity	Median, Chrome	Std. dev. Chrome	Median, Firefox	Std. dev., Firefox
Low	58.5	110.6	114	256.1
Medium	45.5	78.9	106	157.6
High	28.0	35.3	62.5	85.7
Critical	20.0	26.6	76	116.5

Table 6: Median and standard deviation of number of days between vulnerability report and release that patches the vulnerability, for each severity level.

Accordingly, we hypothesize that:

Hypothesis 6 *Successful independent security researchers bubble to the top, where a full-time job awaits them.*

4.4 Other factors

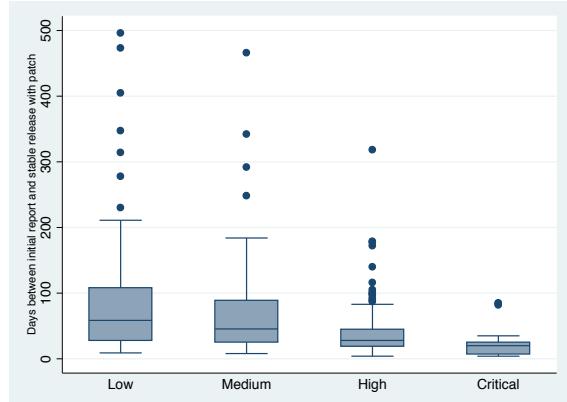
Our dataset provides an additional opportunity to better understand the state of the SDLC (software development lifecycle) at Mozilla and Google. In particular, we analyze (1) the elapsed time between a vulnerability report and the release of a patched browser version that fixes the vulnerability, and (2) how often vulnerabilities are independently discovered, and what the potential implications are of this rediscovery rate.

4.4.1 Time to patch

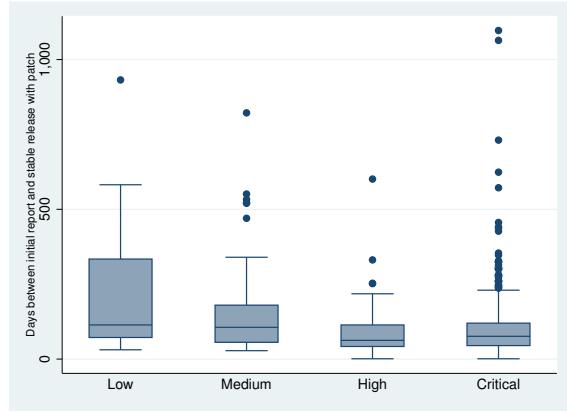
As previously discussed, we choose to study time to release a patched version, *not* time to commit a patch. Although relying on time to release a patch means we analyze only a subset of the data (Section 3), we believe the time to release a patched version of the browser is the more useful metric for end users. Mozilla Firefox and Google Chrome both follow a rapid-release cycle, with a new release every 6 or 7 weeks [11, 25]. In some cases, browser vendors release an out-of-band (or “chemspill”) release for vulnerabilities with active exploits in the wild. Such out-of-band releases are one of the most expensive incidents for software companies, with costs running into millions of dollars [30]. Our metric awards the engineering and management commitment required in choosing to release such versions.

Figure 8 depicts the time between initial report of a vulnerability and the stable release that patches it. Table 6 gives summary statistics for these distributions.

Figure 9 is a scatter plot of the same data, which allows us to see changes in time to patch over time. Figure 10 shows the change in standard deviation of time to patch over time. More specifically, for a given release date, the y-value is the standard deviation for all bugs patched in that release or up to five prior releases. These graphs indicate that the standard deviation in time to patch critical vulnerabilities has slowly dropped for Firefox, while Chrome’s time to patch critical vulnerabilities has remained relatively constant over time.



(a) Chrome

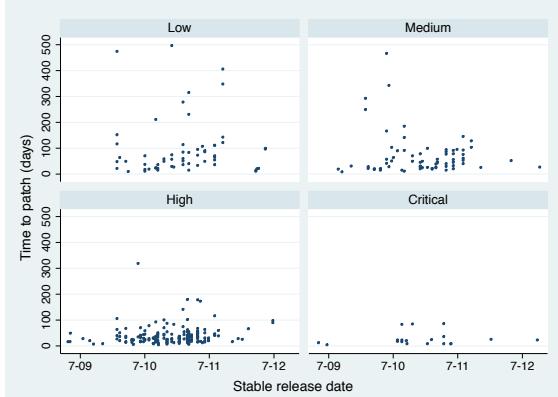


(b) Firefox

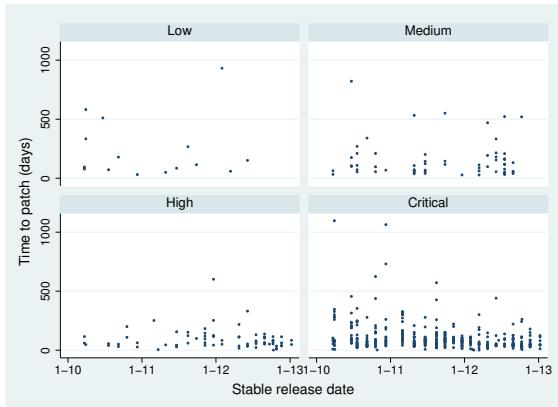
Figure 8: Box and whisker plots depicting the distributions of time between vulnerability report and release that patches the vulnerability, for each severity level.

Discussion For Chrome, both the median time to patch and the variance are lower for higher severity vulnerabilities. This is an important parameter for a VRP because responsible disclosure depends critically on vendor response time [22, 50]. If a vendor does not patch in a reasonable time frame, security researchers are less likely to exercise responsible disclosure. Accordingly, this may be a contributing factor in Firefox’s lower degree of community participation (as compared to Chrome), given that the time to patch critical vulnerabilities in Firefox is longer and has very high variance.

In Chrome, the time to patch is faster for critical vulnerabilities than it is for high severity vulnerabilities. This trend continues for medium- and low-severity vulnerabilities as well. This indicates correct prioritization of higher-severity vulnerabilities by Chrome security engineers. The same cannot be said for Firefox; high and critical severity vulnerabilities tend to take about the same amount of time to fix.



(a) Chrome



(b) Firefox

Figure 9: Scatter plots depicting the time between vulnerability report and release that patches the vulnerability vs. time, for each severity level.

The high variance in Firefox’s time to patch critical vulnerabilities may be partly attributable to the lack of privilege separation in Firefox, since a larger TCB for critical vulnerabilities means that there is a larger pool of engineers owning code that might hold a critical vulnerability. However, it is an encouraging sign that Firefox has gradually reduced this variance. Nonetheless, the variance in patch times and typical time to patch for Firefox both remain far higher than we see for Chrome, suggesting the need for a concerted effort at reducing this.

4.4.2 Independent discovery

Using the Chromium release blog, we manually coded an additional variable `independent`. This variable represents the number of times a vulnerability was independently discovered. We coded it using the text of the `credit` variable, which mentions “independent discovery” of a vulnerability in the case of multiple independent discoveries.



(a) Critical-severity vulnerabilities.



(b) Critical and high severity vulnerabilities.

Figure 10: Standard deviation of time to patch over time. For a given release date, the y-value is the standard deviation of all bugs patched in that release or up to five prior releases. The red solid line is for Firefox, and the blue dashed line is for Chrome.

Our Chrome dataset indicates when a vulnerability was independently discovered by multiple parties, identifies the parties, and in some cases, gives an upper bound on the time between discovery and rediscovery. Of the 668 vulnerabilities in our Chrome VRP dataset, fifteen (2.25%) of them had at least two independent discoveries, and two of these had three independent discoveries. This is a lower bound on the number of independent discoveries of these vulnerabilities, since it represents only those known to the vendor.

Figure 11 displays the independent rediscovery rates for individuals. Each dot represents an individual contributor in our dataset. Its x-value gives the number of vulnerabilities discovered by this individual, and its y-value gives the number of these vulnerabilities independently rediscovered by another contributor in our dataset. Of those individuals who reported five or more vulnerabilities, the highest rediscovery rate is 25% and the mean is

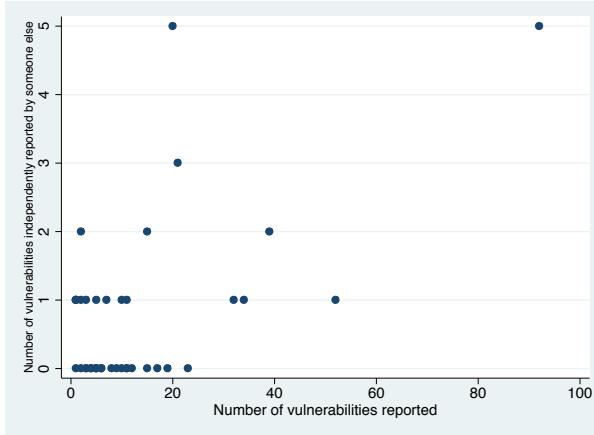


Figure 11: Independent vulnerability discoveries within the Chrome VRP dataset. Each dot represents an individual contributor in our dataset. Its x-value gives the number of vulnerabilities contributed by this individual, and its y-value gives the number of these contributions that were independently discovered by another contributor in our dataset.

4.6%.

Our Firefox dataset does not indicate independent rediscovery, but we have limited data from personal communication with a Firefox security engineer [56]. He indicated that there had been at least 4–7 vulnerabilities reported through the VRP for which there had been two independent discoveries, a rate of 2.7% to 4.7%, which is consistent with what we see in our Chrome dataset.

Discussion Independent rediscovery rates can have implications for estimating the number of latent bugs in software [29] as well as understanding the expected decay rate of a stash of zero-day vulnerabilities.

A zero-day loses its value when the vendor becomes aware of it, which happens via independent discovery of the vulnerability. Thus, a stash of zero-days will decay at some rate. From the limited data available to us via our study, we hypothesize that:

Hypothesis 7 *The decay rate of a stash of zero-day vulnerabilities is low enough to be inconsequential as a result of relatively low empirical independent rediscovery rates.*

We encourage future studies that aim to confirm or refute this hypothesis using larger, more appropriate datasets.

5 Discussion and recommendations

In this section, we synthesize what we have learned and present concrete recommendations for software vendors based on our data analysis.

5.1 Mozilla Firefox vs. Google Chrome

Despite costing approximately the same as the Mozilla program, the Chrome VRP has identified more than three

times as many bugs, is more popular and shows similar participation from repeat and first-time participants. There is a stark difference between the levels of external participation in the two VRPs (Figure 2).

Despite having the oldest bounty program, external contributions lag far behind internal contributions to Firefox’s security advisories. In contrast, external contributions to Chrome’s security advisories closely rival internal contributions. Investigating further, we find three key differences between the two programs:

Tiered structure with large special rewards Mozilla’s program has a fixed payout of \$3,000, which is approximately equal to the normal maximum payout for Chrome (\$3,1337). Nonetheless, Chrome’s tiered structure, with even higher payouts (e.g., \$10,000) possible for clever bugs and special cases appears to be far more effective in encouraging participation. This makes sense with an understanding of incentives in lotteries: the larger the potential prize amount, the more willing participants are to accept a lower expected return, which, for VRPs, means the program can expect more participants [5].

Time to patch We see a far higher variance in the time-to-release-patch metric for critical vulnerabilities in Mozilla Firefox. It is generally accepted that the viability of responsible disclosure depends on a reasonable vendor response time [50]. Thus, the high variance in Mozilla’s response time could affect responsible disclosure through the VRP.

Higher profile Chrome’s VRP has a higher profile, with annual competitions like Pwnium providing particularly high rewards (up to \$150,000). Chrome authors also provide extra reward top-ups for “interesting” bugs. We believe this sort of “gamification” leads to a higher profile for the Chrome VRP, which may help encourage participation, particularly from researchers interested in wider recognition.

Our methodology does not provide insight into the motivations of security researchers and the impact of VRP designs on the same—a topic we leave for future work. Nevertheless, we hypothesize that these three factors combined explain the disparity in participation between the Firefox and Chrome VRPs. Accordingly, we recommend Mozilla change their reward structure to a tiered system like that of Chrome. We urge Mozilla to do whatever it takes to continue to reduce the variance in time to release a patch for critical vulnerabilities, though we also realize the difficulty involved in doing so. Ongoing attempts at privilege separation might enable reducing the variance in time to patch critical vulnerabilities [17, 36, 39]. Mozilla can also consider holding its own annual competitions or otherwise increasing the PR surrounding its VRP.

5.2 Recommendations for vendors

Our study of the Chrome and Firefox VRPs yield a number of observations that we believe can guide vendors interested in launching or evolving their own VRPs.

We find that VRPs appear to provide an economically efficient mechanism for finding vulnerabilities, with a reasonable cost/benefit trade-off (Sections 4.1.1 and 4.1.6). In particular, they appear to be 2-100 times more cost-effective than hiring expert security researchers to find vulnerabilities. We therefore recommend that more vendors consider using them to their (and their users') advantage. The cost/benefit trade-off may vary for other types of (i.e., non-browser) software vendors; in particular, the less costly a security incident is for a vendor, the less useful we can expect a VRP to be. Additionally, we expect that the higher-profile the software project is (among developers and security researchers), the more effective a VRP will be.

Response time, especially for critical vulnerabilities, is important (Section 4.4.1). High variance in time-to-patch is not appreciated by the security community. It can reasonably be expected to reduce participation because it makes responsible disclosure through the VRP a less attractive option than the other options available to security researchers.

VRP incentive design is important and should be carefully considered. Chrome's tiered incentive structure appears more effective at encouraging community participation than Firefox's fixed-amount incentive structure (Section 4.2.1). Additionally, both Chrome and Firefox have increased their rewards over time. Doing so increases publicity, entices participants, and signals that a vendor is betting that their product has become more secure over time.

Our analysis demonstrates the impact of privilege separation on the Chrome VRP (Section 4.1.2). Privilege separation also provides flexibility to the Chrome team. For example, a simple way for Chrome to cut costs while still increasing participation could be to reduce reward amounts for high-severity vulnerabilities and increase reward amounts for critical-severity vulnerabilities. Mozilla does not have this flexibility. Vendors should consider using their security architecture to their advantage.

6 Related Work

Mein and Evans share our motivation and present Google's experience with its vulnerability rewards programs [35]. In contrast, our focus is on understanding and comparing two popular VRPs run by competing browser vendors. We also perform a number of analyses not performed by the previous work as well as make our data available for other researchers. We also independently confirm that, for both Google and Mozilla, VRPs are cost-effective mechanisms for finding security

vulnerabilities.

Development lifecycle datasets Many authors have looked to large datasets, including code repositories, bug trackers, and vulnerability databases, to gather and analyze data in an effort to better understand some aspect of the development lifecycle. Rescorla gathered data from NIST's ICAT database (which has since been updated and renamed to NVD [44]) to analyze whether vulnerability rates tend to decrease over time [49]. He found no evidence that it is in fact worthwhile for software vendors to attempt to find vulnerabilities in their own software because there is no evidence that such efforts are reducing vulnerability rates.

Ozment and Schechter used the OpenBSD CVS repository to ask and answer similar questions as Rescorla [47]. They find that the rate of discovery of what they call *foundational* vulnerabilities—those present since the beginning of the study period—had decreased over the study period.

Neuhaus and Plattner use vulnerability reports for Mozilla, Apache httpd, and Apache Tomcat to evaluate whether vulnerability fix rates have changed over time [42]. They conclude that the supply of vulnerabilities is not declining, and therefore that attackers and/or vulnerability researchers have not hit diminishing returns in looking for vulnerabilities.

Neuhaus et al. use a dataset of Firefox security advisories in combination with the Firefox codebase to map vulnerabilities to software components and predict which components are likely to contain vulnerabilities [43].

Scholte et al. use the NVD to evaluate how cross-site scripting and SQL injection vulnerabilities have evolved over time [52]. They find that the complexity of such vulnerabilities does not appear to have changed over time and that many foundational cross-site scripting vulnerabilities are still being discovered.

Evaluating vulnerability-finding techniques Other work has focused specifically on evaluating the many available techniques for finding vulnerabilities, though we are unaware of any previous work that has considered public-facing VRPs as one such technique.

Austin and Williams evaluated four different techniques for vulnerability discovery on two health record systems: “systematic and exploratory manual penetration testing, static analysis, and automated penetration testing” [2], finding that very few vulnerabilities are in fact found by multiple techniques and that automated penetration testing is the most effective in terms of vulnerabilities found per hour.

Finifter and Wagner compared manual source code analysis to automated penetration testing on a web application, with similar findings: the techniques are complementary, and manual analysis found more vulnerabilities,

but took much more time than automated penetration testing [24].

Edmundson et al. found that different reviewers tend to find different vulnerabilities and, even in a small codebase, it takes many reviewers to spot all or even a significant fraction of the vulnerabilities present [18]. This is consistent with our findings about the effectiveness of crowdsourced VRPs.

A large body of work investigates defect prediction using empirical techniques; we refer the reader to a survey by Catal et al. [10].

7 Conclusion and future work

We examined the characteristics of well-known vulnerability rewards programs (VRPs) by studying two such VRPs. Both programs appear economically efficient, comparing favorably to the cost of hiring full-time security researchers. The Chrome VRP features low expected payouts accompanied by high potential payouts, a strategy that appears to be effective in engaging a broad community of vulnerability researchers.

We hope that our study of these two VRPs serves as a valuable reference for software vendors aiming to evolve an existing VRP or start a new one. Potential future work on understanding VRPs includes economic modeling of VRPs; identifying typical patterns, trajectories, or phases in a VRP; and studying failed or unsuccessful VRPs to get a better sense of possible pitfalls in VRP development. Gathering and analyzing data from more VRPs will surely paint a more complete picture of their potential costs and benefits.

Acknowledgments

We are particularly grateful to Chris Evans and Dan Veditz for their help, encouragement, and feedback throughout the research. We also thank Chris Hofmann, Parisa Tabriz, Vern Paxson, Adrienne Felt, the anonymous reviewers, and our shepherd, Sam King, for their feedback on drafts of the paper.

This work was supported by Intel through the ISTC for Secure Computing; by the National Science Foundation under a Graduate Research Fellowship and grant numbers CCF-0424422, 0842695, and 0831501-CT-L; by the Air Force Office of Scientific Research under MURI awards FA9550-08-1-0352, FA9550-09-1-0539, and FA9550-12-1-0040; and by the Office of Naval Research under MURI grant no. N000140911081. 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 the NSF, the AFOSR, the ONR, or Intel.

References

- [1] ADAMSKI, L. Refresh of the Mozilla Security Bug Bounty Program, July 2010. <https://blog.mozilla.org/security/2010/07/15/refresh/>.
- [2] AUSTIN, A., AND WILLIAMS, L. One technique is not enough: A comparison of vulnerability discovery techniques. In *Empirical Software Engineering and Measurement (ESEM), 2011 International Symposium on* (2011), IEEE, pp. 97–106.
- [3] BARRETT, M. PayPal “Bug Bounty” Program for Security Researchers, June 2012. <https://www.thepaypalblog.com/2012/06/paypal-bug-bounty-program/>.
- [4] BARTH, A., JACKSON, C., REIS, C., AND TEAM, T. G. C. The Security Architecture of the Chromium Browser. Tech. rep., Stanford University, 2008.
- [5] BHATTACHARYYA, N., AND GARRETT, T. A. Why People Choose Negative Expected Return Assets - An Empirical Examination of a Utility Theoretic Explanation. *Federal Reserve Bank of St. Louis Working Paper Series* (March 2006). <http://research.stlouisfed.org/wp/2006/2006-014.pdf>.
- [6] BLINDU, E. Vulnerabilities reward programs, July 2012. <http://www.testalways.com/2012/07/13/vulnerabilities-reward-programs/>.
- [7] BUCHANAN, K., EVANS, C., REIS, C., AND SEPEZ, T. A Tale of Two Pwnies (Part 2), June 2012. <http://blog.chromium.org/2012/06/tale-of-two-pwnies-part-2.html>.
- [8] BUCHANAN, K., EVANS, C., REIS, C., AND SEPEZ, T. Show off Your Security Skills: Pwn2Own and Pwnium 3, January 2013. <http://blog.chromium.org/2013/01/show-off-your-security-skills-pwn2own.html>.
- [9] CARETTONI, L. “No More Free Bugs” Initiative, October 2011. <http://blog.nibblesec.org/2011/10/no-more-free-bugs-initiatives.html>.
- [10] CATAL, C., AND DIRI, B. A systematic review of software fault prediction studies. *Expert Systems with Applications* 36, 4 (2009), 7346–7354.
- [11] Chromium Development Calendar and Release Info. <http://www.chromium.org/developers/calendar>.
- [12] Severity Guidelines for Security Issues. <https://sites.google.com/a/chromium.org/dev/developers/severity-guidelines>.
- [13] Chromium Bug Tracker, 2013. <http://crbug.com>.
- [14] Security: Pwnium 2 tcmalloc profile bug, 2012. <http://crbug.com/154983>.
- [15] DAVIS, N. Secure Software Development Life Cycle Processes, July 2012. <https://buildsecurityin.us-cert.gov/bci/articles/knowledge/sdlc/326-BSI.html>.
- [16] Defense in Depth. http://www.nsa.gov/ia/_files/support/defenseindepth.pdf.
- [17] MozillaWiki: Electrolysis, April 2011. <https://wiki.mozilla.org/Electrolysis>.
- [18] EDMUNDSON, A., HOLTKAMP, B., RIVERA, E., FINIFTER, M., METTLER, A., AND WAGNER, D. An Empirical Study on the Effectiveness of Security Code Review. In *Proceedings of the International Symposium on Engineering Secure Software and Systems* (March 2013).
- [19] EVANS, C. Celebrating Six Months of Chromium Security Rewards, July 2010. <http://blog.chromium.org/2010/07/celebrating-six-months-of-chromium.html>.
- [20] EVANS, C. Bug bounties vs. black (& grey) markets, May 2011. <http://scarybeastsecurity.blogspot.com/2011/05/bug-bounties-vs-black-grey-markets.html>.

- [21] EVANS, C. Personal Communication, March 2013.
- [22] EVANS, C., GROSSE, E., MEHTA, N., MOORE, M., ORMANDY, T., TINNES, J., ZALEWSKI, M., AND TEAM, G. S. Rebooting Responsible Disclosure: a focus on protecting end users, July 2010. <http://googleonlinesecurity.blogspot.com/2010/07/rebooting-responsible-disclosure-focus.html>.
- [23] EVANS, C., AND SCHUH, J. Pwnium: rewards for exploits, February 2012. <http://blog.chromium.org/2012/02/pwnium-rewards-for-exploits.html>.
- [24] FINIFTER, M., AND WAGNER, D. Exploring the relationship between web application development tools and security. In *USENIX conference on Web application development* (2011).
- [25] MozillaWiki: RapidRelease/Calendar, January 2013. <https://wiki.mozilla.org/RapidRelease/Calendar>.
- [26] FISHER, D. Microsoft Says No to Paying Bug Bounties, July 2010. <http://threatpost.com/microsoft-says-no-paying-bug-bounties-072210/>.
- [27] Chrome Releases: Stable Updates. <http://googlechromereleases.blogspot.com/search/label/Stable%20updates>.
- [28] GORENC, B. Pwn2Own 2013, January 2013. <http://dvlabs.tippingpoint.com/blog/2013/01/17/pwn2own-2013>.
- [29] HATTON, L. Predicting the Total Number of Faults Using Parallel Code Inspections. <http://www.leshatton.org/2005/05/total-number-of-faults-using-parallel-code-inspections/>, May 2005.
- [30] HOFMANN, C. Personal Communication, March 2013.
- [31] HOLLER, C. Trying new code analysis techniques, January 2012. <https://blog.mozilla.org/decoder/2012/01/27/trying-new-code-analysis-techniques/>.
- [32] Creating a Computer Security Incident Response Team: A Process for Getting Started, February 2006. <https://www.cert.org/csirts/Creating-A-CSIRT.html>.
- [33] MATTHEW FINIFTER AND DEVADATTA AKHAWA AND DAVID WAGNER. Chrome and Firefox VRP datasets, June 2013. <https://gist.github.com/devd/a62f2afae9f1c93397f5>.
- [34] The MEGA Vulnerability Reward Program, February 2013. https://mega.co.nz/#blog_6.
- [35] MEIN, A., AND EVANS, C. Dosh4Vulns: Google's Vulnerability Reward Programs”, March 2011.
- [36] MELVEN, I. MozillaWiki: Features/Security/Low rights Firefox, August 2012. https://wiki.mozilla.org/Features/Security/Low_rights_Firefox.
- [37] MILLER, C. The legitimate vulnerability market: the secretive world of 0-day exploit sales. In *WEIS* (2007).
- [38] MILLS, E. Facebook launches bug bounty program, July 2011. http://news.cnet.com/8301-27080_3-20085163-245/facebook-launches-bug-bounty-program/.
- [39] MOZILLA BUGZILLA. Bug 790923: Content process sandboxing via seccomp filter. https://bugzilla.mozilla.org/show_bug.cgi?id=790923.
- [40] MOZILLA FOUNDATION. Mozilla Foundation Security Advisories, January 2013. <https://www.mozilla.org/security/announce/>.
- [41] Netscape announces "netscape bugs bounty" with release of netscape navigator 2.0 beta. The Internet Archive. <http://web.archive.org/web/19970501041756/www101.netscape.com/newsref/pr/newsrelease48.html>.
- [42] NEUHAUS, S., AND PLATTNER, B. Software security economics: Theory, in practice. In *WEIS* (2012).
- [43] NEUHAUS, S., ZIMMERMANN, T., HOLLER, C., AND ZELLER, A. Predicting vulnerable software components. In *Proceedings of the 14th ACM conference on Computer and communications security* (2007), ACM, pp. 529–540.
- [44] National Vulnerability Database. <http://nvd.nist.gov/>.
- [45] OBES, J. L., AND SCHUH, J. A Tale of Two Pwnies (Part 1), May 2012. <http://blog.chromium.org/2012/05/tale-of-two-pwnies-part-1.html>.
- [46] Understanding Operational Security. <http://www.cisco.com/web/about/security/intelligence/opsecurity.html>.
- [47] OZMENT, A., AND SCHECHTER, S. E. Milk or wine: does software security improve with age. In *In USENIX-SS06: Proceedings of the 15th conference on USENIX Security Symposium* (2006), USENIX Association.
- [48] RAYMOND, E. S. *The Cathedral and the Bazaar*, 1st ed. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 1999.
- [49] RESCORLA, E. Is finding security holes a good idea? *IEEE Security & Privacy* 3, 1 (2005), 14–19.
- [50] CERT/CC Vulnerability Disclosure Policy, November 2012. https://www.cert.org/kb/vul_disclosure.html.
- [51] Chromium bug tracker: Sandbox bypasses found in review, 2013. <http://goo.gl/13z1R>.
- [52] SCHOLTE, T., BALZAROTTI, D., AND KIRDA, E. Quo vadis? a study of the evolution of input validation vulnerabilities in web applications. *Financial Cryptography and Data Security* (2012), 284–298.
- [53] Secunia Vulnerability Coordination Reward Program (SVCRP). <https://secunia.com/community/research/svcrp/>.
- [54] THE BLUEHAT TEAM. Microsoft Security Bounty Programs. <http://www.microsoft.com/security/msrc/report/bountyprograms.aspx>, June 2013.
- [55] THE CHROMIUM AUTHORS. Vulnerability Rewards Program: Rewards FAQ, 2010. <http://goo.gl/m1MdV>.
- [56] VEDITZ, D. Personal Communication, February 2013.
- [57] Vulnerability Remediation, September 2010. <https://www.cert.org/vuls/remediation.html>.

Secure Outsourced Garbled Circuit Evaluation for Mobile Devices

Henry Carter

Georgia Institute of Technology
carterh@gatech.edu

Patrick Traynor

Georgia Institute of Technology
traynor@cc.gatech.edu

Benjamin Mood

University of Oregon
bmood@cs.uoregon.edu

Kevin Butler

University of Oregon
butler@cs.uoregon.edu

Abstract

Garbled circuits provide a powerful tool for jointly evaluating functions while preserving the privacy of each user’s inputs. While recent research has made the use of this primitive more practical, such solutions generally assume that participants are symmetrically provisioned with massive computing resources. In reality, most people on the planet only have access to the comparatively sparse computational resources associated with their mobile phones, and those willing and able to pay for access to public cloud computing infrastructure cannot be assured that their data will remain unexposed. We address this problem by creating a new SFE protocol that allows mobile devices to securely outsource the majority of computation required to evaluate a garbled circuit. Our protocol, which builds on the most efficient garbled circuit evaluation techniques, includes a new outsourced oblivious transfer primitive that requires significantly less bandwidth and computation than standard OT primitives and outsourced input validation techniques that force the cloud to prove that it is executing all protocols correctly. After showing that our extensions are secure in the malicious model, we conduct an extensive performance evaluation for a number of standard SFE test applications as well as a privacy-preserving navigation application designed specifically for the mobile use-case. Our system reduces execution time by 98.92% and bandwidth by 99.95% for the edit distance problem of size 128 compared to non-outsourced evaluation. These results show that even the least capable devices are capable of evaluating some of the largest garbled circuits generated for any platform.

1 Introduction

Secure Function Evaluation (SFE) allows two parties to compute the result of a function without either side having to expose their potentially sensitive inputs to the other. While considered a generally theoretical curios-

ity even after the discovery of Yao’s garbled circuit [43], recent advances in this space have made such computation increasingly practical. Today, functions as complex as AES-128 and approaching one billion gates in size are possible at reasonable throughputs, even in the presence of a malicious adversary.

While recent research has made the constructions in this space appreciably more performant, the majority of related work makes a crucial assumption - *that both parties are symmetrically provisioned with massive computing resources*. For instance, Kreuter et al. [25] rely on the Ranger cluster at the Texas Advanced Computing Center to compute their results using 512 cores. In reality, the extent of a user’s computing power may be their mobile phone, which has many orders of magnitude less computational ability. Moreover, even with access to a public compute cloud such as Amazon EC2 or Windows Azure, the sensitive nature of the user’s data and the history of data leakage from cloud services [40, 42] prevent the direct porting of known SFE techniques.

In this paper, we develop mechanisms for the secure outsourcing of SFE computation from constrained devices to more capable infrastructure. Our protocol maintains the privacy of both participant’s inputs and outputs while significantly reducing the computation and network overhead required by the mobile device for garbled circuit evaluation. We develop a number of extensions to allow the mobile device to check for malicious behavior from the circuit generator or the cloud and a novel Outsourced Oblivious Transfer for sending garbled input data to the cloud. We then implement the new protocol on a commodity mobile device and reasonably provisioned servers and demonstrate significant performance improvements over evaluating garbled circuits directly on the mobile device.

We make the following contributions:

- **Outsourced oblivious transfer & outsourced consistency checks:** Instead of blindly trusting the cloud with sensitive inputs, we develop a highly

efficient Outsourced Oblivious Transfer primitive that allows mobile devices to securely delegate the majority of computation associated with oblivious transfers. We also provide mechanisms to outsource consistency checks to prevent a malicious circuit generator from providing corrupt garbled values. These checks are designed in such a way that the computational load is almost exclusively on the cloud, but cannot be forged by a malicious or “lazy” cloud. We demonstrate that both of our additions are secure in the malicious model as defined by Kamara et al. [21].

- **Performance Analysis:** Extending upon the implementation by Kreuter et al. [25], we conduct an extensive performance analysis against a number of simple applications (e.g., edit distance) and cryptographic benchmarks (e.g., AES-128). Our results show that outsourcing SFE provides improvements to both execution time and bandwidth overhead. For the edit distance problem of size 128, we reduce execution time by 98.92% and bandwidth by 99.95% compared to direct execution without outsourcing on the mobile device.
- **Privacy Preserving Navigation App:** To demonstrate the practical need for our techniques, we design and implement an outsourced version of Dijkstra’s shortest path algorithm as part of a Navigation mobile app. Our app provides directions for a Presidential motorcade without exposing its location, destination, or known hazards that should be avoided (but remain secret should the mobile device be compromised). *The optimized circuits generated for this app represent the largest circuits evaluated to date.* Without our outsourcing techniques, such an application is far too processor, memory and bandwidth intensive for any mobile phone.

While this work is similar in function and provides equivalent security guarantees to the Salus protocols recently developed by Kamara et al. [21], our approach is dramatically different. The Salus protocol framework builds their scheme on a completely different assumption, specifically, that they are outsourcing work from low-computation devices with *high communication bandwidth*. With provider-imposed bandwidth caps and relatively slow and unreliable cellular data connections, this is not a realistic assumption when developing solutions in the mobile environment. Moreover, rather than providing a proof-of-concept work demonstrating that offloading computation is possible, this work seeks to develop and thoroughly demonstrate the practical potential for evaluating large garbled circuits in a resource-constrained mobile environment.

The remainder of this work is organized as follows: Section 2 presents important related work and discusses

how this paper differs from Salus; Section 3 provides cryptographic assumptions and definitions; Section 4 formally describes our protocols; Section 5 provides security discussion - we direct readers to our technical report [6] for full security proofs; Section 6 shows the results of our extensive performance analysis; Section 7 presents our privacy preserving navigation application for mobile phones; and Section 8 provides concluding remarks.

2 Related Work

Beginning with Fairplay [32], several secure two-party computation implementations and applications have been developed using Yao garbled circuits [43] in the semi-honest adversarial model [3, 15, 17, 19, 26, 28, 31, 38]. However, a malicious party using corrupted inputs or circuits can learn more information about the other party’s inputs in these constructions [23]. To resolve these issues, new protocols have been developed to achieve security in the malicious model, using cut-and-choose constructions [30], input commitments [41], and other various techniques [22, 34]. To improve the performance of these schemes in both the malicious and semi-honest adversarial models, a number of circuit optimization techniques have also been developed to reduce the cost of generating and evaluating circuits [8, 11, 24, 35]. Kreuter et al. [25] combined several of these techniques into a general garbled circuit protocol that is secure in the malicious model and can efficiently evaluate circuits on the order of billions of gates using parallelized server-class machines. This SFE protocol is currently the most efficient implementation that is fully secure in the malicious model. (The dual execution construction by Huang et al. leaks one bit of input [16].)

Garbled circuit protocols rely on oblivious transfer schemes to exchange certain private values. While several OT schemes of various efficiencies have been developed [1, 30, 36, 39], Ishai et al. demonstrated that any of these schemes can be extended to reduce k^c oblivious transfers to k oblivious transfers for any given constant c [18]. Using this extension, exchanging potentially large inputs to garbled circuits became much less costly in terms of cryptographic operations and network overhead. Even with this drastic improvement in efficiency, oblivious transfers still tend to be a costly step in evaluating garbled circuits.

Currently, the performance of garbled circuit protocols executed directly on mobile devices has been shown to be feasible only for small circuits in the semi-honest adversarial model [5, 13]. While outsourcing general computation to the cloud has been widely considered for improving the efficiency of applications running on mobile devices, the concept has yet to be widely applied to cryp-

tographic constructions. Green et al. began exploring this idea by outsourcing the costly decryption of ABE ciphertexts to server-class machines while still maintaining data privacy [12]. Considering the costs of exchanging inputs and evaluating garbled circuits securely, an outsourcing technique would be useful in allowing limited capability devices to execute SFE protocols. Naor et al. [37] develop an oblivious transfer technique that sends the chooser’s private selections to a third party, termed a proxy. While this idea is applied to a limited application in their work, it could be leveraged more generally into existing garbled circuit protocols. Our work develops a novel extension to this technique to construct a garbled circuit evaluation protocol that securely outsources computation to the cloud.

In work performed concurrently and independently from our technique, Kamara et al. recently developed two protocols for outsourcing secure multiparty computation to the cloud in their Salus system [21]. While their work achieves similar functionality to ours, we distinguish our work in the following ways: first, their protocol is constructed with the assumption that they are outsourcing work from devices with low-computation but high-bandwidth capabilities. With cellular providers imposing bandwidth caps on customers and cellular data networks providing highly limited data transmission speed, we construct our protocol without this assumption using completely different cryptographic constructions. Second, their work focuses on demonstrating outsourced SFE as a proof-of-concept. Our work offers a rigorous performance analysis on mobile devices, and outlines a practical application that allows a mobile device to participate in the evaluation of garbled circuits that are orders of magnitude larger than those evaluated in the Salus system. Finally, their protocol that is secure in the malicious model requires that all parties share a secret key, which must be generated in a secure fashion before the protocol can be executed. Our protocol does not require any shared information prior to running the protocol, reducing the overhead of performing a multiparty fair coin tossing protocol *a priori*. While our work currently considers only the two-party model, by not requiring a preliminary multiparty fair coin toss, expanding our protocol to more parties will not incur the same expense as scaling such a protocol to a large number of participants. To properly compare security guarantees, we apply their security definitions in our analysis.

3 Assumptions and Definitions

To construct a secure scheme for outsourcing garbled circuit evaluation, some new assumptions must be considered in addition to the standard security measures taken in a two-party secure computation. In this section, we discuss the intuition and practicality of assuming a non-

colluding cloud, and we outline our extensions on standard techniques for preventing malicious behavior when evaluating garbled circuits. Finally, we conclude the section with formal definitions of security.

3.1 Non-collusion with the cloud

Throughout our protocol, we assume that none of the parties involved will ever collude with the cloud. This requirement is based in theoretical bounds on the efficiency of garbled circuit evaluation and represents a realistic adversarial model. The fact that theoretical limitations exist when considering collusion in secure multiparty computation has been known and studied for many years [2, 7, 27], and other schemes considering secure computation with multiple parties require similar restrictions on who and how many parties may collude while preserving security [4, 9, 10, 20, 21]. Kamara et al. [21] observe that if an outsourcing protocol is secure when both the party generating the circuit and the cloud evaluating the circuit are malicious and colluding, this implies a secure two-party scheme where one party has sub-linear work with respect to the size of the circuit, which is currently only possible with fully homomorphic encryption. However, making the assumption that the cloud will not collude with the participating parties makes outsourcing securely a theoretical possibility. In reality, many cloud providers such as Amazon or Microsoft would not allow outside parties to control or affect computation within their cloud system for reasons of trust and to preserve a professional reputation. In spite of this assumption, we cannot assume the cloud will always be semi-honest. For example, our protocol requires a number of consistency checks to be performed by the cloud that ensure the participants are not behaving maliciously. Without mechanisms to force the cloud to make these checks, a “lazy” cloud provider could save resources by simply returning that all checks verified without actually performing them. Thus, our adversarial model encompasses a non-colluding but potentially malicious cloud provider that is hosting the outsourced computation.

3.2 Attacks in the malicious setting

When running garbled circuit based secure multiparty computation in the malicious model, a number of well-documented attacks exist. We address here how our system counters each.

Malicious circuit generation: In the original Yao garbled circuit construction, a malicious generator can garble a circuit to evaluate a function f' that is not the function f agreed upon by both parties and could compromise the security of the evaluator’s input. To counter this, we

employ an extension of the random seed technique developed by Goyal et al. [11] and implemented by Kreuter et al. [25]. Essentially, the technique uses a cut-and-choose, where the generator commits to a set of circuits that all presumably compute the same function. The parties then use a fair coin toss to select some of the circuits to be evaluated and some that will be re-generated and hashed by the cloud given the random seeds used to generate them initially. The evaluating party then inspects the circuit commitments and compares them to the hash of the regenerated circuits to verify that all the check circuits were generated properly.

Selective failure attack: If, when the generator is sending the evaluator’s garbled inputs during the oblivious transfer, he lets the evaluator choose between a valid garbled input bit and a corrupted garbled input, the evaluator’s ability to complete the circuit evaluation will reveal to the generator which input bit was used. To prevent this attack, we use the input encoding technique from Lindell and Pinkas [29], which lets the evaluator encode her input in such a way that a selective failure of the circuit reveals nothing about the actual input value. To prevent the generator from swapping garbled wire values, we use a commitment technique employed by Kreuter et al. [25].

Input consistency: Since multiple circuits are evaluated to ensure that a majority of circuits are correct, it is possible for either party to input different inputs to different evaluation circuits, which could reveal information about the other party’s inputs. To keep the evaluator’s inputs consistent, we again use the technique from Lindell and Pinkas [29], which sends all garbled inputs for every evaluation circuit in one oblivious transfer execution. To keep the generator’s inputs consistent, we use the malleable claw-free collection construction of shelat and Shen [41]. This technique is described in further detail in Section 4.

Output consistency: When evaluating a two-output function, we ensure that outputs of both parties are kept private from the cloud using an extension of the technique developed by Kiraz [23]. The outputs of both parties are XORed with random strings within the garbled circuit, and the cloud uses a witness-indistinguishable zero-knowledge proof as in the implementation by Kreuter et al. [25]. This allows the cloud to choose a majority output value without learning either party’s output or undetectably tampering with the output. At the same time, the witness-indistinguishable proofs prevent either party from learning the index of the majority circuit. This prevents the generator from learning anything by knowing which circuit evaluated to the majority output value.

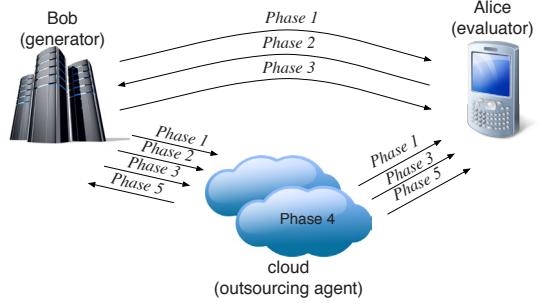


Figure 1: The complete outsourced SFE protocol.

3.3 Malleable claw-free collections

To prevent the generating party from providing different inputs for each evaluation circuit, we implement the malleable claw-free collections technique developed by shelat and Shen [41]. Their construction essentially allows the generating party to prove that all of the garbled input values were generated by exactly one function in a function pair, while the ability to find an element that is generated by both functions implies that the generator can find a claw. It is composed of a four-tuple of algorithms (G, D, F, R) , where G is the index selection algorithm for selecting a specific function pair, D is an algorithm for sampling from the domain of the function pair, F is the algorithm for evaluating the functions in the pair (in which it should be difficult to find a claw), and R is the “malleability” function. The function R maps elements from the domain of F to the range of F such that for $b \in \{0, 1\}$, any I in the range of G , and any m_1, m_2 in the domain of F , we have for the function indexed by I and b $f_I^b(m_1 \star m_2) = f_I^b(m_1) \diamond R_I(m_2)$, where \star and \diamond represent the group operations over the domain and range of F . We provide full definitions of their construction in our technical report [6].

3.4 Model and Definitions

The work of Kamara et al. [21] presents a definition of security based on the ideal-model/real-model security definitions common in secure multiparty computation. Because their definition formalizes the idea of a non-colluding cloud, we apply their definitions to our protocol for the two-party case in particular. We summarize their definitions below.

Real-model execution. The protocol takes place between two parties (P_1, P_2) executing the protocol and a server P_3 , where each of the executing parties provides input x_i , auxiliary input z_i , and random coins r_i and the server provides only auxiliary input z_3 and random coins r_3 . In the execution, there exists some subset of independent parties $(A_1, \dots, A_m), m \leq 3$ that are malicious adversaries. Each adversary corrupts one executing party and

does not share information with other adversaries. For all honest parties, let OUT_i be its output, and for corrupted parties let OUT_i be its view of the protocol execution. The i^{th} partial output of a real execution is defined as:

$$REAL^{(i)}(k, x; r) = \{OUT_j : j \in H\} \cup OUT_i$$

where H is the set of honest parties and r is all random coins of all players.

Ideal-model execution. In the ideal model, the setup of participants is the same except that all parties are interacting with a trusted party that evaluates the function. All parties provide inputs x_i , auxiliary input z_i , and random coins r_i . If a party is semi-honest, it provides its actual inputs to the trusted party, while if the party is malicious (and non-colluding), it provides arbitrary input values. In the case of the server P_3 , this means simply providing its auxiliary input and random coins, as no input is provided to the function being evaluated. Once the function is evaluated by the trusted third party, it returns the result to the parties P_1 and P_2 , while the server P_3 does not receive the output. If a party aborts early or sends no input, the trusted party immediately aborts. For all honest parties, let OUT_i be its output to the trusted party, and for corrupted parties let OUT_i be some value output by P_i . The i^{th} partial output of an ideal execution in the presence of some set of independent simulators is defined as:

$$IDEAL^{(i)}(k, x; r) = \{OUT_j : j \in H\} \cup OUT_i$$

where H is the set of honest parties and r is all random coins of all players. In this model, the formal definition of security is as follows:

Definition 1. A protocol securely computes a 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, \dots, A_3) , x , z , and for all $i \in [3]$:

$$\{REAL^{(i)}(k, x; r)\}_{k \in N} \stackrel{c}{\approx} \{IDEAL^{(i)}(k, x; r)\}_{k \in N}$$

Where $S = (S_1, \dots, S_3)$, $S_i = Sim_i(A_i)$, and r is random and uniform.

4 Protocol

Our protocol can be divided into five phases, illustrated in Figure 1. Given a circuit generator Bob, and an evaluating mobile device Alice, the protocol can be summarized as follows:

- Phase 1: Bob generates a number of garbled circuits, some of which will be checked, others will be evaluated. After Bob commits to the circuits, Alice and Bob use a fair coin toss protocol to select which circuits will be checked or evaluated. For the check

Inputs: Alice has a string of encoded input bits ea of length $\ell \cdot n$ and Bob has pairs of input values $(x_{0,j}, x_{1,j})$ for $j = 1 \dots \ell \cdot n$.

1. **Setup:** Alice generates random matrix T of size $\ell \cdot n \times t$, Bob generates random string s of length t .
2. **Primitive OT:** Alice and Bob execute t 1-out-of-2 oblivious transfers with Alice inputting $(T^i, T^i \oplus ea)$ and Bob inputting selection bits s (T^i denotes the i^{th} column of the T matrix). Bob sets the resulting columns as matrix Q .
3. **Permuting the output:** Alice generates random string p of length $\ell \cdot n$ and sends it to Bob.
4. **Encrypting the output:** Bob sets the encrypted output pairs $y_{0,j}, y_{1,j}$ where $y_{b,j} = x_{b,j} \oplus H_1(j, Q_j \oplus (b \cdot s))$ (Q_j denotes the j^{th} row of the Q matrix).
5. **Permuting the outputs:** Bob permutes the encrypted output pairs as $y_{0 \oplus p_j, j}, y_{1 \oplus p_j, j}$ and sends the resulting set of pairs Y to the cloud.
6. **Decrypting the output:** Alice sends $h = ea \oplus p$ and T to the cloud. The cloud recovers $z_j = y_{h_j, j} \oplus H_1(j, T_j)$ for $j = 1 \dots \ell \cdot n$ (T_j denotes the j^{th} row of the T matrix).

Figure 2: The Outsourced Oblivious Transfer protocol

circuits, Bob sends the random seeds used to generate the circuits to the cloud and the hashes of each circuit to Alice. These are checked to ensure that Bob has not constructed a circuit that is corrupted or deviates from the agreed-upon function.

- Phase 2: Alice sends her inputs to Bob via an outsourced oblivious transfer. Bob then sends the corresponding garbled inputs to the cloud. This allows the cloud to receive Alice’s garbled inputs without Bob or the cloud ever learning her true inputs.
- Phase 3: Bob sends his garbled inputs to the cloud, which verifies that they are consistent for each evaluation circuit. This prevents Bob from providing different inputs to different evaluation circuits.
- Phase 4: The cloud evaluates the circuit given Alice and Bob’s garbled inputs. Since the cloud only sees garbled values during the evaluation of the circuit, it never learns anything about either party’s input or output. Since both output values are blinded with one-time pads, they remain private even when the cloud takes a majority vote.
- Phase 5: The cloud sends the encrypted output values to Alice and Bob, who are guaranteed its authenticity through the use of commitments and zero-knowledge proofs.

4.1 Participants

Our protocols reference three different entities:

Evaluator: The evaluating party, called Alice, is assumed to be a mobile device that is participating in a secure two-party computation.

Generator: The party generating the garbled circuit, called Bob, is an application- or web- server that is the second party participating with Alice in the secure computation.

Proxy: The proxy, called cloud, is a third party that is performing heavy computation on behalf of Alice, but is not trusted to know her input or the function output.

4.2 Outsourced Protocol

Common inputs: a function $f(x,y)$ that is to be securely computed, a claw-free collection $(G_{CLW}, D_{CLW}, F_{CLW}, R_{CLW})$, two hash functions $H_1 : \{0,1\}^* \rightarrow \{0,1\}^n$ and $H_2 : \{0,1\}^* \rightarrow \{0,1\}^w$, a primitive 1-out-of-2 oblivious transfer protocol, a perfectly hiding commitment scheme $com_H(key, message)$, and security parameters for the number of circuits built k , the number of primitive oblivious transfers t , and the number of encoding bits for each of Alice's input wires ℓ .

Private inputs: The generating party Bob inputs a bit string b and a random string of bits b_r that is the length of the output string. The evaluating party Alice inputs a bit string a and a random string of bits a_r that is the length of the output string. Assume without loss of generality that all input and output strings are of length $|a| = n$.

Output: The protocol outputs separate private values fa for Alice and fb for Bob.

Phase 1: Circuit generation and checking

1. *Circuit preparation:* Before beginning the protocol, both parties agree upon a circuit representation of the function $f(a,b)$, where the outputs of the function may be defined separately for Alice and Bob as $f_A(a,b)$ and $f_B(a,b)$. The circuit must also meet the following requirements:
 - (a) Additional XOR gates must be added such that Bob's output is set to $fb = f_B(a,b) \oplus b_r$ and Alice's output is set to $fa = f_A(a,b) \oplus a_r$.
 - (b) For each of Alice's input bits, the input wire w_i is split into ℓ different input wires $w_{j,i}$ such that $w_i = w_{1,i} \oplus w_{2,i} \oplus \dots \oplus w_{\ell,i}$ following the input encoding scheme by Lindell and Pinkas [29]. This prevents Bob from correlating a selective failure attack with any of Alice's input bit values.
2. *Circuit garbling:* the generating party, Bob, constructs k garbled circuits using a circuit garbling

technique $Garble(\cdot, \cdot)$. When given a circuit representation C of a function and random coins rc , $Garble(C, rc)$ outputs a garbled circuit GC that evaluates C . Given the circuit C and random coins $rc_1 \dots rc_k$, Bob generates garbled circuits $Garble(C, rc_i) = GC_i$ for $i = 1 \dots k$. For Bob's j^{th} input wire on the i^{th} circuit, Bob associates the value $H_2(\beta_{b,j,i})$ with the input value b , where $\beta_{b,j,i} = F_{CLW}(b, I, \alpha_{b,j,i})$. For Alice's j^{th} input wire, Bob associates the value $H_2(\delta_{b,j,i})$ with the input value b , where $\delta_{b,j,i} = F_{CLW}(b, I, \gamma_{b,j,i})$. All the values $\alpha_{b,j,i}$ and $\gamma_{b,j,i}$ for $b = \{0,1\}$, $j = 1 \dots n$, $i = 1 \dots k$ are selected randomly from the domain of the claw-free pair using D .

3. *Circuit commitment:* Bob generates commitments for all circuits by hashing $H_1(GC_i) = HC_i$ for $i = 1 \dots k$. Bob sends these hashes to Alice. In addition, for every output wire $w_{b,j,i}$ for $b = \{0,1\}$, $j = 1 \dots n$ and $i = 1 \dots k$, Bob generates commitments $CO_{j,i} = com_H(ck_{j,i}, (H_2(w_{0,j,i}), H_2(w_{1,j,i})))$ using commitment keys $ck_{j,i}$ for $j = 1 \dots n$ and $i = 1 \dots k$ and sends them to both Alice and the cloud.
4. *Input label commitment:* Bob commits to Alice's garbled input values as follows: for each generated circuit $i = 1 \dots k$ and each of Alice's input wires $j = 1 \dots \ell \cdot n$, Bob creates a pair of commitment keys $ik_{0,j,i}, ik_{1,j,i}$ and commits to the input wire label seeds $\delta_{0,j,i}$ and $\delta_{1,j,i}$ as $CI_{b,j,i} = com_H(ik_{b,j,i}, \delta_{b,j,i})$. For each of Alice's input wires $j = 1 \dots \ell \cdot n$, Bob randomly permutes the commitments within the pair $CI_{0,j,i}, CI_{1,j,i}$ across every $i = 1 \dots k$. This prevents the cloud from correlating the location of the commitment with Alice's input value during the OOT phase.
5. *Cut and choose:* Alice and Bob then run a fair coin toss protocol to agree on a set of circuits that will be evaluated, while the remaining circuits will be checked. The coin toss generates a set of indices $Chk \subset \{1, \dots, k\}$ such that $|Chk| = \frac{3}{5}k$, as in shelat and Shen's cut-and-choose protocol [41]. The remaining indices are placed in the set Evl for evaluation, where $|Evl| = e = \frac{2}{5}k$. For every $i \in Chk$, Bob sends rc_i and the values $[\alpha_{b,1,i}, \dots, \alpha_{b,n,i}]$ and $[\gamma_{b,1,i}, \dots, \gamma_{b,\ell,n,i}]$ for $b = \{0,1\}$ to the cloud. Bob also sends all commitment keys $ck_{j,i}$ for $j = 1 \dots n$ and $i \in Chk$ to the cloud. Finally, Bob sends the commitment keys $ik_{b,j,i}$ for $b = \{0,1\}$, $i \in Chk$, and $j = 1 \dots \ell \cdot n$ to the cloud. The cloud then generates $Garble(C, rc_i) = GC'_i$ for $i \in Chk$. For each $i \in Chk$, the cloud then hashes each check circuit $H_1(GC'_i) = HC'_i$ and checks that:

- each commitment $CO_{j,i}$ for $j = 1 \dots n$ is well formed
- the value $H_2(\beta_{b,j,i})$ is associated with the input value b for Bob's j^{th} input wire
- the value $H_2(\delta_{b,j,i})$ is associated with the input value b for Alice's j^{th} input wire
- for every bit value b and input wire j , the values committed in $CI_{b,j,i}$ are correct

If any of these checks fail, the cloud immediately aborts. Otherwise, it sends the hash values HC'_i for $i \in Chk$ to Alice. For every $i \in Chk$, Alice checks if $HC_i = HC'_i$. If any of the hash comparisons fail, Alice aborts.

Phase 2: Outsourced Oblivious Transfer (OOT)

1. *Input encoding:* For every bit $j = 1 \dots n$ in her input a , Alice sets encoded input ea_j as a random string of length ℓ such that $ea_{1,j} \oplus ea_{2,j} \oplus \dots \oplus ea_{\ell,j} = a_j$ for each bit in ea_j . This new encoded input string ea is of length $\ell \cdot n$.
2. *OT setup:* Alice initializes an $\ell \cdot n \times t$ matrix T with uniformly random bit values, while Bob initializes a random bit vector s of length t . See Figure 2 for a more concise view.
3. *Primitive OT operations:* With Alice as the sender and Bob as the chooser, the parties initiate t 1-out-of-2 oblivious transfers. Alice's input to the i^{th} instance of the OT is the pair $(T^i, T^i \oplus ea)$ where T^i is the i^{th} column of T , while Bob's input is the i^{th} selection bit from the vector s . Bob organizes the t selected columns as a new matrix Q .
4. *Permuting the selections:* Alice generates a random bit string p of length $\ell \cdot n$, which she sends to Bob.
5. *Encrypting the commitment keys:* Bob generates a matrix of keys that will open the committed garbled input values and proofs of consistency as follows: for Alice's j^{th} input bit, Bob creates a pair $(x_{0,j}, x_{1,j})$, where $x_{b,j} = [ik_{b,j,Evl_1}, ik_{b,j,Evl_2}, \dots, ik_{b,j,Evl_e}] \parallel [\gamma_{b,j,Evl_2} \star (\gamma_{b,j,Evl_1})^{-1}, \gamma_{b,j,Evl_3} \star (\gamma_{b,j,Evl_1})^{-1}, \dots, \gamma_{b,j,Evl_e} \star (\gamma_{b,j,Evl_1})^{-1}]$ and Evl_i denotes the i^{th} index in the set of evaluation circuits. For $j = 1 \dots \ell \cdot n$, Bob prepares $(y_{0,j}, y_{1,j})$ where $y_{b,j} = x_{b,j} \oplus H_1(j, Q_j \oplus (b \cdot s))$. Here, Q_j denotes the j^{th} row in the Q matrix. Bob permutes the entries using Alice's permutation vector as $(y_{0 \oplus p_{j,j}}, y_{1 \oplus p_{j,j}})$. Bob sends this permuted set of ciphertexts Y to the cloud.
6. *Receiving Alice's garbled inputs:* Alice blinds her input as $h = ea \oplus p$ and sends h and T to the cloud. The cloud recovers the commitment keys

and consistency proofs $x_{b,j} = y_{h,j} \oplus H_1(j, T_j)$ for $j = 1 \dots \ell \cdot n$. Here, h_j denotes the j^{th} bit of the string h and T_j denotes the j^{th} row in the T matrix. Since for every $j \in Evl$, the cloud only has the commitment key for the b garbled value (not the $b \oplus 1$ garbled value), the cloud can correctly decommit only the garbled labels corresponding to Alice's input bits.

7. *Verifying consistency across Alice's inputs:* Given the decommitted values $[\delta_{b,1,i}, \dots, \delta_{b,\ell \cdot n,i}]$ and the modified pre images $[\gamma_{b,j,Evl_2} \star (\gamma_{b,j,Evl_1})^{-1}, \gamma_{b,j,Evl_3} \star (\gamma_{b,j,Evl_1})^{-1}, \dots, \gamma_{b,j,Evl_e} \star (\gamma_{b,j,Evl_1})^{-1}]$, the cloud checks that:

$$\delta_{b,j,i} = \delta_{b,j,Evl_1} \diamond R_{CLW}(I, \gamma_{b,j,i} \star (\gamma_{b,j,Evl_1})^{-1})$$

for $i = 2 \dots e$. If any of these checks fails, the cloud aborts the protocol.

Phase 3: Generator input consistency check

1. *Delivering inputs:* Bob delivers the hash seeds for each of his garbled input values $[\beta_{b,1,1,i}, \beta_{b,2,2,i}, \dots, \beta_{b,n,n,i}]$ for every evaluation circuit $i \in Evl$ to the cloud, which forwards a copy of these values to Alice. Bob then proves the consistency of his inputs by sending the modified preimages $[\alpha_{b,j,Evl_2} \star (\alpha_{b,j,Evl_1})^{-1}, \alpha_{b,j,Evl_3} \star (\alpha_{b,j,Evl_1})^{-1}, \dots, \alpha_{b,j,Evl_e} \star (\alpha_{b,j,Evl_1})^{-1}]$ such that $F_{CLW}(b_i, I, \alpha_{b,i,j,i}) = \beta_{b,i,j,i}$ for $j = 1 \dots n$ and $i \in Evl$ such that GC_i was generated with the claw-free function pair indexed at I .
2. *Check consistency:* Alice then checks that all the hash seeds were generated by the same function by checking if:

$$\beta_{b,j,i} = \beta_{b,j,Evl_1} \diamond R_{CLW}(I, \alpha_{b,j,i} \star (\alpha_{b,j,Evl_1})^{-1})$$

for $i = 2 \dots e$. If any of these checks fails, Alice aborts the protocol.

Phase 4: Circuit evaluation

1. *Evaluating the circuit:* For each evaluation circuit, the cloud evaluates $GC_i(ga_i, gb_i)$ for $i \in Evl$ in the pipelined manner described by Kreuter et al. in [25]. Each circuit produces two garbled output strings, (gfa_i, gfb_i) .
2. *Checking the evaluation circuits:* Once these output have been computed, the cloud hashes each evaluation circuit as $H_1(GC_i) = HC'_i$ for $i \in Evl$ and sends these hash values to Alice. Alice checks that for every i , $HC_i = HC'_i$. If any of these checks do not pass, Alice aborts the protocol.

Phase 5: Output check and delivery

1. *Committing the outputs:* The cloud then generates random commitment keys ka_i, kb_i and commits the output values to their respective parties according to the commitment scheme defined by Kiraz [23], generating $CA_{j,i} = \text{commit}(ka_{j,i}, gfa_{j,i})$ and $CB_{j,i} = \text{commit}(kb_{j,i}, gfb_{j,i})$ for $j = 1\dots n$ and $i = 1\dots e$. The cloud then sends all CA to Alice and CB to Bob.
2. *Selection of majority output:* Bob opens the commitments $CO_{j,i}$ for $j = 1\dots n$ and $i = 1\dots e$ for both Alice and the Cloud. These commitments contain the mappings from the hash of each garbled output wire $H_2(w_{b,j,i})$ to real output values $b_{j,i}$ for $j = 1\dots n$ and $i = 1\dots e$. The cloud selects a circuit index maj such that the output of that circuit matches the majority of outputs for both Alice and Bob. That is, $f_{maj} = fa_i$ and $f_{b,maj} = fb_i$ for i in a set of indices IND that is of size $|IND| > \frac{\epsilon}{2}$
3. *Proof of output consistency:* Using the OR-proofs as described by Kiraz [23], the cloud proves to Bob that CB contains valid garbled output bit values based on the de-committed output values from the previous step. The cloud then performs the same proof to Alice for her committed values CA . Note that these proofs guarantee the output was generated by one of the circuits, but the value maj remains hidden from both Alice and Bob.
4. *Output release:* The cloud then decommits gfa_{maj} to Alice and gfb_{maj} to Bob. Given these garbled outputs and the bit values corresponding to the hash of each output wire, Alice recovers her output string fa , and Bob recovers his output string fb .
5. *Output decryption:* Alice recovers her output $f_A(a,b) = fa \oplus ar$, while Bob recovers $f_B(a,b) = fb \oplus br$.

5 Security Guarantees

In this section, we provide a summary of the security mechanisms used in our protocol and an informal security discussion of our new outsourced oblivious transfer primitive. Due to space limitations, we provide further discussion and proofs of security in our technical report [6].

Recall from Section 3 that there are generally four security concerns when evaluating garbled circuits in the malicious setting. To solve the problem of malicious circuit generation, we apply the random seed check variety of cut-&-choose developed by Goyal et al. [11]. To

solve the problem of selective failure attacks, we employ the input encoding technique developed by Lindell and Pinkas [29]. To prevent an adversary from using inconsistent inputs across evaluation circuits, we employ the witness-indistinguishable proofs from shelat and Shen [41]. Finally, to ensure the majority output value is selected and not tampered with, we use the XOR-and-prove technique from Kiraz [23] as implemented by Kreuter et al. [25]. In combination with the standard semi-honest security guarantees of Yao garbled circuits, these security extensions secure our scheme in the malicious security model.

Outsourced Oblivious Transfer: Our outsourced oblivious transfer is an extension of a technique developed by Naor et al. [37] that allows the chooser to select entries that are forwarded to a third party rather than returned to the chooser. By combining their concept of a proxy oblivious transfer with the semi-honest OT extension by Ishai et al. [18], our outsourced oblivious transfer provides a secure OT in the malicious model. We achieve this result for four reasons:

1. First, since Alice never sees the outputs of the OT protocol, she cannot learn anything about the garbled values held by the generator. This saves us from having to implement Ishai’s extension to prevent the chooser from behaving maliciously.
2. Since the cloud sees only random garbled values and Alice’s input blinded by a one-time pad, the cloud learns nothing about Alice’s true inputs.
3. Since Bob’s view of the protocol is almost identical to his view in Ishai’s standard extension, the same security guarantees hold (i.e., security against a malicious sender).
4. Finally, if Alice does behave maliciously and uses inconsistent inputs to the primitive OT phase, there is a negligible probability that those values will hash to the correct one-time pad keys for recovering either commitment key, which will prevent the cloud from de-committing the garbled input values.

It is important to note that this particular application of the OOT allows for this efficiency gain since the evaluation of the garbled circuit will fail if Alice behaves maliciously. By applying the maliciously secure extension by Ishai et al. [18], this primitive could be applied generally as an oblivious transfer primitive that is secure in the malicious model. Further discussion and analysis of this general application is outside the scope of this work.

We provide the following security theorem here, which gives security guarantees identical to the Salus protocol by Kamara et al. [21]. However, we use different constructions and require a completely different proof, which is available in our technical report [6].

Theorem 1. *The outsourced two-party SFE protocol securely computes a function $f(a,b)$ in the following two*

corruption scenarios: (1)The cloud is malicious and non-cooperative with respect to the rest of the parties, while all other parties are semi-honest, (2)All but one party is malicious, while the cloud is semi-honest.

6 Performance Analysis

We now characterize how garbled circuits perform in the constrained-mobile environment with and without outsourcing.¹ Two of the most important constraints for mobile devices are computation and bandwidth, and we show that order of magnitude improvements for both factors are possible with outsourced evaluation. We begin by describing our implementation framework and testbed before discussing results in detail.

6.1 Framework and Testbed

Our framework is based on the system designed by Kreuter et al. [25], hereafter referred to as *KSS* for brevity. We implemented the outsourced protocol and performed modifications to allow for the use of the mobile device in the computation. Notably, *KSS* uses MPI [33] for communication between the multiple nodes of the multi-core machines relied on for circuit evaluation. Our solution replaces MPI calls on the mobile device with sockets that communicate directly with the Generator and Proxy. To provide a consistent comparison, we revised the *KSS* codebase to allow for direct evaluation between the mobile device (the Evaluator) and the cloud-based Generator.²

Our deployment platform consists of two Dell R610 servers, each containing dual 6-core Xeon processors with 32 GB of RAM and 300 GB 10K RPM hard drives, running the Linux 3.4 kernel and connected as a VLAN on an internal 1 Gbps switch. These machines perform the roles of the Generator and Proxy, respectively, as described in Section 4.1. The mobile device acts as the Evaluator. We use a Samsung Galaxy Nexus phone with a 1.2 GHz dual-core ARM Cortex-A9 processor and 1 GB of RAM, running the Android 4.0 “Ice Cream Sandwich” operating system. We connect an Apple Airport Express wireless access point to the switch attaching the servers. The Galaxy Nexus communicates to the Airport Express over an 802.11n 54Mbps WiFi connection in an isolated environment to minimize co-channel interference. All tests are run 10 times with error bars on figures representing 95% confidence intervals.

¹We contacted the authors of the Salus protocol [21] in an attempt to acquire their framework to compare the performance of their scheme with ours, but they were unable to release their code.

²The full technical report [6] describes a comprehensive list of modifications and practical improvements made to *KSS*, including fixes that were added back into the codebase of *KSS* by the authors. We thank those authors for their assistance.

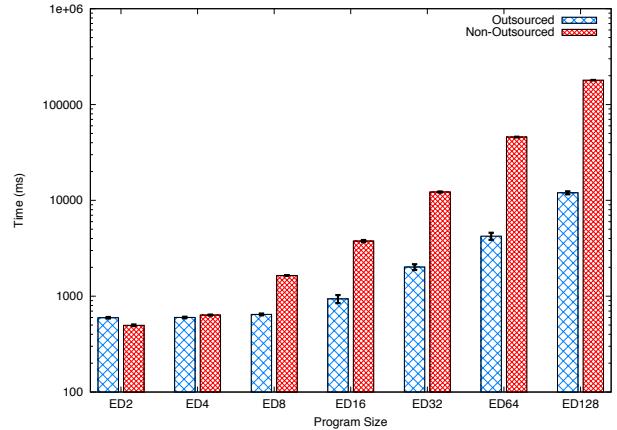


Figure 3: Execution time for the Edit Distance program of varying input sizes, with 2 circuits evaluated.

We measured both the total execution time of the programs and microbenchmarks for each program. All results are from the phone’s standpoint. We do not measure the time the programs take to compile as we used the standard compiler from Kreuter et al. For our microbenchmarks, the circuit garbling and evaluation pair is referred to as the ‘evaluation’.

6.2 Execution Time

Our tests evaluated the following problems:

Millionaires: This problem models the comparison of two parties comparing their net worth to determine who has more money without disclosing the actual values. We perform the test on input values ranging in size from 4 to 8192 bits.

Edit (Levenshtein) Distance: This is a string comparison algorithm that compares the number of modifications required to convert one string into another. We performed the comparison based on the circuit generated by Jha et al. [19] for strings sized between 4 and 128 bytes.

Set Intersection: This problem matches elements between the private sets of two parties without learning anything beyond the intersecting elements. We base our implementation on the SCS-WN protocol proposed by Huang et al. [14], and evaluate for sets of size 2 to 128.

AES: We compute AES with a 128-bit key length, based on a circuit evaluated by Kreuter et al. [25].

Figure 3 shows the result of the edit distance computation for input sizes of 2 to 128 with two circuits evaluated. This comparison represents worst-case operation due to the cost of setup for a small number of small circuits—with input size 2, the circuit is only 122 gates in size. For larger input sizes, however, outsourced computation becomes significantly faster. Note that the graph is logarithmic such that by the time strings of size 32 are evaluated, the outsourced execution is over 6 times

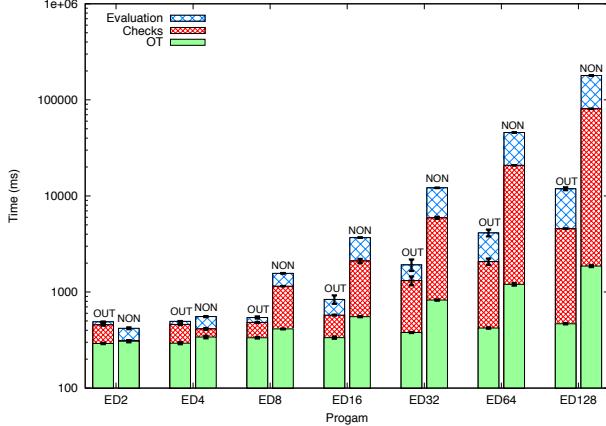


Figure 4: Execution time for significant stages of garbled circuit computation for outsourced and non-outsourced evaluation. The Edit Distance program is evaluated with variable input sizes for the two-circuit case.

faster than non-outsourced execution, while for strings of size 128 (comprising over 3.4 million gates), outsourced computation is over 16 times faster.

The reason for this becomes apparent when we examine Figure 4. There are three primary operations that occur during the SFE transaction: the oblivious transfer (OT) of participant inputs, the circuit commit (including the circuit consistency check), and the circuit generation and evaluation pair. As shown in the figure, the OT phase takes 292 ms for input size 2, but takes 467 ms for input size 128. By contrast, in the non-outsourced execution, the OT phase takes 307 ms for input size 2, but increases to 1860 ms for input size 128. The overwhelming factor, however, is the circuit evaluation phase. It increases from 34 ms (input size 2) to 7320 ms (input size 128) for the outsourced evaluation, a 215 factor increase. For non-outsourced execution however, this phase increases from 108 ms (input size 2) to 98800 ms (input size 128), a factor of 914 increase.

6.3 Evaluating Multiple Circuits

The security parameter for the garbled circuit check is $2^{-0.32k}$ [25], where k is the number of generated circuits. To ensure a sufficiently low probability (2^{-80}) of evaluating a corrupt circuit, 256 circuits must be evaluated. However, there are increasing execution costs as increasing numbers of circuits are generated. Figure 5 shows the execution time of the Edit Distance problem of size 32 with between 2 and 256 circuits evaluated. In the outsourced scheme, costs rise as the number of circuits evaluated increases. Linear regression analysis shows we can model execution time T as a function of the number of evaluated circuits k with the equation $T = 243.2k + 334.6$ ms, with a coef-

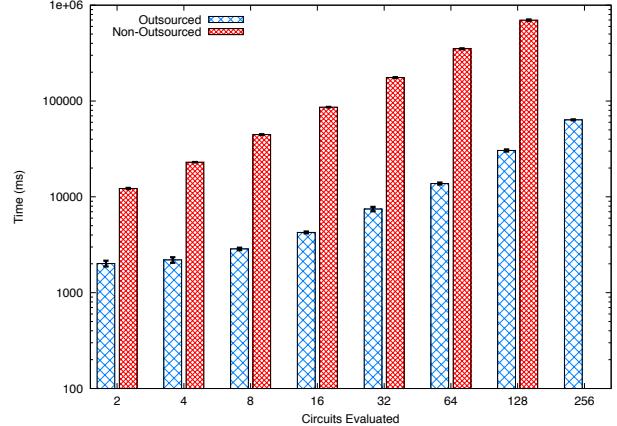


Figure 5: Execution time for the Edit Distance problem of size 32, with between 2 and 256 circuits evaluated. In the non-outsourced evaluation scheme, the mobile phone runs out of memory evaluating 256 circuits.

ficient of determination R^2 of 0.9971. However, note that in the non-outsourced scheme, execution time increases over 10 times as quickly compared to outsourced evaluation. Regression analysis shows execution time $T = 5435.7k + 961$ ms, with $R^2 = 0.9998$. Because in this latter case, the mobile device needs to perform all computation locally as well as transmit all circuit data to the remote parties, these costs increase rapidly. Figure 6 provides more detail about each phase of execution. Note that the OT costs are similar between outsourced and non-outsourced execution for this circuit size, but that the costs of consistency checks and evaluation vastly increase execution time for non-outsourced execution.

Note as well that in the non-outsourced scheme, there are no reported values for 256 circuits, as the Galaxy Nexus phone ran out of memory before the execution completed. We observe that a single process on the phone is capable of allocating 512 MB of RAM before the phone would report an out of memory error, providing insight into how much intermediate state is required for non-outsourced evaluation. Thus, to handle circuits of any meaningful size with enough check circuits for a strong security parameter, the *only way* to be able to perform these operations is through outsourcing.

Table 1 presents the execution time of a representative subset of circuits that we evaluated. It spans circuits from small to large input size, and from 8 circuits evaluated to the 256 circuits required for a 2^{-80} security parameter. Note that in many cases it is impossible to evaluate the non-outsourced computation because of the mobile device's inability to store sufficient amounts of state. Note as well that particularly with complex circuits such as set intersection, even when the non-outsourced evaluation is capable of returning an answer, it can require orders of

Program	8 Circuits		32 Circuits		128 Circuits		256 Circuits	
	Outsourced	KSS	Outsourced	KSS	Outsourced	KSS	Outsourced	KSS
Millionaires 128	2150.0 ± 1%	6130.0 ± 0.6%	8210.0 ± 3%	23080.0 ± 0.6%	38100.0 ± 7%	91020.0 ± 0.8%	75700.0 ± 1%	180800.0 ± 0.5%
Millionaires 1024	4670.0 ± 6%	46290.0 ± 0.4%	17800.0 ± 1%	180500.0 ± 0.3%	75290.0 ± 1%	744500.0 ± 0.7%	151000.0 ± 1%	1507000.0 ± 0.5%
Millionaires 8192	17280.0 ± 0.9%	368800.0 ± 0.4%	76980.0 ± 0.5%	1519000.0 ± 0.4%	351300.0 ± 0.7%	-	880000.0 ± 20%	-
Edit Distance 2	1268.0 ± 0.9%	794.0 ± 1%	4060.0 ± 1%	2125.0 ± 0.7%	19200.0 ± 2%	7476.0 ± 0.5%	42840.0 ± 0.4%	14600.0 ± 0.8%
Edit Distance 32	2860.0 ± 3%	44610.0 ± 0.7%	7470.0 ± 5%	175600.0 ± 0.5%	30500.0 ± 3%	699000.0 ± 2%	63600.0 ± 1%	-
Edit Distance 128	12800.0 ± 2%	702400.0 ± 0.5%	30300.0 ± 2%	2805000.0 ± 0.8%	106200.0 ± 0.6%	-	213400.0 ± 0.3%	-
Set Intersection 2	1598.0 ± 0.8%	1856.0 ± 0.9%	5720.0 ± 0.7%	6335.0 ± 0.4%	26100.0 ± 2%	24420.0 ± 0.6%	56350.0 ± 0.8%	48330.0 ± 0.6%
Set Intersection 32	5200.0 ± 10%	96560.0 ± 0.6%	13800.0 ± 1%	400800.0 ± 0.6%	59400.0 ± 1%	-	125300.0 ± 0.9%	-
Set Intersection 128	24300.0 ± 2%	1398000.0 ± 0.4%	55400.0 ± 3%	5712000.0 ± 0.4%	1998000.0 ± 0.5%	-	395200.0 ± 0.8%	-
AES-128	2450.0 ± 2%	15040.0 ± 0.7%	9090.0 ± 5%	58920.0 ± 0.5%	39000.0 ± 2%	276200.0 ± 0.6%	81900.0 ± 1%	577900.0 ± 0.5%

Table 1: Execution time (in ms) of outsourced vs non-outsourced (KSS) evaluation for a subset of circuits. Results with a dash indicate evaluation that the phone was incapable of performing.

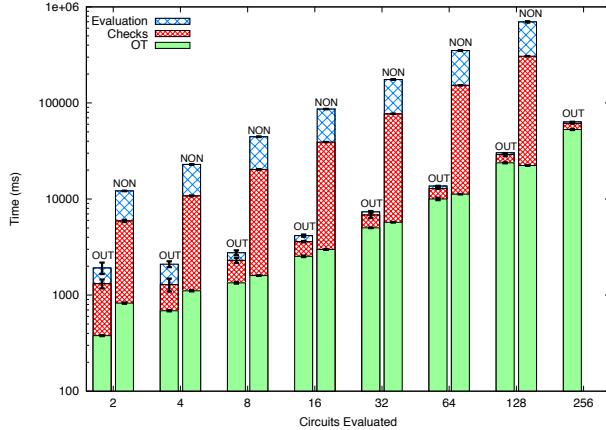


Figure 6: Microbenchmarks of execution time for Edit Distance with input size 32, evaluating from 2 to 256 circuits. Note that the y-axis is log-scale; consequently, the vast majority of execution time is in the check and evaluation phases for non-outsourced evaluation.

magnitude more time than with outsourced evaluation. For example, evaluating the set intersection problem with 128 inputs over 32 circuits requires just over 55 seconds for outsourced evaluation but *over an hour and a half* with the non-outsourced KSS execution scheme. Outsourced evaluation represents a time savings of 98.92%. For space concerns, we have omitted certain values; full results can be found in our technical report [6].

Multicore Circuit Evaluation We briefly note the effects of multicore servers for circuit evaluation. The servers in our evaluation each contain dual 6-core CPUs, providing 12 total cores of computation. The computation process is largely CPU-bound: while circuits on the servers are being evaluated, each core was reporting approximately 100% utilization. This is evidenced by regression analysis when evaluating between 2 and 12 circuit copies; we find that execution time $T = 162.6k + 1614.6$ ms, where k is the number of circuits evaluated, with a coefficient of determination R^2 of 0.9903. As the number of circuits to be evaluated increases beyond the number of available cores, the incremental costs of

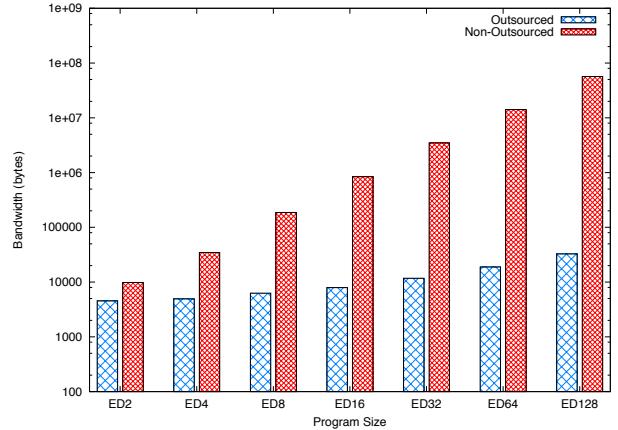


Figure 7: Bandwidth measurements from the phone to remote parties for the Edit Distance problem with varying input sizes, executing two circuits.

adding new circuits becomes higher; in our observation of execution time for 12 to 256 circuits, our regression analysis provided the equation $T = 247.4k - 410.6$ ms, with $R^2 = 0.998$. This demonstrates that evaluation of large numbers of circuits is optimal when every evaluated circuit can be provided with a dedicated core.

The results above show that as many-way servers are deployed in the cloud, it becomes easier to provide optimal efficiency computing outsourced circuits. A 256-core machine would be able to evaluate 256 circuits in parallel to provide the accepted standard 2^{-80} security parameter. Depending on the computation performed, there can be a trade-off between a slightly weaker security parameter and maintaining *optimal* evaluation on servers with lower degrees of parallelism. In our testbed, optimal evaluation with 12 cores provides a security parameter of $2^{-3.84}$. Clearly more cores would provide stronger security while keeping execution times proportional to our results. A reasonable trade-off might be 32 circuits, as 32-core servers are readily available. Evaluating 32 circuits provides a security parameter of $2^{-10.2}$, equivalent to the adversary having less than a $\frac{1}{512}$ chance of causing the evaluator to compute over a majority of corrupt circuits. Stronger security guarantees on less par-

Program	32 Circuits		Factor
	Outsourced	KSS	Improvement
Millionaires 128	336749	1445369	4.29X
Millionaires 1024	2280333	11492665	5.04X
Millionaires 8192	17794637	91871033	5.16X
Edit Distance 2	56165	117245	2.09X
Edit Distance 32	134257	41889641	312.01X
Edit Distance 128	350721	682955633	1947.29X
Set Intersection 2	117798	519670	4.41X
Set Intersection 32	1173844	84841300	72.28X
Set Intersection 128	4490932	1316437588	293.13X
AES-128	367364	9964576	27.12X

Table 2: Total Bandwidth (Bytes) transmitted to and from the phone during execution.

allel machines can be achieved at the cost of increasing execution time, as individual cores will not be dedicated to circuit evaluation. However, if a 256-core system is available, it will provide optimal results for achieving a 2^{-80} security parameter.

6.4 Bandwidth

For a mobile device, the costs of transmitting data are intrinsically linked to power consumption, as excess data transmission and reception reduces battery life. Bandwidth is thus a critical resource constraint. In addition, because of potentially uncertain communication channels, transmitting an excess of information can be a rate-limiting factor for circuit evaluation. Figure 7 shows the bandwidth measurement between the phone and remote parties for the edit distance problem with 2 circuits. When we compared execution time for this problem in Figure 3, we found that trivially small circuits could execute in less time without outsourcing. Note, however, that *there are no cases where the non-outsourced scheme consumes less bandwidth than with outsourcing*.

This is a result of the significant improvements garnered by using our outsourced oblivious transfer (OOT) construction described in Section 4. Recall that with the OOT protocol, the mobile device sends inputs for evaluation to the generator; however, after this occurs, the majority of computation until the final output verification from the cloud occurs between the generator and the cloud, with the mobile device only performing minor consistency checks. Figure 7 shows that the amount of data transferred increases only nominally compared to the non-outsourced protocol. Apart from the initial set of inputs transmitted to the generator, data demands are largely constant. This is further reflected in Table 2, which shows the vast bandwidth savings over the 32-circuit evaluation of our representative programs. In particular, for large, complex circuits, the savings are vast: outsourced AES-128 requires 96.3% less bandwidth, while set intersection of size 128 requires 99.7% less bandwidth than in the non-outsourced evalua-

tion. Remarkably, the edit distance 128 problem requires *99.95%, over 1900 times less bandwidth*, for outsourced execution. The full table is in our technical report [6].

The takeaway from our evaluation is simple: outsourcing the computation allows for faster and larger circuit evaluation than previously possible on a mobile device. Specifically, outsourcing allows users to evaluate garbled circuits with adequate malicious model security (256 circuits), which was previously not possible on mobile devices. In addition, outsourcing is by far the most efficient option if the bandwidth use of the mobile devices is a principle concern.

7 Evaluating Large Circuits

Beyond the standard benchmarks for comparing garbled circuit execution schemes, we aimed to provide compelling applications that exploit the mobile platform with large circuits that would be used in real-world scenarios. We discuss public-key cryptography and the Dijkstra shortest path algorithm, then describe how the latter can be used to implement a privacy-preserving navigation application for mobile phones.

7.1 Large Circuit Benchmarks

Table 3 shows the execution time required for a blinded RSA circuit of input size 128. For these tests we used a more powerful server with 64 cores and 1 Terabyte of memory. Our testbed is able to give dedicated CPUs when running 32 circuits in parallel. Each circuit would have 1 core for the generation and 1 core for the evaluation. As described in Section 6, larger testbeds capable of executing 128 or 256 cores in parallel would be able to provide similar results for executing the 256 circuits necessary for a 2^{-80} security parameter as they could evaluate the added circuits in parallel. The main difference in execution time would come from the multiple OTs from the mobile device to the outsourced proxy. The RSA circuit has been previously evaluated with KSS, but never from the standpoint of a mobile device.

We only report the outsourced execution results, as the circuits are far too large to evaluate directly on the phone. As with the larger circuits described in Section 6, the phone runs out of memory from merely trying to store a representation of the circuit. Prior to optimization, the blinded RSA circuit is 192,537,834 gates and afterward, comprises 116,083,727 gates, or 774 MB in size.

The implementation of Dijkstra’s shortest-path algorithm results in very large circuits. As shown in Table 3, the pre-optimized size of the shortest path circuit for 20 vertices is 20,288,444 gates and after optimization is 1,653,542 gates. The 100-node graph is even larger, with 168,422,382 gates post optimization, 1124 MB in size. This final example is among the largest evaluated

	32 Circuits Time (ms)	64 Circuits (ms)	128 Circuits (ms)	Optimized Gates	Unoptimized Gates	Size (MB)
RSA128	505000.0 ± 2%	734000.0 ± 4%	1420000.0 ± 1%	116,083,727	192,537,834	774
Dijkstra20	25800.0 ± 2%	49400.0 ± 1%	106000.0 ± 1%	1,653,542	20,288,444	11
Dijkstra50	135000.0 ± 1%	197000.0 ± 3%	389000.0 ± 2%	22,109,732	301,846,263	147
Dijkstra100	892000.0 ± 2%	1300000.0 ± 2%	2560000.0 ± 1%	168,422,382	2,376,377,302	1124

Table 3: Execution time for evaluating a 128-bit blinded RSA circuit and Dijkstra shortest path solvers over graphs with 20, 50, and 100 vertices. All numbers are for outsourced evaluation, as the circuits are too large to be computed without outsourcing to a proxy.

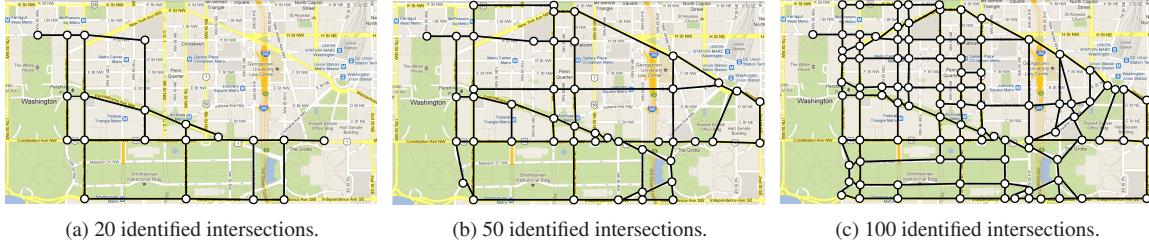


Figure 8: Map of potential presidential motorcade routes through Washington, DC. As the circuit size increases, a larger area can be represented at a finer granularity.

garbled circuits to date. While it may be possible for existing protocols to evaluate circuits of similar size, it is significant that we are evaluating comparably massive circuits from a resource-constrained mobile device.

7.2 Privacy-Preserving Navigation

Mapping and navigation are some of the most popular uses of a smartphone. Consider how directions may be given using a mobile device and an application such as Google Maps, without revealing the user’s current location, their ultimate destination, or the route that they are following. That is, the navigation server should remain oblivious of these details to ensure their mutual privacy and to prevent giving away potentially sensitive details if the phone is compromised. Specifically, consider planning of the motorcade route for the recent Presidential inauguration. In this case, the route is generally known in advance but is potentially subject to change if sudden threats emerge. A field agent along the route wants to receive directions without providing the navigation service any additional details, and without sensitive information about the route loaded to the phone. Moreover, because the threats may be classified, the navigation service does not want the holder of the phone to be given this information directly. In our example, the user of the phone is trying to determine the shortest path.

To model this scenario, we overlay a graph topology on a map of downtown Washington D.C., encoding intersections as vertices. Edge weights are a function of their distance and heuristics such as potential risks along a graph edge. Figure 8 shows graphs generated based on vertices of 20, 50, and 100 nodes, respectively. Note that the 100-node graph (Figure 8c) encompasses a larger area and provides finer-grained resolution of individual

intersections than the 20-node graph (Figure 8a).

There is a trade-off between detail and execution time, however; as shown in Table 3, a 20-vertex graph can be evaluated in under 26 seconds, while a 100-vertex graph requires almost 15 minutes with 32 circuits in our 64-core server testbed. The 64 circuit evaluation requires more time: almost 50 seconds for the 20-vertex graph, and almost 22 minutes for a 100-vertex graph. We anticipate that based on the role a particular agent might have on a route, they will be able to generate a route that covers their particular geographical jurisdiction and thus have an appropriately sized route, with only certain users requiring the highest-resolution output. Additionally, as described in Section 6.3, servers with more parallel cores can simultaneously evaluate more circuits, giving faster results for the 64 circuit evaluation.

Figure 9 reflects two routes. The first, overlaid with a dashed blue line, is the shortest path under optimal conditions that is output by our directions service, based on origin and destination points close to the historical start and end points of the past six presidential inaugural motorcades. Now consider that incidents have happened along the route, shown in the figure as a car icon in a hazard zone inside a red circle. The agent recalculates the optimal route, which has been updated by the navigation service to assign severe penalties to those corresponding graph edges. The updated route returned by the navigation service is shown in the figure as a path with a dotted purple line. In the 50-vertex graph in Figure 8, the updated directions would be available in just over 135 seconds for 32-circuit evaluation, and 196 and a half seconds for 64-circuit evaluation.

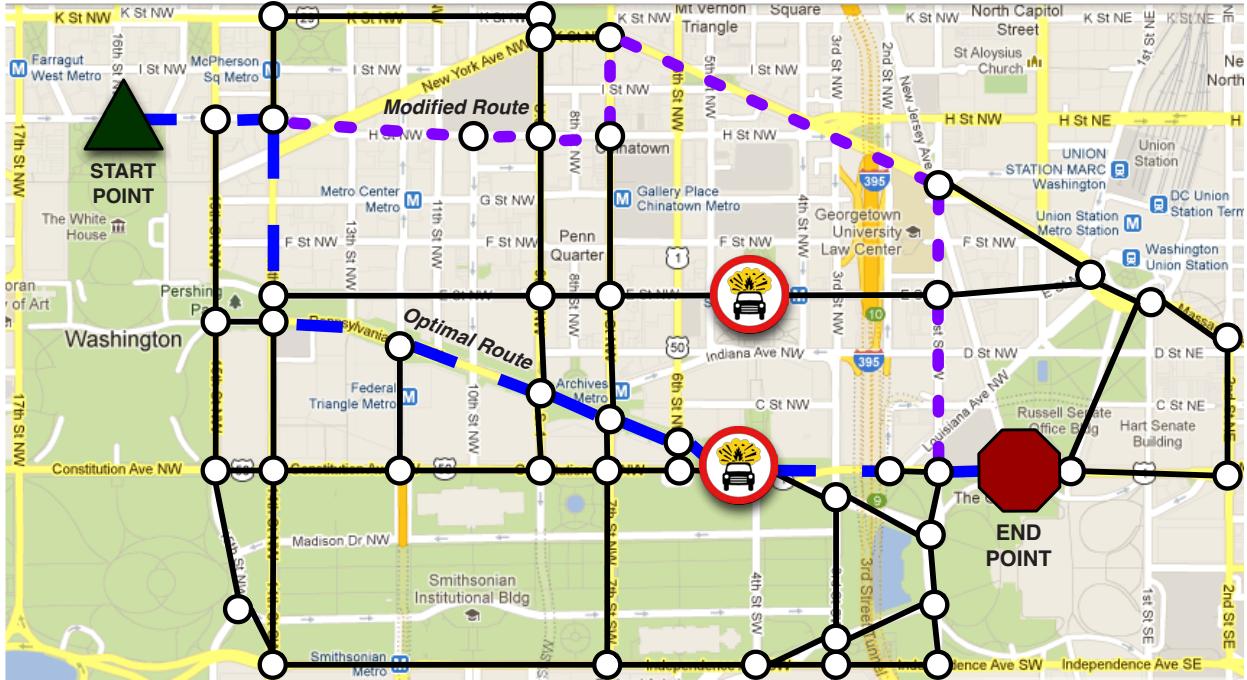


Figure 9: Motorcade route with hazards along the route. The dashed blue line represents the optimal route, while the dotted violet line represents the modified route that takes hazards into account.

8 Conclusion

While garbled circuits offer a powerful tool for secure function evaluation, they typically assume participants with massive computing resources. Our work solves this problem by presenting a protocol for outsourcing garbled circuit evaluation from a resource-constrained mobile device to a cloud provider in the malicious setting. By extending existing garbled circuit evaluation techniques, our protocol significantly reduces both computational and network overhead on the mobile device while still maintaining the necessary checks for malicious or lazy behavior from all parties. Our outsourced oblivious transfer construction significantly reduces the communication load on the mobile device and can easily accommodate more efficient OT primitives as they are developed. The performance evaluation of our protocol shows dramatic decreases in required computation and bandwidth. For the edit distance problem of size 128 with 32 circuits, computation is reduced by 98.92% and bandwidth overhead reduced by 99.95% compared to non-outsourced execution. These savings are illustrated in our privacy-preserving navigation application, which allows a mobile device to efficiently evaluate a massive garbled circuit securely through outsourcing. These results demonstrate that the recent improvements in garbled circuit efficiency can be applied in practical privacy-preserving mobile applications on even the most resource-constrained devices.

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. We would like to thank Benjamin Kreuter, abhi shelat, and Chih-hao Shen for working with us on their garbled circuit compiler and evaluation framework; Chris Peikert for providing helpful feedback on our proofs of security; Thomas DuBuisson and Galois for their assistance in the performance evaluation; and Ian Goldberg for his guidance during the shepherding process.

References

- [1] M. Bellare and S. Micali. Non-interactive oblivious transfer and applications. In *Advances in Cryptology–CRYPTO*, 1990.
- [2] M. Ben-Or, S. Goldwasser, and A. Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation. In *Proceedings of the annual ACM symposium on Theory of computing*, 1988.

- [3] 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.
- [4] R. Canetti, Y. Lindell, R. Ostrovsky, and A. Sahai. Universally composable two-party and multi-party secure computation. In *Proceedings of the annual ACM symposium on Theory of computing*, 2002.
- [5] H. Carter, C. Amrutkar, I. Dacosta, and P. Traynor. Efficient oblivious computation techniques for privacy-preserving mobile applications. *Journal of Security and Communication Networks (SCN)*, To appear 2013.
- [6] H. Carter, B. Mood, P. Traynor, and K. Butler. Secure outsourced garbled circuit evaluation for mobile devices. Technical Report GT-CS-12-09, College of Computing, Georgia Institute of Technology, 2012.
- [7] D. Chaum, C. Crépeau, and I. Damgård. Multiparty unconditionally secure protocols. In *Proceedings of the annual ACM symposium on Theory of computing*, 1988.
- [8] S. G. Choi, J. Katz, R. Kumaresan, and H.-S. Zhou. On the security of the "free-xor" technique. In *Proceedings of the international conference on Theory of Cryptography*, 2012.
- [9] I. Damgård and Y. Ishai. Scalable secure multiparty computation. In *Proceedings of the annual international conference on Advances in Cryptology*, 2006.
- [10] I. Damgård and J. B. Nielsen. Scalable and unconditionally secure multiparty computation. In *Proceedings of the annual international cryptology conference on Advances in cryptology*, 2007.
- [11] V. Goyal, P. Mohassel, and A. Smith. Efficient two party and multi party computation against covert adversaries. In *Proceedings of the theory and applications of cryptographic techniques annual international conference on Advances in cryptology*, 2008.
- [12] M. Green, S. Hohenberger, and B. Waters. Outsourcing the decryption of abe ciphertexts. In *Proceedings of the USENIX Security Symposium*, 2011.
- [13] Y. Huang, P. Chapman, and D. Evans. Privacy-Preserving Applications on Smartphones. In *Proceedings of the USENIX Workshop on Hot Topics in Security*, 2011.
- [14] Y. Huang, D. Evans, and J. Katz. Private set intersection: Are garbled circuits better than custom protocols? In *NDSS '12: Proceedings of the 19th ISOC Symposium on Network and Distributed Systems Security*, San Diego, CA, USA, Feb. 2012.
- [15] 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.
- [16] Y. Huang, J. Katz, and D. Evans. Quid-pro-quotocols: Strengthening semi-honest protocols with dual execution. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2012.
- [17] A. Iliev and S. W. Smith. Small, stupid, and scalable: Secure computing with faerieplay. In *The ACM Workshop on Scalable Trusted Computing*, 2010.
- [18] Y. Ishai, J. Kilian, K. Nissim, and E. Petrank. Extending oblivious transfers efficiently. In *Proceedings of the Annual International Cryptology Conference*, 2003.
- [19] S. Jha, L. Kruger, and V. Shmatikov. Towards practical privacy for genomic computation. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2008.
- [20] S. Kamara, P. Mohassel, and M. Raykova. Outsourcing multi-party computation. *Cryptology ePrint Archive*, Report 2011/272, 2011. <http://eprint.iacr.org/>.
- [21] 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.
- [22] M. S. Kiraz. *Secure and Fair Two-Party Computation*. PhD thesis, Technische Universiteit Eindhoven, 2008.
- [23] M. S. Kiraz and B. Schoenmakers. A protocol issue for the malicious case of yaos garbled circuit construction. In *Proceedings of Symposium on Information Theory in the Benelux*, 2006.
- [24] V. Kolesnikov and T. Schneider. Improved garbled circuit: Free xor gates and applications. In *Proceedings of the international colloquium on Automata, Languages and Programming, Part II*, 2008.
- [25] B. Kreuter, a. shelat, and C. Shen. Billion-gate secure computation with malicious adversaries. In

- Proceedings of the USENIX Security Symposium*, 2012.
- [26] 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.
 - [27] Y. Lindell. Lower bounds and impossibility results for concurrent self composition. *Journal of Cryptology*, 21(2):200–249, 2008.
 - [28] Y. Lindell and B. Pinkas. Privacy preserving data mining. In *Proceedings of the Annual International Cryptology Conference on Advances in Cryptology*, 2000.
 - [29] 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.
 - [30] 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.
 - [31] L. Malka. Vmcrypt: modular software architecture for scalable secure computation. In *Proceedings of the 18th ACM conference on Computer and communications security*, 2011.
 - [32] D. Malkhi, N. Nisan, B. Pinkas, and Y. Sella. Fairplay—a secure two-party computation system. In *Proceedings of the USENIX Security Symposium*, 2004.
 - [33] Message Passing Interface Forum. The message passing interface (mpi) standard. <http://www.mcs.anl.gov/research/projects/mpi/>, 2009.
 - [34] P. Mohassel and M. Franklin. Efficiency tradeoffs for malicious two-party computation. In *Proceedings of the Public Key Cryptography conference*, 2006.
 - [35] 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.
 - [36] M. Naor and B. Pinkas. Efficient oblivious transfer protocols. In *Proceedings of the annual ACM-SIAM symposium on Discrete algorithms*, 2001.
 - [37] M. Naor, B. Pinkas, and R. Sumner. Privacy preserving auctions and mechanism design. In *Proceedings of the ACM conference on Electronic commerce*, 1999.
 - [38] 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.
 - [39] C. Peikert, V. Vaikuntanathan, and B. Waters. A framework for efficient and composable oblivious transfer. In *Advances in Cryptology (CRYPTO)*, 2008.
 - [40] W. Rash. Dropbox password breach highlights cloud security weaknesses. <http://www.eweek.com/c/a/Security/Dropbox-Password-Breach-Highlights-Cloud-Security-Weaknesses-266215/>, 2012.
 - [41] 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.
 - [42] K. Thomas. Microsoft cloud data breach heralds things to come. http://www.pcworld.com/article/214775/microsoft_cloud_data_breach_sign_of_future.html, 2010.
 - [43] A. C. Yao. Protocols for secure computations. In *Proceedings of the Annual Symposium on Foundations of Computer Science*, 1982.

On the Security of RC4 in TLS¹

Nadhem J. AlFardan

Information Security Group,

Royal Holloway, University of London

Daniel J. Bernstein

University of Illinois at Chicago and

Technische Universiteit Eindhoven

Kenneth G. Paterson

Information Security Group,

Royal Holloway, University of London

Bertram Poettering

Information Security Group,

Royal Holloway, University of London

Jacob C. N. Schuldt

Information Security Group,

Royal Holloway, University of London

Abstract

The Transport Layer Security (TLS) protocol aims to provide confidentiality and integrity of data in transit across untrusted networks. TLS has become the *de facto* protocol standard for secured Internet and mobile applications. TLS supports several symmetric encryption options, including a scheme based on the RC4 stream cipher. In this paper, we present ciphertext-only plaintext recovery attacks against TLS when RC4 is selected for encryption. Our attacks build on recent advances in the statistical analysis of RC4, and on new findings announced in this paper. Our results are supported by an experimental evaluation of the feasibility of the attacks. We also discuss countermeasures.

1 Introduction

TLS is arguably the most widely used secure communications protocol on the Internet today. Starting life as SSL, the protocol was adopted by the IETF and specified as an RFC standard under the name of TLS 1.0 [7]. It has since evolved through TLS 1.1 [8] to the current version TLS 1.2 [9]. Various other RFCs define additional TLS cryptographic algorithms and extensions. TLS is now used for securing a wide variety of application-level traffic: It serves, for example, as the basis of the HTTPS protocol for encrypted web browsing, it is used in conjunction with IMAP or SMTP to cryptographically protect email traffic, and it is a popular tool to secure communication with embedded systems, mobile devices, and in payment systems.

Technically speaking, TLS sessions consist of two consecutive phases: the execution of the TLS Handshake Protocol which typically deploys asymmetric techniques to establish a secure session key, followed by the execution of the TLS Record Protocol which uses symmetric key cryptography (block ciphers, the RC4 stream cipher, MAC algorithms) in combination with the established session key and sequence numbers to build a se-

cure channel for transporting application-layer data. In the Record Protocol, there are mainly three encryption options:

- HMAC followed by CBC-mode encryption using a block cipher,
- HMAC followed by encryption using the RC4 stream cipher, or
- authenticated encryption using GCM or CCM mode of operation of a block cipher.

The third of these three options is only available with TLS 1.2 [21, 18], which is yet to see widespread adoption.² The first option has seen significant cryptanalysis (padding oracle attacks [6], BEAST [10], Lucky 13 [3]). While countermeasures to the attacks on CBC-mode in TLS exist, many commentators now recommend, and many servers now offer, RC4-based encryption options ahead of CBC-mode.³ Indeed, the ICSI Certificate Notary⁴ recently performed an analysis of 16 billion TLS connections and found that around 50% of the traffic was protected using RC4 ciphersuites [5].

This makes it timely to examine the security of RC4 in TLS. While the RC4 algorithm is known to have a variety of cryptographic weaknesses (see [23] for an excellent survey), it has not been previously explored how these weaknesses can be exploited in the context of TLS. Here we show that new and recently discovered biases in the RC4 keystream do create serious vulnerabilities in TLS when using RC4 as its encryption algorithm.

While the main focus of this paper lies on the security of RC4 in TLS, our attacks (or variants thereof) might also be applicable to other protocols where RC4 is meant to ensure data confidentiality. Indeed, the WPA protocol used for encrypting wireless network traffic also utilizes the RC4 stream cipher in a way that allows (partial) plaintext recovery in specific settings — using basically the same attack strategies as in the TLS case.

We hope that this work will help spur the adoption of TLS 1.2 and its authenticated encryption algorithms, as well as the transition from WPA to (the hopefully more secure) WPA2.

1.1 Overview of Results

We present two plaintext recovery attacks on RC4 that are exploitable in specific but realistic circumstances when this cipher is used for encryption in TLS. Both attacks require a fixed plaintext to be RC4-encrypted and transmitted many times in succession (in the same, or in multiple independent RC4 keystreams). Interesting candidates for such plaintexts include passwords and, in the setting of secure web browsing, HTTP cookies.

A statistical analysis of ciphertexts forms the core of our attacks. We stress that the attacks are ciphertext-only: no sophisticated timing measurement is needed on the part of the adversary, the attacker does not need to be located close to the server, and no packet injection capability is required (all premises for Lucky 13). Instead, it suffices for the adversary to record encrypted traffic for later offline analysis. Provoking the required repeated encryption and transmission of the target plaintext, however, might require more explicit action: e.g., resetting TCP connections or guiding the victim to a website with specially prepared JavaScript (see examples below).

Since both our attacks require large amounts of ciphertext, their practical relevance could be questioned. However, they do show that the strength of RC4 in TLS is much lower than the employed 128-bit key would suggest. We freely admit that our attacks are not particularly deep, nor sophisticated: they only require an understanding of how TLS uses RC4, solid statistics on the biases in RC4 keystreams, and some experience of how modern browsers handle cookies. We consider it both surprising and alarming that such simple attacks are possible for such an important and heavily-studied protocol as TLS. We further discuss the implications of our attack in Section 6 and in the full version of this paper [4].

1.1.1 Our single-byte bias attack

Our first attack targets the initial 256 bytes of RC4 ciphertext. It is fixed-plaintext and multi-session, meaning that it requires a fixed sequence of plaintext bytes to be independently encrypted under a large number of (random) keys. This setting corresponds to what is called a “broadcast attack” in [17, 15, 23]. As we argue below, such attacks are a realistic attack vector in TLS. Observe that, in TLS, the first 36 bytes of the RC4 keystream are used to encrypt a TLS Handshake Finished message. This message is not fixed across TLS sessions. As a consequence, our methods can be applied only to recover up

to 220 bytes of the TLS application plaintext.

Our attack exploits statistical biases occurring in the first 256 bytes of RC4 keystream. Such biases, i.e., deviations from uniform in the distributions of the keystream bytes at certain positions, have been reported and theoretically analyzed by [17], [15], and [23]. The corresponding authors also propose algorithms to exploit such biases for plaintext recovery. In this paper, we discuss shortcomings of their algorithms, empirically obtain a *complete view* of all single-byte biases occurring in the first 256 keystream positions, and propose a generalized algorithm that fully exploits all these biases for advanced plaintext recovery. As a side result of our research, in Section 3.1 we report on significant biases in the RC4 keystream that seemingly follow specific patterns and that have not been identified or analysed previously.

For concreteness, we describe how our single-byte bias attack could be applied to recover cookies in HTTPS traffic. Crucial here is to find an automated mechanism for efficiently generating a large number of encryptions of the target cookie. In line with the scenario employed by the BEAST and Lucky 13 attacks against CBC-mode encryption in TLS [3, 10], a candidate mechanism is for JavaScript malware downloaded from an attacker-controlled website and running in the victim’s browser to repeatedly send HTTPS requests to a remote server. The corresponding cookies are automatically included in each of these requests in a predictable location, and can thus be targeted in our attack. If client and server are configured to use TLS session resumption, the renewal of RC4 keys could be arranged to happen with particularly high frequency — as required for our attack to be successful.⁵ Alternatively, the attacker can cause the TLS session to be terminated after the target encrypted cookie is sent; the browser will automatically establish a new TLS session when the next HTTPS request is sent.

As a second example, consider the case where IMAP passwords⁶ are attacked. In a setup where an email client regularly connects to an IMAP server for (password-authenticated) mail retrieval, let the adversary reset the TCP connection between client and server immediately after the encrypted password is transmitted. In some client configurations this might trigger an automatic resumption of the session, including a retransmission of the (encrypted) password. If this is the case, the adversary is in the position to harvest a large set of independently encrypted copies of the password —one per reset— precisely fulfilling the precondition of our attack.

Our single-byte bias attack is on the verge of practicality. In our experiments, the first 40 bytes of TLS application data after the Finished message were recovered with a success rate of over 50% per byte, using 2^{26} sessions. With 2^{32} sessions, the per-byte success rate is more than 96% for the first 220 bytes (and is 100%

for all but 12 of these bytes). If, for example, a target plaintext byte is known to be a character from a set of cardinality 16 (e.g., in a 4-bits-per-byte-encoded HTTP cookie), our algorithm recovers the first 112 bytes of plaintext with a success rate of more than 50% per byte, using 2^{26} sessions. For further details, see Section 5.

1.1.2 Our double-byte bias attack

As we have seen, our single-byte bias attack on RC4 is quite effective in recovering ‘early’ plaintext bytes in the fixed-plaintext multi-session setting. It has, however, a couple of limitations when it comes to attacking practical systems that employ TLS. Focussing on the recovery of cookies in HTTPS-secured web sessions, we note that modern web browsers typically send a large number of HTTP headers before any cookies (these headers carry information about the particular client or server software, accepted MIME types, compression options, etc.). In practice, cookie data appears only at positions that come after the attackable initial 220 bytes of the ciphertext⁷. Independently of this issue, in the attack scenarios proposed above, a large number of HTTPS sessions would have to be established and torn down again, inducing non-negligible computing and bandwidth overheads via the TLS Handshake. Lastly, it has been proposed to routinely drop the first few hundred keystream bytes of RC4 before starting encryption in order to avoid the relatively strong early keystream biases [19] — if this were to be implemented in TLS, our single-byte bias attack would effectively be defeated.

Complementary to our single-byte bias attack, we present a second fixed-plaintext ciphertext-only attack on RC4. It exploits biases that appear in the entire keystream (and not just in the first 256 positions) and does not assume, but tolerates, frequent changes of the encryption key. Our second attack hence covers some scenarios where our single-byte bias attack does not seem to be applicable; it would, for example, be able to recover cookies from (long-persisting) HTTPS sessions. It would also be applicable if the initial keystream bytes were to be discarded.

In contrast to our first attack, our second attack exploits certain biases in consecutive pairs of bytes in the RC4 keystream that were first reported by Fluhrer and McGrew [12]. We empirically evaluate the probability of occurrence for each possible pair of bytes beginning at each position (modulo 256), obtaining a complete view of the distributions of pairs of bytes in positions $(i, i + 1)$ (modulo 256). Our analysis strongly suggests that there are no further biases in consecutive positions of the same strength as the Fluhrer-McGrew biases. We use the obtained results in a specially designed attack algorithm to recover repeatedly encrypted plaintexts.

Our double-byte bias attack is again close to being practical. In our experiments, we focus on our attack’s ability to correctly recover 16 consecutive bytes of plaintext, roughly equating to an HTTP cookie. With $13 \cdot 2^{30}$ encryptions of the plaintext, we achieve a success rate of 100% in recovering all 16 bytes. We obtain better success rates for restricted plaintexts, as in the single-byte case. For further details, see Section 5.

1.2 Related Work

In independent and concurrent work, Isobe *et al.* [13] have considered the security of RC4 against broadcast attacks. They present attacks based on both single-byte and multi-byte biases. They identify three biases in the first output bytes Z_r of RC4 that we also identify (specifically, the biases towards $Z_3 = 0x83$, $Z_r = r$, and $Z_r = -r$ when r is a multiple of 16) as well as a new conditional bias $Z_1 = 0 | Z_2 = 0$.

The single-byte bias attack in [13] only considers the *strongest* bias at each position, whereas our single-byte bias attack simultaneously exploits *all* biases in each keystream position. Specifically, we use Bayes’s law to compute the *a posteriori* plaintext distribution from the *a priori* plaintext distribution and the precomputed distributions of the Z_r . This explains why our single-byte attack out-performs that of [13]. For example, we achieve reliable plaintext recovery in the first 256 positions with 2^{32} ciphertexts, while Isobe *et al.* [13] require 2^{34} ciphertexts. We also achieve uniformly higher success rates for lower numbers of sessions. Previous authors exploring broadcast attacks on RC4 also only used single biases, leading to attacks that simply do not work [15, 23] or which have inferior performance to ours [22].

The multi-byte bias attack in [13] exploits the positive bias towards the pattern *ABSAB* that was identified by Mantin [16]. Here *A* and *B* are keystream bytes and \mathcal{S} is a short string consisting of any keystream bytes (possibly of length 0). The attack in [13] assumes that 3-out-of-4 bytes in particular positions are known and uses the Mantin bias to recover the fourth. A limited experimental evaluation of the attack is reported in [13]: the attack is applied only to recovery of plaintext bytes 258–261, assuming all previous plaintext bytes have been successfully recovered, with success rates of 1 (for each of the 4 targeted bytes) using 2^{34} ciphertexts. As explained in [13], this multi-byte attack would fail if the initial bytes of RC4 output were to be discarded. By contrast, our double-byte bias attack, which exploits the Fluhrer-McGrew biases, recovers more bytes with comparable success rate using slightly fewer ciphertexts and is resilient to initial byte discarding. It is an interesting open problem to determine whether the Mantin *ABSAB* bias can be combined with the Fluhrer-McGrew biases to

gain enhanced attack performance.

A further point of comparison between our work and that of [13] concerns practical implementation. We have extensively explored the applicability of our attacks to RC4 as used in TLS, while [13] makes only brief mention of TLS in its concluding section and gives no mechanisms for generating the large numbers of ciphertexts needed for the attacks.

Finally, the authors of [13] claim in their abstract that their methods “can recover the first 2^{50} bytes $\approx 1000 T$ bytes of the plaintext, with probability close to 1, from only 2^{34} ciphertexts”. We point out that their methods would only recover 2^{16} distinct bytes of output, rather than the advertised 2^{50} bytes, since their attacks require the same plaintext to be encrypted 2^{34} times. Furthermore, their multi-byte bias attack is not resilient to errors occurring in the recovery of early plaintext bytes (whereas ours is), so this claim would only be true if their multi-byte bias attack does not fail at any stage, and this is as yet untested.

1.3 Paper Organisation

Section 2 provides further background on the RC4 stream cipher and the TLS Record Protocol. Section 3 summarises weaknesses in RC4 that we exploit in our attacks. Section 4 describes our two plaintext recovery attacks on RC4. We evaluate the attacks in Section 5, with our main focus there being on TLS. Finally, Section 6 discusses countermeasures to our attacks, and concludes with a recap of the main issues raised by our work.

2 Further Background

2.1 The RC4 Stream Cipher

The stream cipher RC4, originally designed by Ron Rivest, became public in 1994 and found application in a wide variety of cryptosystems; well-known examples include SSL/TLS, WEP [1], WPA [2], and some Kerberos-related encryption modes [14]. RC4 has a remarkably short description and is extremely fast when implemented in software. However, these advantages come at the price of lowered security: several weaknesses have been identified in RC4 [12, 11, 17, 16, 15, 23, 25, 24, 26], some of them being confirmed and exploited in the current paper.

Technically, RC4 consists of two algorithms: a *key scheduling algorithm* (KSA) and a *pseudo-random generation algorithm* (PRGA), which are specified in Figure 1. The KSA takes as input a key K , typically a byte-array of length between 5 and 32 (i.e., 40 to 256 bits), and produces the initial internal state $st_0 = (i, j, \mathcal{S})$, where \mathcal{S} is the canonical representation of a permutation on the set

Algorithm 1: RC4 key scheduling (KSA)

```

input : key  $K$  of  $l$  bytes
output: internal state  $st_0$ 
begin
    for  $i = 0$  to 255 do
         $\mathcal{S}[i] \leftarrow i$ 
     $j \leftarrow 0$ 
    for  $i = 0$  to 255 do
         $j \leftarrow$ 
         $j + \mathcal{S}[i] + K[i \bmod l]$ 
        swap( $\mathcal{S}[i], \mathcal{S}[j]$ )
     $i, j \leftarrow 0$ 
     $st_0 \leftarrow (i, j, \mathcal{S})$ 
return  $st_0$ 
```

Algorithm 2: RC4 keystream generator (PRGA)

```

input : internal state  $st_r$ 
output: keystream byte  $Z_{r+1}$ 
internal state  $st_{r+1}$ 
begin
    parse  $(i, j, \mathcal{S}) \leftarrow st_r$ 
     $i \leftarrow i + 1$ 
     $j \leftarrow j + \mathcal{S}[i]$ 
    swap( $\mathcal{S}[i], \mathcal{S}[j]$ )
     $Z_{r+1} \leftarrow \mathcal{S}[i] + \mathcal{S}[j]$ 
     $st_{r+1} \leftarrow (i, j, \mathcal{S})$ 
return  $(Z_{r+1}, st_{r+1})$ 
```

Figure 1: Algorithms implementing the RC4 stream cipher. All additions are performed modulo 256.

$[0, 255]$ as an array of bytes, and i, j are indices into this array. The PRGA will, given an internal state st_r , output ‘the next’ keystream byte Z_{r+1} , together with the updated internal state st_{r+1} . Particularly interesting to note is the fact that updated index j is computed in dependence on current i , j , and \mathcal{S} , while i is just a counter (modulo 256).

2.2 The TLS Record Protocol

We describe in detail the cryptographic operation of the TLS Record Protocol in the case that RC4 is selected as the encryption method.

Data to be protected by TLS is received from the application and may be fragmented and compressed before further processing. An individual record R (viewed as a sequence of bytes) is then processed as follows. The sender maintains an 8-byte sequence number SQN which is incremented for each record sent, and forms a 5-byte field HDR consisting of a 2-byte version field, a 1-byte type field, and a 2-byte length field. It then calculates an HMAC over the string $\text{HDR}||\text{SQN}||R$; let T denote the resulting tag.

For RC4 encryption, record and tag are concatenated to create the plaintext $P = R||T$. This plaintext is then xored in a byte-by-byte fashion using the RC4 keystream, i.e., the ciphertext bytes are computed as

$$C_r = P_r \oplus Z_r \quad \text{for } r = 1, 2, 3, \dots,$$

where P_r are the individual bytes of P , and Z_r are the RC4 keystream bytes. The data transmitted over the wire then has the form

$$\text{HDR}||C,$$

where C is the concatenation of the bytes C_r .

The RC4 algorithm itself is initialised at the start of each TLS connection, using a 128 bit encryption key K . This key K is computed with a hash-function-based key

derivation function from the TLS master secret that is established during the TLS Handshake Protocol. In more detail, the key K may be established either via a full TLS Handshake or via TLS session resumption. In a full TLS Handshake, a total of 4 communication round-trips are needed, and usually some public key cryptographic operations are required of both client and server. A full TLS Handshake run establishes a new TLS *session* and a new TLS master secret from which all other keys, including RC4 key K , are derived. TLS session resumption involves a lightweight version of the TLS Handshake Protocol being run to establish a new *connection* within an existing session: essentially, an exchange of nonces takes place, followed by an exchange of *Finished* messages; no public key cryptographic operations are involved. The keys for the new connection, including K , are derived from the existing master secret and the new nonces. Given the design of the key derivation process, it is reasonable to model K as being uniformly random in the different sessions/connections.

The initialisation of RC4 in TLS is the standard one for this algorithm. Notably, none of the initial keystream bytes is discarded when RC4 is used in TLS, despite these bytes having known weaknesses. Note also that the first record sent under the protection of RC4 for each session or connection will be a *Finished* message, typically of length 36 bytes, consisting of a Handshake Protocol header, a PRF output, and a MAC on that output. This is typically 36 bytes in size. This record will not be targeted in our attacks, since it is not constant across multiple sessions.

The decryption process reverses this sequence of steps, but its details are not germane to our attacks. For TLS, any error arising during decryption should be treated as fatal, meaning an (encrypted) error message is sent to the sender and the session terminated with all keys and other cryptographic material being disposed of. This gives an attacker a convenient method to cause a session to be terminated and force new encryption and MAC keys to be set up. Another method is to somehow induce the client or server to initiate session resumption.

3 Biases in the RC4 Keystream

In this section, we summarise known biases in the RC4 keystream, and report new biases that we have observed experimentally.

3.1 Single-byte Biases

The first significant bias in the RC4 keystream was observed by Mantin and Shamir in [17]. Their main result can be stated as:

Result 1. [17, Thm 1] *The probability that Z_2 , the second byte of keystream output by RC4, is equal to 0x00 is approximately 1/128 (where the probability is taken over the random choice of the key).*

Since this result concerns only the second byte of the keystream, and this byte is always used to encrypt a *Finished* message in TLS, we are unable to exploit it in our attacks. More recently, the following result was obtained by Sen Gupta *et al.* in [23] as a refinement of an earlier result of Maitra *et al.* [15]:

Result 2. [23, Thm 14 and Cor 3] *For $3 \leq r \leq 255$, the probability that Z_r , the r -th byte of keystream output by RC4, is equal to 0x00 is*

$$\Pr(Z_r = 0x00) = \frac{1}{256} + \frac{c_r}{256^2},$$

where the probability is taken over the random choice of the key, $c_3 = 0.351089$, and c_4, c_5, \dots, c_{255} is a decreasing sequence with terms that are bounded as follows:

$$0.242811 \leq c_r \leq 1.337057.$$

In other words, bytes 3 to 255 of the keystream have a bias towards 0x00 of approximately $1/2^{16}$. This result was experimentally verified in [23] and found to be highly accurate (see Figure 11 of that paper). The biases here are substantially smaller than those observed in Result 1.

Additionally, Sen Gupta *et al.* [23] have identified a key-length-dependent bias in RC4 keystreams. Specifically, [23, Theorem 5] shows that when the key-length is ℓ bytes, then byte Z_ℓ is biased towards value $256 - \ell$, with the bias always being greater than $1/2^{16}$. For RC4 in TLS, we have $\ell = 16$.

Experimentally, we have observed additional biases in the RC4 keystream that do not yet have a theoretical explanation. As an example, Figure 2 shows the empirical distribution for the RC4 keystream bytes Z_{16} , Z_{32} and Z_{50} , calculated over 2^{44} independent, random 128-bit keys. For Z_{16} , we have 3 main biases: the bias towards 0x00, the very dominant key-length-dependent bias towards 0xF0 (decimal 240) from [23], and a new bias towards 0x10 (decimal 16). For Z_{32} , we also have 3 main biases: the bias towards 0x00, a large, new bias towards 0xE0 (decimal 224), and a new bias towards 0x20 (decimal 32). For Z_{50} , there are significant biases towards byte values 0x00 and 0x32 (decimal 50), as well as an upward trend in probability as the byte value increases.

Individual inspection of ciphertext distributions at all positions $1 \leq r \leq 256$ reveals two new significant biases that occur with specific regularities: a bias towards value r for all r , and a bias towards value $256 - r$ at positions r that are *multiples* of (key-length) 16; note that

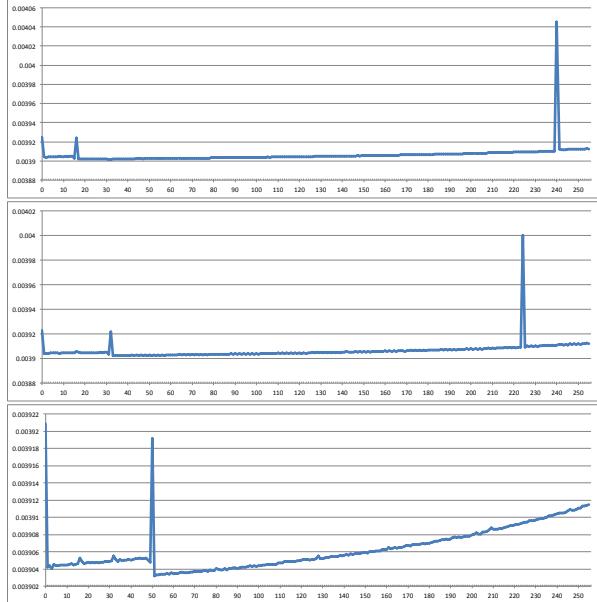


Figure 2: Measured distributions of RC4 keystream bytes Z_{16} (top), Z_{32} (middle), and Z_{50} (bottom).

the latter finding both confirms and extends the results from [23]. Both of these new biases were also observed by Isobe *et al.* [13], with a theoretical explanation being given for the bias towards r . Figure 3 shows the estimated strength of these biases in comparison with the strength of the bias towards 0x00 for the keystream bytes Z_1, \dots, Z_{256} . The estimates are based on the empirical distribution of the RC4 keystream bytes, calculated over 2^{44} random 128-bit RC4 keys. We note that the key-length dependent bias dominates the other two biases until position Z_{112} , and that the bias of Z_r towards r dominates the bias towards 0x00 observed by [15] between positions Z_5 and Z_{31} , except for byte Z_{16} where the bias towards 0x00 is slightly stronger.

Furthermore, for the first keystream byte Z_1 , we have observed a bias *away* from value 0x81 (decimal 129) in the addition to the known bias away from value 0x00. This additional bias is not consistent with the recent results of Sen Gupta *et al.* [23] who provide a theoretical treatment of the distribution of Z_1 . The disparity likely arises because Sen Gupta *et al.* work with 256-byte keys, while our work is exclusively concerned with 128-bit (16-byte) keys as used in TLS; in other words, our observed bias in $Z_1 = 0x81$ seems to be key-length-dependent. Finally, our computations have revealed a number of other, smaller biases in the initial bytes of the RC4 keystream.

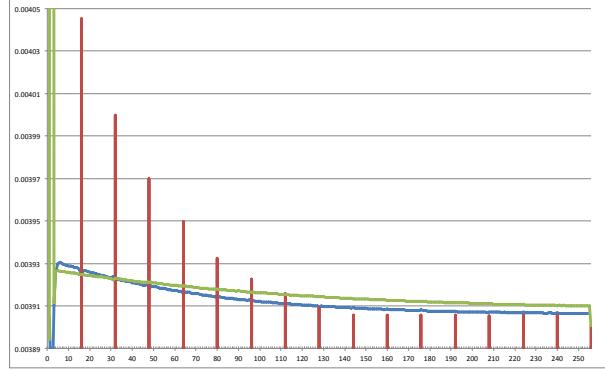


Figure 3: Measured strength of the bias towards 0x00 (green), the bias towards value r in Z_r (blue), and the key-length dependent bias towards byte value $256 - r$ (red) for keystream bytes Z_1, \dots, Z_{256} , based on keystreams generated by 2^{44} independent random keys. Note that the large peak for the 0x00 bias in Z_2 extends beyond the bounds of the graph and is not fully shown for illustrative purposes.

3.2 Multi-byte Biases

Besides the single-byte biases highlighted above, several multi-byte biases have been identified in the RC4 keystream. In contrast to the single-byte biases, most of the identified multi-byte biases are “long term” biases which appear periodically at regular intervals in the keystream.

The most extensive set of multi-byte biases was identified by Fluhrer and McGrew [12] who analyzed the distribution of pairs of byte values for consecutive keystream positions (Z_r, Z_{r+1}) , $r \geq 1$. More precisely, they estimated the distribution of consecutive keystream bytes for scaled-down⁸ versions of RC4 by assuming an idealized internal state of RC4 in which the permutation \mathcal{S} and the internal variable j are random (see Figure 1), and then extrapolated the results to standard RC4.

The reported biases for standard RC4 are listed in Table 1. Note that all biases are dependent on the internal variable i which is incremented (modulo 256) for each keystream byte generated. It should also be noted that, due to the assumption that \mathcal{S} and j are random, the biases cannot be expected to hold for the initial keystream bytes. However, this idealization becomes a close approximation to the internal state of RC4 after a few invocations of the RC4 keystream generator, [12].

We experimentally verified the Fluhrer-McGrew biases by analysing the output of 2^{10} RC4 instances using 128-bit keys and generating 2^{40} keystream bytes each. For each keystream, the initial 1024 bytes were dropped. Based on this data, we found the biases from [12] to be accurate, also for 128-bit keys. This is in-line with the

experiments and observations reported in [12]. Furthermore, we did not identify any additional significant long term biases for consecutive keystream bytes which are repeated with a periodicity that is a proper divisor of 256. Hence, for the purpose of implementing the attack presented in Section 4.2, we assume that the biases identified in [12] are the only existing long term biases for consecutive keystream bytes, and that all other pairs of byte-values are uniformly distributed.

Byte pair	Condition on i	Probability
(0,0)	$i = 1$	$2^{-16}(1+2^{-9})$
(0,0)	$i \neq 1, 255$	$2^{-16}(1+2^{-8})$
(0,1)	$i \neq 0, 1$	$2^{-16}(1+2^{-8})$
($i+1, 255$)	$i \neq 254$	$2^{-16}(1+2^{-8})$
(255, $i+1$)	$i \neq 1, 254$	$2^{-16}(1+2^{-8})$
(255, $i+2$)	$i \neq 0, 253, 254, 255$	$2^{-16}(1+2^{-8})$
(255,0)	$i = 254$	$2^{-16}(1+2^{-8})$
(255,1)	$i = 255$	$2^{-16}(1+2^{-8})$
(255,2)	$i = 0, 1$	$2^{-16}(1+2^{-8})$
(129,129)	$i = 2$	$2^{-16}(1+2^{-8})$
(255,255)	$i \neq 254$	$2^{-16}(1-2^{-8})$
(0, $i+1$)	$i \neq 0, 255$	$2^{-16}(1-2^{-8})$

Table 1: Fluhrer-McGrew biases for consecutive pairs of byte values. In the table, i is the internal variable of the RC4 keystream generation algorithm (see Section 2.1).

Independently of [12], Mantin [16] identified a positive bias towards the pattern $ABSAB$, where A and B represent byte values and S is a short string of bytes (possibly of length 0). The shorter the string S is, the more significant is the bias. Additionally, Sen Gupta *et al.* [23] identified a bias towards the byte values (0,0) for keystream positions (Z_r, Z_{r+2}) , separated by any single keystream byte for $r \geq 1$. However, we do not make use of these biases in the attacks presented in this paper.

4 Plaintext Recovery Attacks

For the purpose of exposition, we first explain how the broadcast attack by Maitra *et al.* [15] and Sen Gupta *et al.* [23] is meant to work. Suppose byte Z_r of the RC4 keystream has a dominant bias towards value 0x00. As RC4 encryption is defined as $C_r = P_r \oplus Z_r$, the corresponding ciphertext byte C_r has a bias towards plaintext byte P_r . Thus, obtaining sufficiently many ciphertext samples C_r for a fixed plaintext P_r allows inference of P_r by a majority vote: P_r is equal to the value of C_r that occurs most often. This is the core idea of Algorithm 3 that we reproduce from [15, 23]. Let S denote the number of ciphertexts available to the attacker and, for all $1 \leq j \leq S$, let $C_{j,r}$ denote the r -th byte of ciphertext C_j . For a fixed position r , Algorithm 3 runs through all j , and in each iteration increments one out of 256 counters,

Algorithm 3: Basic plaintext recovery attack

```

input :  $S$  independent encryptions  $(C_j)_{1 \leq j \leq S}$  of
       fixed plaintext  $P$ , position  $r$ 
output: estimate  $P_r^*$  for plaintext byte  $P_r$ 
begin
   $N_{0x00} \leftarrow 0, \dots, N_{0xFF} \leftarrow 0$ 
  for  $j = 1$  to  $S$  do
     $\quad\quad\quad\lfloor N_{C_{j,r}} \leftarrow N_{C_{j,r}} + 1$ 
     $P_r^* \leftarrow \arg \max_{\mu \in \{0x00, \dots, 0xFF\}} N_\mu$ 

```

namely the one that corresponds to value $C_{j,r}$. After processing all ciphertexts, the character corresponding to the largest counter in the obtained histogram is the output of the algorithm.

The algorithm is tailor-made for plaintext recovery in the case described by Result 2: it assumes that the largest bias in the RC4 keystream is towards 0x00. However, it is highly likely to fail to reliably suggest the correct plaintext byte P_r if the RC4 keystream has, in position r , additional biases of approximately the same size (or larger) as the bias towards 0x00. Such additional biases would simply be misinterpreted as the bias towards 0x00 and hence falsify the result. As we observed in Section 3.1 (and Figure 3), several other quite strong biases in the RC4 keystream do indeed exist. This clearly invalidates Algorithm 3 for practical use.

4.1 Our Single-byte Bias Attack

We propose a plaintext-recovery algorithm that takes into account *all* possible single-byte RC4 biases at the same time, along with their strengths. The idea is to first obtain a detailed picture of the distributions of RC4 keystream bytes Z_r , for all positions r , by gathering statistics from keystreams generated using a large number of independent keys (2^{44} in our case). That is, for all r , we (empirically) estimate

$$p_{r,k} := \Pr(Z_r = k), \quad k = 0x00, \dots, 0xFF ,$$

where the probability is taken over the random choice of the RC4 encryption key (i.e., 128 bit keys in the TLS case). Using these biases $p_{r,k}$, in a second step, plaintext can be recovered with optimal accuracy using a maximum-likelihood approach, as follows.

Suppose we have S ciphertexts C_1, \dots, C_S available for our attack. For any fixed position r and any candidate plaintext byte μ for that position, vector $(N_{0x00}^{(\mu)}, \dots, N_{0xFF}^{(\mu)})$ with

$$N_k^{(\mu)} = |\{j \mid C_{j,r} = k \oplus \mu\}_{1 \leq j \leq S}| \quad (0x00 \leq k \leq 0xFF)$$

represents the distribution on Z_r required to obtain the observed ciphertexts $\{C_{j,r}\}_{1 \leq j \leq S}$ by encrypting μ . We

Algorithm 4: Single-byte bias attack

input : S independent encryptions $\{C_j\}_{1 \leq j \leq S}$ of fixed plaintext P , position r , keystream distribution $(p_{r,k})_{0x00 \leq k \leq 0xFF}$ at position r

output: estimate P_r^* for plaintext byte P_r

begin

```

 $N_{0x00} \leftarrow 0, \dots, N_{0xFF} \leftarrow 0$ 
for  $j = 1$  to  $S$  do
     $N_{C_{j,r}} \leftarrow N_{C_{j,r}} + 1$ 
    for  $\mu = 0x00$  to  $0xFF$  do
        for  $k = 0x00$  to  $0xFF$  do
             $N_k^{(\mu)} \leftarrow N_{k \oplus \mu}$ 
             $\lambda_\mu \leftarrow \sum_{k=0x00}^{0xFF} N_k^{(\mu)} \log p_{r,k}$ 
     $P_r^* \leftarrow \arg \max_{\mu \in \{0x00, \dots, 0xFF\}} \lambda_\mu$ 
return  $P_r^*$ 

```

compare these *induced* distributions (one for each possible μ) with the accurate distribution $p_{r,0x00}, \dots, p_{r,0xFF}$ and interpret a close match as an indication for the corresponding plaintext candidate μ being the correct one, i.e., $P_r = \mu$. More formally, we observe that the probability λ_μ that plaintext byte μ is encrypted to ciphertext bytes $\{C_{j,r}\}_{1 \leq j \leq S}$ follows a multinomial distribution and can be precisely calculated as

$$\lambda_\mu = \frac{S!}{N_{0x00}^{(\mu)}! \cdots N_{0xFF}^{(\mu)}!} \prod_{k \in \{0x00, \dots, 0xFF\}} P_{r,k}^{N_k^{(\mu)}}. \quad (1)$$

By computing λ_μ for all $0x00 \leq \mu \leq 0xFF$ and identifying μ such that λ_μ is largest, we determine the (optimal) maximum-likelihood plaintext byte value. Algorithm 4 specifies the details of the described single-byte bias attack, including the optimizations discussed next.

Observe that, for each fixed position r and set of ciphertexts $\{C_{j,r}\}_{1 \leq j \leq S}$, values $N_k^{(\mu)}$ can be computed from values $N_k^{(\mu')}$ by equation $N_k^{(\mu)} = N_{k \oplus \mu' \oplus \mu}^{(\mu')}$, for all k . In other words, vectors $(N_{0x00}^{(\mu)}, \dots, N_{0xFF}^{(\mu)})$ and $(N_{0x00}^{(\mu')}, \dots, N_{0xFF}^{(\mu')})$ are permutations of each other; by consequence, term $S!/(N_{0x00}^{(\mu)}! \cdots N_{0xFF}^{(\mu)}!)$ in equation (1) can safely be ignored when determining the largest λ_μ . Furthermore, computing and comparing $\log(\lambda_\mu)$ instead of λ_μ makes the computation slightly more efficient.

4.2 Our Double-byte Bias Attack

As we have seen, Algorithm 4 allows the recovery of the initial 256 bytes of plaintext when multiple encryptions under different keys are observed by the attacker. In the following, we describe an algorithm which allows the recovery of plaintext bytes at *any* position in the plaintext.

Furthermore, the algorithm does not require the plaintext to be encrypted under many different keys but works equally well for plaintexts repeatedly encrypted under a *single* key.

Our algorithm is based on biases in the distribution of consecutive bytes (Z_r, Z_{r+1}) of the RC4 keystream that occur as long term biases, i.e., that appear periodically at regular intervals in the keystream. As described in Section 3, we empirically measured the biases which are repeated with a period of 256 bytes. However, in 2^{50} experimentally generated keystream bytes we observed no significant new biases besides those already identified by Fluhrer and McGrew [12]; for the purpose of constructing our algorithm, we hence use the biases described in Table 1 and assume that all other consecutive byte pairs are equally likely to appear in the keystream. In other words, we assume that we have accurate estimates p_{r,k_1,k_2} such that

$$p_{r,k_1,k_2} = \Pr[(Z_r, Z_{r+1}) = (k_1, k_2)]$$

for $1 \leq r \leq 256$ and $0x00 \leq k_1, k_2 \leq 0xFF$, where the probability is taken over all possible configurations of the internal state S and the index j of the RC4 keystream generation algorithm.⁹ Note that, since these probabilities express biases that are repeated with a period of 256 bytes, we have $p_{r,k_1,k_2} = p_{(r \bmod 256), k_1, k_2}$ for all r, k_1, k_2 .

Let L be an integer multiple of 256. In the following description of our plaintext recovery algorithm, we assume that a fixed L -byte plaintext $P = P_1 || \cdots || P_L$ is encrypted repeatedly under a single key, i.e., we consider a ciphertext C obtained by encrypting $P || \cdots || P$. (In fact, it is sufficient for our attack that the target plaintext bytes form a subsequence of consecutive bytes that are constant across blocks of L bytes.) Let C_j denote the substring of C corresponding to the encryption of the j -th copy of P , and let $C_{j,r}$ denote the r -th byte of C_j (i.e., $C_{j,r}$ corresponds to byte $(j-1) \cdot L + r$ of C).

Given this setting, it seems reasonable to take an approach towards plaintext recovery similar to that of Algorithm 4: for each position r , the most likely plaintext pair (μ_r, μ_{r+1}) could be computed from the ciphertext bytes $\{(C_{j,r}, C_{j,r+1})\}_{1 \leq j \leq S}$ and the probability estimates $\{p_{r,k_1,k_2}\}_{0x00 \leq k_1, k_2 \leq 0xFF}$. In other words, a plaintext candidate would be obtained by splitting ciphertexts C into byte pairs and individually computing the most likely corresponding plaintext pairs.

However, by considering overlapping byte pairs, it is possible to construct a more accurate estimate of the likelihood of a plaintext candidate being correct than by just considering the likelihood of individual byte-pairs. More specifically, for any plaintext candidate $P' = \mu_1 || \cdots || \mu_L$ we compute an estimated likelihood $\lambda_{P'} = \lambda_{\mu_1 || \cdots || \mu_L}$ for

P' being correct via the recursion

$$\lambda_{\mu_1||\cdots||\mu_{\ell-1}||\mu_\ell} = \delta_{\mu_\ell|\mu_{\ell-1}} \cdot \lambda_{\mu_1||\cdots||\mu_{\ell-1}} \quad (\ell \leq L), \quad (2)$$

where $\delta_{\mu_\ell|\mu_{\ell-1}}$ denotes the probability that $P_\ell = \mu_\ell$ assuming $P_{\ell-1} = \mu_{\ell-1}$, and $\lambda_{\mu_1||\cdots||\mu_{\ell-1}}$ is the estimated likelihood of $\mu_1||\cdots||\mu_{\ell-1}$ being the correct $(\ell-1)$ -length prefix of P . We show below how values $\delta_{\mu_\ell|\mu_{\ell-1}}$ can be computed given the ciphertext bytes $\{(C_{j,\ell-1}, C_{j,\ell})\}_{1 \leq j \leq S}$ and the probability estimates $\{p_{\ell-1,k_1,k_2}\}_{0x00 \leq k_1, k_2 \leq 0xFF}$. Note that, by rewriting equation (2) and assuming that $\lambda_{\mu_1} = \Pr[P_1 = \mu_1]$ is accurately known, we obtain likelihood estimate $\lambda_{P'} = \Pr[P_1 = \mu_1] \prod_{\ell=2}^L \delta_{\mu_\ell|\mu_{\ell-1}}$.

Our algorithm computes the plaintext candidate $P^* = \mu_1||\cdots||\mu_L$ which maximizes the estimated likelihood λ_{P^*} . This is done by exploiting the following easy-to-see optimality-preserving property: for all prefixes $\mu_1||\cdots||\mu_\ell$ of P^* , $\ell \leq L$, we have that $\lambda_{\mu_1||\cdots||\mu_{\ell-1}}$ is the largest likelihood among all $(\ell-1)$ -length plaintext candidates with $\mu_{\ell-1}$ as the last byte.

The basic idea of our algorithm is to iteratively construct P^* by considering the prefixes of P^* with increasing length. As just argued, these correspond to the (partial) plaintext candidates with the highest likelihood and a specific choice of the last byte value. However, when computing a candidate for a length $\ell \leq L$, it is not known in advance what the specific value of the last byte μ_ℓ should be. Our algorithm hence computes the most likely partial plaintext candidates for *all* possible values of μ_ℓ . More specifically, for each $(\ell-1)$ -length partial candidate $\mu_1||\cdots||\mu_{\ell-1}$ and any value μ_ℓ , we compute the likelihood of the ℓ -length plaintext candidate $\mu_1||\cdots||\mu_{\ell-1}||\mu_\ell$ via equation (2) as $\lambda_{\mu_1||\cdots||\mu_\ell} = \delta_{\mu_\ell|\mu_{\ell-1}} \cdot \lambda_{\mu_1||\cdots||\mu_{\ell-1}}$. Due to the optimality-preserving property, the string $\mu_1||\cdots||\mu_\ell$ with the highest likelihood will correspond to the most likely plaintext candidates of length ℓ with the last byte μ_ℓ . This guarantees that the ℓ -length prefix of (optimal) P^* will be among the computed candidates and, furthermore, when the length of P^* is reached, that P^* itself will be obtained.

To initialize the above process, the algorithm assumes that the first plaintext byte μ_1 of P is known with certainty, i.e., $\lambda_{\mu_1} = 1$ (this can, for example, be assumed if the attack is used to recover HTTP cookies from an encrypted HTTP(S) header). Likewise, the algorithm assumes that the last byte μ_L of P is known, i.e., $\lambda_{\mu_L} = 1$ (also this is the case when recovering HTTP cookies). This leads to a single μ_L being used in the last iteration of the above process which will then return the most likely plaintext candidate P^* . (See Remark 1 for how the algorithm can be modified to work without these assumptions.)

It remains to show the details of how $\delta_{\mu_{i+1}|\mu_i}$ can be computed. This is done similarly to the

maximum-likelihood computation of the probability estimate used in Algorithm 4. More precisely, each combination of index i , pair (μ_i, μ_{i+1}) , and ciphertext bytes $\{(C_{j,i}, C_{j,i+1})\}_{1 \leq j \leq S}$ induces a distribution on the keystream bytes $\{(Z_{(j-1)L+i}, Z_{(j-1)L+i+1})\}_{1 \leq j \leq S}$. The latter can be represented as a vector $(N_{i,0x00,0x00}, \dots, N_{i,0xFF,0xFF})$, where

$$N_{i,k_1,k_2} = |\{j | (C_{j,i}, C_{j,i+1}) = (k_1 \oplus \mu_i, k_2 \oplus \mu_{i+1})\}_{1 \leq j \leq S}| .$$

As in Section 4.1, we see that this vector follows a multinomial distribution, and that the probability that $(N_{i,0x00,0x00}, \dots, N_{i,0xFF,0xFF})$ will arise (i.e., the probability that (μ_i, μ_{i+1}) corresponds to the i -th and the $(i+1)$ -th plaintext bytes) is given by

$$\Pr[P_i = \mu_i \wedge P_{i+1} = \mu_{i+1} | C] = \quad (3)$$

$$\frac{S!}{N_{i,0x00,0x00}! \cdots N_{i,0xFF,0xFF}!} \prod_{k_1, k_2 \in \{0x00, \dots, 0xFF\}} p_{i,k_1,k_2}^{N_{i,k_1,k_2}} .$$

We can now compute $\delta_{\mu_{i+1}|\mu_i}$ as

$$\begin{aligned} \delta_{\mu_{i+1}|\mu_i} &= \Pr[P_{i+1} = \mu_{i+1} | P_i = \mu_i \wedge C] \\ &= \frac{\Pr[P_i = \mu_i \wedge P_{i+1} = \mu_{i+1} | C]}{\Pr[P_i = \mu_i | C]} . \end{aligned} \quad (4)$$

We assume that no significant single-byte biases are present in the keystream, i.e., that $\Pr[P_i = \mu_i | C]$ is uniform over the possible plaintext values μ_i . Under this condition, since the term will stay invariant for all plaintext candidates, we can ignore the contribution of factor $1/\Pr[P_i = \mu_i | C]$ in (4), when comparing probability estimates. This is likewise the case for the terms $S!/(N_{i,0x00,0x00}, \dots, N_{i,0xFF,0xFF})$ in (3), due to similar observations as made for Algorithm 4.

We combine the results of the discussion from the preceding paragraphs, including the proposed optimizations, to obtain our double-byte bias attack in Algorithm 5.

Remark 1. The above assumption, that the first and last byte of the plaintext P is known, can easily be avoided. Specifically, if the first byte is unknown, Algorithm 5 can be initialized by computing, for each possible value μ_2 , the most likely pairs (μ_1, μ_2) . This can be done based on the ciphertext bytes $\{(C_{j,1}, C_{j,2})\}_{1 \leq j \leq S}$ and the probability estimates $\{p_{1,k_1,k_2}\}_{0x00 \leq k_1, k_2 \leq 0xFF}$. Likewise, if the last byte is unknown, the algorithm will identify P^* as the plaintext candidate with the highest likelihood estimate among the computed plaintext candidates of length L . Note, however, that knowing the first and last plaintext byte will lead to a more accurate likelihood estimate and will thereby increase the success rate of the algorithm.

Algorithm 5: Double-byte bias attack

input : C – encryption of S copies of fixed plaintext P
($C_{j,r}$ denotes the r -th byte of the substring of C encrypting the j -th copy of P)
 L – length of P in bytes (must be a multiple of 256)
 μ_1 and μ_L – the first and last byte of P
 $\{p_{r,k_1,k_2}\}_{1 \leq r \leq L-1, 0x00 \leq k_1, k_2 \leq 0xFF}$ – keystream distribution

output: estimate P^* for plaintext P

notation: let $\max_2(Q)$ denote $(P, \lambda) \in Q$ such that $\lambda \geq \lambda' \forall (P', \lambda') \in Q$

begin

```
 $N_{(r,k_1,k_2)} \leftarrow 0 \quad \text{for all } 1 \leq r < L, 0x00 \leq k_1, k_2 \leq 0xFF$ 
for  $j = 1$  to  $S$  do
  for  $r = 1$  to  $L - 1$  do
     $N_{(r,C_{j,r},C_{j,r+1})} \leftarrow N_{(r,C_{j,r},C_{j,r+1})} + 1$ 
   $Q \leftarrow \{(\mu_1, 0)\}$ 
  for  $r = 1$  to  $L - 2$  do
     $Q_{ext} \leftarrow \{\} \quad \text{// List of plaintext candidates of length } r + 1$ 
    for  $\mu_{r+1} = 0x00$  to  $0xFF$  do
       $Q_{\mu_{r+1}} \leftarrow \{\} \quad \text{// List of plaintext candidates ending with } \mu_{r+1}$ 
      for each  $(P', \lambda_{P'}) \in Q$  do
         $P' \rightarrow \mu_1 || \dots || \mu_r$ 
         $\lambda_{P'||\mu_{r+1}} \leftarrow \lambda_{P'} + \sum_{k_1=0x00}^{0xFF} \sum_{k_2=0x00}^{0xFF} N_{(r, k_1 \oplus \mu_r, k_2 \oplus \mu_{r+1})} \cdot \log p_{(r, k_1, k_2)}$ 
         $Q_{\mu_{r+1}} \leftarrow Q_{\mu_{r+1}} \cup \{(P' || \mu_{r+1}, \lambda_{P'||\mu_{r+1}})\}$ 
       $Q_{ext} \leftarrow Q_{ext} \cup \{\max_2(Q_{\mu_{r+1}})\}$ 
     $Q \leftarrow Q_{ext}$ 
   $Q_{\mu_L} \leftarrow \{\} \quad \text{// List of plaintext candidates ending with } \mu_L$ 
  for each  $(P', \lambda_{P'}) \in Q$  do
     $P' \rightarrow \mu_1 || \dots || \mu_{L-1}$ 
     $\lambda_{P'||\mu_L} \leftarrow \lambda_{P'} + \sum_{k_1=0x00}^{0xFF} \sum_{k_2=0x00}^{0xFF} N_{(r, k_1 \oplus \mu_{L-1}, k_2 \oplus \mu_L)} \cdot \log p_{(r, k_1, k_2)}$ 
     $Q_{\mu_L} \leftarrow Q_{\mu_L} \cup \{(P' || \mu_L, \lambda_{P'||\mu_L})\}$ 
   $(P^*, \lambda_{P^*}) \leftarrow \max_2(Q_{\mu_L})$ 
return  $P^*$ 
```

5 Experimental Results

Through simulation, we measured the performance of the single-byte and double-byte bias attacks. We furthermore validated our algorithms in real attack scenarios.

5.1 Simulation of Single-byte Bias Attack

We simulated the first plaintext recovery attack described in Section 4. We used RC4 keystreams for 2^{44} random keys to estimate the per-output-byte probabilities $\{p_{r,k}\}_{1 \leq r \leq 256, 0x00 \leq k \leq 0xFF}$. We then ran the attack in Algorithm 4 256 times for each of $S = 2^{24}, 2^{25}, \dots, 2^{32}$ sessions to estimate the attack’s success rate. The results for $S = 2^{24}, 2^{26}, \dots, 2^{30}$ are shown in Figures 4–7. In each figure, we show the success rate in recovering the correct plaintext byte versus the position r of the byte in the output stream (but recall that, in practice, the first 36 bytes

are not interesting as they contain the Finished message). Some notable features of these figures are:

- Even with as few as 2^{24} sessions, some positions of the plaintext are correctly recovered with high probability. The ones with highest probability seem to arise because of the key-length-dependent biases that we observed in positions that are multiples of 16. These large biases make it easier to recover the correct plaintext bytes when compared to other ciphertext positions.
- With $S = 2^{26}$ sessions, the first 46 plaintext bytes are recovered with rate at least 50% per byte.
- With $S = 2^{32}$ sessions (not shown here; see [4]), all of the first 256 bytes of output are recovered with rate close to 100%: the rate is at least 96% in all positions, and is 100% for all but 12 positions.

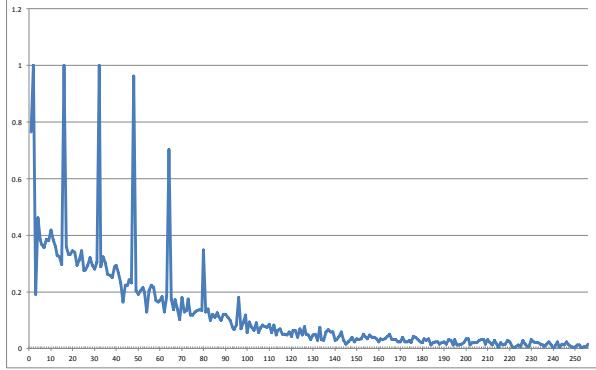


Figure 4: Recovery rate of the single-byte bias attack for $S = 2^{24}$ sessions for first 256 bytes of plaintext (based on 256 experiments).

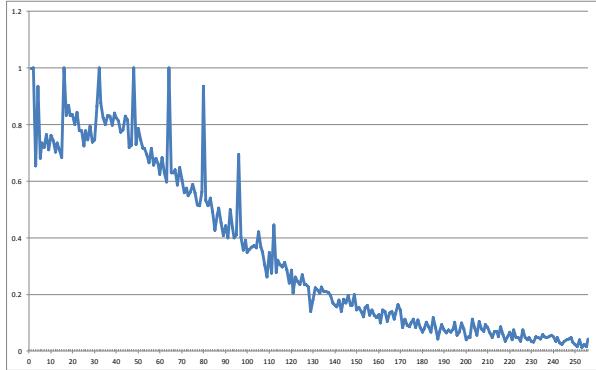


Figure 5: Recovery rate of the single-byte bias attack for $S = 2^{26}$ sessions for the first 256 bytes of plaintext (based on 256 experiments).

- The rate at which bytes are correctly recovered increases steadily as the number of sessions S is increased, with all but the last few bytes being reliably recovered already for 2^{31} trials.

Secondly, we executed the recovery attack in a setting where plaintexts are encoded with a 4-bits-per-byte encoding scheme using the characters ‘0’ to ‘9’ and ‘a’ to ‘f’. Such restricted plaintext character sets are routinely used in different applications [4]; for instance, in the popular PHP server-side scripting language, the encoding of HTTP cookies can be limited to a representation with 4 bits per character [20]. We reused the probability estimates $\{p_{r,k}\}_{1 \leq r \leq 256, 0 \leq k \leq 0xFF}$ for the RC4 keystream bytes generated for the simulation above, and ran a modified version of Algorithm 4 which takes into account the restricted plaintext space. The modified algorithm was run 256 times for each of $S = 2^{24}, 2^{25}, \dots, 2^{32}$ sessions. The results for $S = 2^{24}$, $S = 2^{26}$ and $S = 2^{28}$ are shown in Figures 8–10. For comparison, the figures include the success rate of the original attack

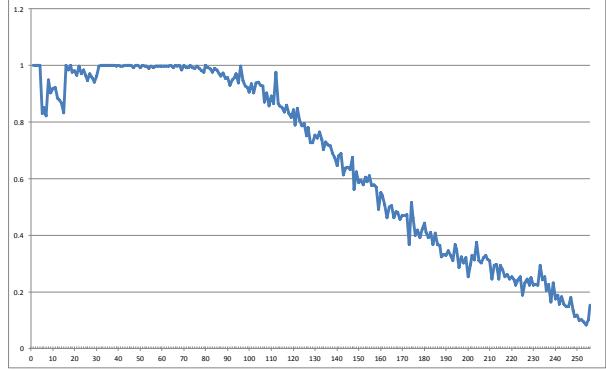


Figure 6: Recovery rate of the single-byte bias attack for $S = 2^{28}$ sessions for the first 256 bytes of plaintext (based on 256 experiments).

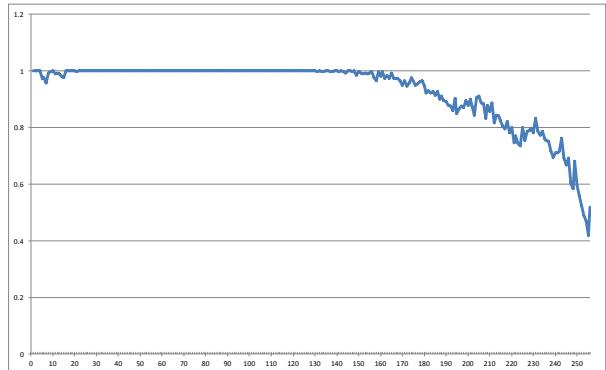


Figure 7: Recovery rate of the single-byte bias attack for $S = 2^{30}$ sessions for the first 256 bytes of plaintext (based on 256 experiments).

for an unrestricted plaintext space. We note:

- With $S = 2^{26}$ sessions, the first 112 plaintext bytes are recovered with rate at least 50% per byte. This represents a marked improvement over the case of an unrestricted plaintext space, where only the first 46 bytes were recovered with rate at least 50% per byte.
- With $S = 2^{24}, \dots, 2^{28}$ sessions, the recovery attack for the restricted plaintext space has a better success rate than the recovery attack for the unrestricted plaintext space with twice the number of sessions (i.e. $S = 2^{25}, \dots, 2^{29}$) for almost all positions.

5.2 Simulation of Double-byte Bias Attack

We simulated the second plaintext recovery attack based on Algorithm 5. In the simulation, we encrypted $S = 1 \cdot 2^{30}, \dots, 13 \cdot 2^{30}$ copies of the same 256-byte plaintext

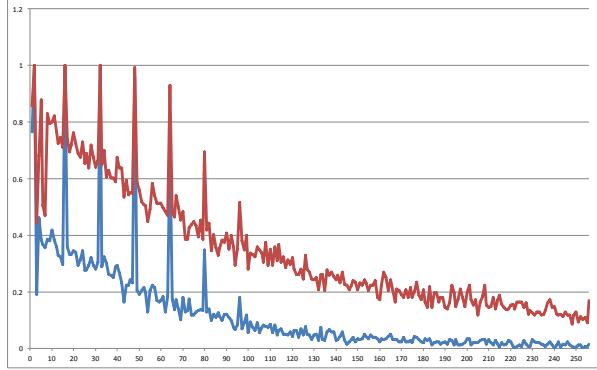


Figure 8: Recovery rates for the restricted plaintext space (red) and the original single-byte bias attack (blue) for $S = 2^{24}$ sessions (based on 256 experiments).

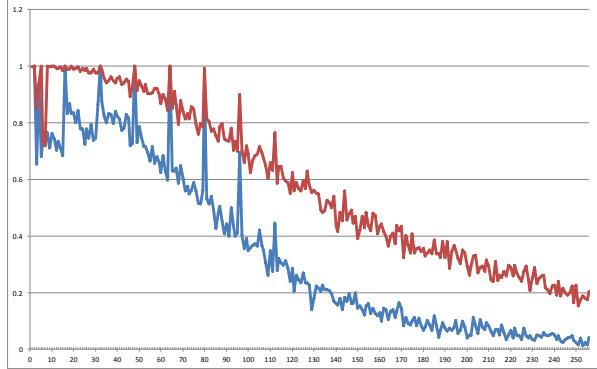


Figure 9: Recovery rates for the restricted plaintext space (red) and the original single-byte bias attack (blue) for $S = 2^{26}$ sessions (based on 256 experiments).

and attempted to recover 16 bytes located at a fixed position in the plaintext. More precisely, we simulated an attack in which we assume the first byte of the plaintext is known, the following 16 bytes are the unknown bytes targeted by the attack, and the byte immediately following these is known. The remaining bytes are assumed not to be of interest in the attack. This attack scenario is very similar to the case in which an adversary attempts to recover a cookie value from an HTTP request. Depending on the number of plaintext copies, we used between one and five 128-bit RC4 keys for the encryption¹⁰. As highlighted in Section 4.2, we used the biases described by Fluhrer-McGrew [12] to compute the probability estimates $\{p_{r,k_1,k_2}\}_{1 \leq r \leq 255, 0x00 \leq k_1, k_2 \leq 0xFF}$ required by Algorithm 5.

The attack was run 128 times for each of $S = 1 \cdot 2^{30}, \dots, 13 \cdot 2^{30}$ encrypted copies of the plaintext to estimate the success rate of the attack. The results are shown in Figure 11: the dashed line shows the average fraction of successfully recovered plaintext bytes versus the num-

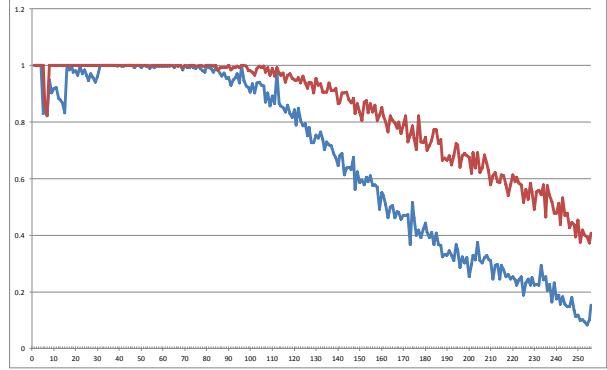


Figure 10: Recovery rates for the restricted plaintext space (red) and the original single-byte bias attack (blue) for $S = 2^{28}$ sessions (based on 256 experiments).

ber of encrypted plaintexts, whereas the solid line shows the success rate of recovering the full 16-byte plaintext versus the number of encrypted plaintexts. We note:

- With $S = 6 \cdot 2^{30}$ encrypted copies of the plaintext, more than 50% of the plaintext is correctly recovered on average. Furthermore, in 19% of the 128 trials, the full 16-byte plaintext was recovered.
- With $S = 8 \cdot 2^{30}$ encrypted copies of the plaintext, the full plaintext is correctly recovered in significantly more than 50% of the 128 trials (more precisely, the full plaintext was recovered in 72% of the trials).
- With $S = 13 \cdot 2^{30}$ the full plaintext was recovered in all trials.
- The rate at which the full plaintext is correctly recovered increases fairly rapidly after $S = 5 \cdot 2^{30}$ copies of the plaintext are encrypted, and with $S = 11 \cdot 2^{30}$, the full plaintext is correctly recovered in nearly all trials (99%).

In addition, similar to Section 5.1, we simulated the attack for plaintexts encoded with a 6-bits-per-byte (base64) and a 4-bits-per-byte encoding scheme. Specifically, we firstly ran a modified version of Algorithm 5 which takes into account the restricted plaintext space by only considering candidate plaintext bytes which correspond to byte-values used in a base64 encoding. Furthermore, we used a plaintext where the 16 bytes targeted by the attack consisted of bytes with a byte-value corresponding to the character ‘b’, which is a valid base64 encoded message. As in the attack above for a non-restricted plaintext space, the probability estimates $\{p_{r,k_1,k_2}\}_{1 \leq r \leq 255, 0x00 \leq k_1, k_2 \leq 0xFF}$ were based on the biases from [12]. The attack was run 128 times for each of

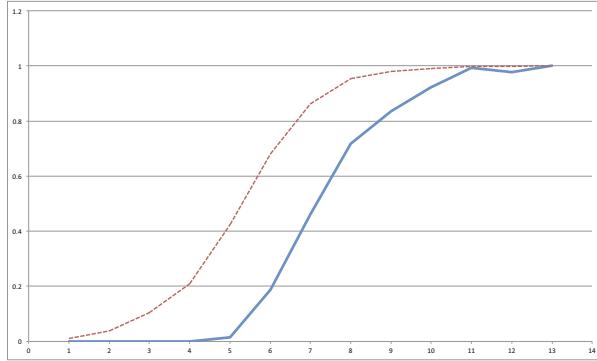


Figure 11: Average fraction of successfully recovered plaintext bytes (dashed line), and success rate for recovering the full 16-byte plaintext (solid line) of the double-byte bias attack based on 128 experiments. The unit of the x -axis is 2^{30} encrypted copies of the plaintext.

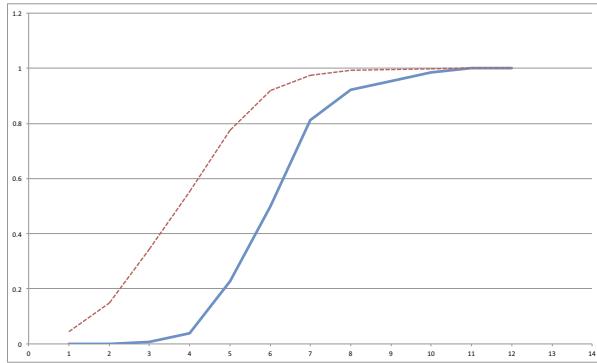


Figure 12: Average fraction of successfully recovered plaintext bytes (dashed line), and success rate for recovering the full 16-byte plaintext (solid line) of the double-byte bias attack for base64 encoded plaintexts (based on 128 experiments). The unit of the x -axis is 2^{30} encrypted copies of the plaintext.

$S = 1 \cdot 2^{30}, \dots, 12 \cdot 2^{30}$ encrypted copies of the plaintext, and the results are shown in Figure 12. We note:

- With $S = 4 \cdot 2^{30}$ encrypted copies of the plaintext, more than 50% of the plaintext is correctly recovered on average. Furthermore, in 4% of the 128 trials, the full 16-byte plaintext is recovered.
- With $S = 6 \cdot 2^{30}$ encrypted copies of the plaintext, the full plaintext is correctly recovered in 50% of the 128 trials.
- With $S = 10 \cdot 2^{30}$ encrypted copies of the plaintext, the full plaintext is correctly recovered in nearly all trials (98%).

Regarding the 4-bit-per-byte encoding scheme, we again assumed a plaintext character set consisting of

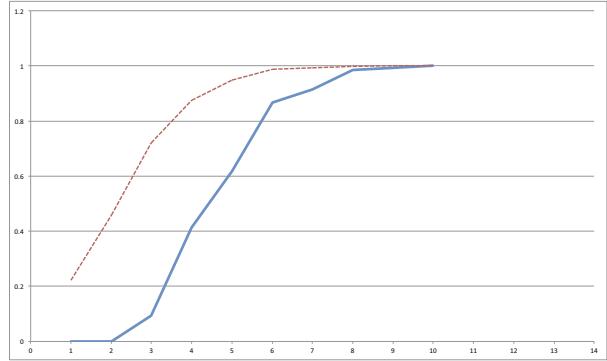


Figure 13: Average fraction of successfully recovered plaintext bytes (dashed line), and success rate for recovering the full 16-byte plaintext (solid line) of the double-byte bias attack for 4-bit-per-byte encoded plaintexts (based on 128 experiments). The unit of the x -axis is 2^{30} encrypted copies of the plaintext.

‘0’ to ‘9’ and ‘a’ to ‘f’. The setup was similar to the above experiment for base64 encoded messages: we ran a modified version of Algorithm 5 which takes into account the restricted plaintext space, the probability estimates $\{p_{r,k_1,k_2}\}_{1 \leq r \leq 255, 0 \leq k_1, k_2 \leq 0xFF}$ was based on the biases from [12], and we used a plaintext consisting of bytes with a byte-value corresponding to the character ‘b’. The attack was run 128 times for each of $S = 1 \cdot 2^{30}, \dots, 10 \cdot 2^{30}$ encrypted copies of the plaintext, and the results can be seen in Figure 13. We note:

- With $S = 3 \cdot 2^{30}$ encrypted copies of the plaintext, significantly more than 50% of the plaintext is correctly recovered on average (more precisely, 72% is recovered correctly on average).
- With $S = 5 \cdot 2^{30}$ encrypted copies of the plaintext, the full plaintext is recovered in more than 50% of the 128 trials.
- With $S = 8 \cdot 2^{30}$ encrypted copies of the plaintext, the full plaintext is recovered in nearly all trials (98%).

5.3 Practical Validation

We tested the success rates of our plaintext recovery algorithms in realistic attack settings involving web servers and browsers that are connected through TLS-secured network links. Here, we report on the results.

5.3.1 Validating the operation of RC4 in TLS

We first experimentally verified that the OpenSSL implementation of TLS does indeed use RC4 in the way

explained in Section 2.2, in particular without discarding any initial keystream bytes. We did this by setting up an OpenSSL version 1.0.1c client and server running in a virtualised environment, making use of `s_client` and `s_server`, generic tools that are available as part of the OpenSSL distribution package. The two virtual machines were running Ubuntu 12.10 and kernel version 3.5.0-17.

5.3.2 Validating the single-byte bias attack

Recall that our single-byte bias attack targets the first 256 bytes of plaintext across multiple TLS sessions or connections with random keys. In order to efficiently generate the large number of ciphertexts needed to test our attack, we again used the `s_client` and `s_server` tools, this time modifying the `s_client` source code to force a session resumption for each TLS packet sent.

Using this approach, we were able to generate around 2^{21} encryptions of a fixed plaintext per hour; with 2^{25} recorded ciphertexts, we obtained results comparable to the simulation of our single-byte bias attack reported in Section 5.1 above. A second possible approach to ensure frequently enough rekeying is to actively interfere with the TLS session after each ciphertext is sent, causing it to fail and be restarted, by injecting a bad TLS packet or by resetting the corresponding TCP connection.

We admit that we do not currently have an automated mechanism for forcing session resumption, e.g., from JavaScript. However, JavaScript running in the browser can trigger the browser to establish a fresh TLS session (with a fresh, random key) after each HTTP connection torn down by the attacker. We estimate that this second approach would be significantly slower than using session resumption because of the additional overhead of running the full TLS Handshake. Thus, even though our double-byte bias attack has higher complexity in terms of its ciphertext requirements than our single-byte bias attack, in practice it could be the more efficient attack in terms of total running time, because it can be executed in a single session (or a small number of sessions).

Furthermore, while the single-byte bias attack successfully recovered fixed plaintext bytes in the initial 256 bytes of the TLS ciphertexts, our subsequent experimentation with modern web browsers revealed that these bytes consisted mostly of less interesting HTTP headers rather than cookies. For this reason, after this basic validation, we switched our experimental focus to the double-byte bias attack.

5.3.3 Validating the double-byte bias attack

The double-byte bias attack does not rely on session resumption or session renegotiation and is hence easier to

implement in practice. As our experimental setup for this attack, we used a network comprising three (non-virtualized) nodes: a legitimate web server (`www.abc.com`) that serves 16-byte secure cookies over HTTPS, a malicious web server (`www.evil.com`) serving a malicious JavaScript, and a client running a web browser representing a user. The legitimate and malicious web servers run Apache and PHP. For the client, we experimented with various browsers, including Firefox, Opera and Chrome. The nodes were connected through a 100 Mbps Ethernet link; they were equipped with Intel Core i7 processors with 2.3 GHz cores and 16 GB of RAM. None of our experiments used all available CPU resources, nor saturated the network bandwidth.

In this setup, we let the client visit `https://www.abc.com`. This will result in the legitimate web server sending the client a secure cookie which will be stored by the client’s browser. This cookie will be the target of the attack. We then let the client visit `http://www.evil.com` and run the malicious JavaScript served by the malicious web server. Note that the same-origin policy (SOP) implemented by the client’s browser will prevent the JavaScript from directly accessing the secure cookie. However, the JavaScript will direct repeated HTTP requests to the legitimate server over TLS (i.e. using HTTPS)¹¹. The client’s browser will then automatically attach the cookie to each request and thereby repeatedly encrypt the target cookie as required in our attack.

The JavaScript uses XMLHttpRequest objects¹² to send the requests. We tested GET, POST, and HEAD requests, but found that POST requests gave the best performance (using Firefox). Furthermore, we found that the requests needed to be send in blocks to ensure that the browser stayed responsive and didn’t become overloaded.

For all the browsers we tested (Firefox, Chrome, and Opera), we found that the requests generated by the JavaScript resulted in TLS messages containing more than 256 bytes of ciphertext. To keep the target cookie in a fixed position in the TLS message (modulo 256) as needed for the double-byte bias attack, we therefore added padding by manipulating the HTTP headers in the request to bring the encrypted POST requests up to exactly 512 bytes. This padding introduces some overhead to the attack. The exact amount and location of padding needed is browser-dependent, since different browsers behave differently in terms of the content and order of HTTP headers included in POST requests. In practice, then, the attacker’s JavaScript would need to perform some browser fingerprinting before carrying out its attack.

As an alternative method for generating request to the legitimate web server, we tried replacing the JavaScript

code with basic HTML code, using HTML tags such as `img`, pointing to `https://www.abc.com`. The target cookie was still sent in every request, but we found this approach to be less effective (i.e. slower) than using JavaScript.

For Firefox with 512-byte ciphertexts encrypting padded XMLHttpRequest POST requests, we were able to generate 6 million ciphertexts per hour on our network, with each request containing the target cookie in the same position (modulo 256) in the corresponding plaintext. Given that our attack needs on the order of $13 \cdot 2^{30}$ encryptions to recover a 16-byte plaintext with high success probability, we estimate that the running time for the whole attack would be on the order of 2000 hours using our experimental setup. The attack generates large volumes of network traffic over long periods of time, and so should not be considered a practical threat. Nevertheless, it demonstrates that our double-byte bias attack does work in principle.

6 Discussion and Conclusions

We have shown that plaintext recovery for RC4 in TLS is possible for the first about 200 or so bytes of the plaintext stream (after the `Finished` message), provided sufficiently many independent encryptions of the same plaintext are available. The number of encryptions required (around 2^{28} to 2^{32} for reliable recovery) is large, but not completely infeasible. We have also shown that plaintext recovery for RC4 is possible from arbitrary positions in the plaintext, given enough encryptions of the same plaintext bytes. Here, the number of encryptions required is rather higher (around $13 \cdot 2^{30}$), but the attack is more flexible and more efficient in practice because it avoids rerunning the TLS Handshake. Certainly, the security level provided by RC4 in TLS is *far* below the strength implied by the 128-bit key in TLS.

This said, it would be incorrect to describe the attacks as being a practical threat to TLS today. However, our attacks are open to further enhancement, using, for example, the ability of our algorithms to output likelihoods for candidate plaintext bytes coupled with more sophisticated plaintext models. It may also be possible to enhance the rate of ciphertext generation in browsers using methods beyond our knowledge. It would seem dangerous to assume that the attacks will not be improved by other researchers in future.

There are countermeasures to the attacks. We discussed these countermeasures extensively with vendors during the disclosure process that we followed prior to making our attacks public. They include: discarding the initial keystream bytes output by RC4, as recommended in [19]; fragmenting the initial HTTP requests at the browser so that the initial keystream bytes are mostly (or entirely) used to encrypt MAC fields; adding random

padding to HTTP requests; and limiting the lifetime of cookies or the number of times they can be sent from the browser. The first countermeasure cannot easily be implemented in TLS because it would require mass co-ordination between the many different client and server implementations. The first two countermeasures are not effective against our double-byte bias attack. The third countermeasure can be relatively easily implemented in browsers but increases the complexity of our attacks rather than defeating them completely. The fourth countermeasure is currently effective, but not immune to further improvements of our attacks. Some vendors (e.g. Opera¹³) have implemented a combination of these (and other) countermeasures; others (e.g. Google in Chrome) are focussing on implementing TLS 1.2 and AES-GCM.

We recognise that, with around 50% of TLS traffic currently using RC4, recommending that it be avoided completely in TLS is not a suggestion to be made lightly. Nevertheless, given the rather small security margin provided by RC4 against our attacks, our recommendation is that RC4 should henceforth be avoided in TLS, and deprecated as soon as possible.

Acknowledgements

We thank David McGrew for raising the question of the security of RC4 in TLS.

References

- [1] Wireless LAN medium access control (MAC) and physical layer (PHY) specification, 1997.
- [2] Wireless LAN medium access control (MAC) and physical layer (PHY) specification: Amendment 6: Medium access control (MAC) security enhancements, 2004.
- [3] ALFARDAN, N., AND PATERSON, K. G. Lucky 13: Breaking the TLS and DTLS record protocols. In *IEEE Symposium on Security and Privacy* (2013).
- [4] ALFARDAN, N. J., BERNSTEIN, D. J., PATERSON, K. G., POETTERING, B., AND SCHULDT, J. C. N. On the security of RC4 in TLS and WPA. Information Security Group at Royal Holloway, University of London, 2013. <http://www.isg.rhul.ac.uk/tls/RC4biases.pdf>.
- [5] AMMAN, B. Personal communication, February 2013.
- [6] CANVEL, B., HILTGEN, A., VAUDENAY, S., AND VUAGNOUX, M. Password interception in a SSL/TLS channel. *Advances in Cryptology-CRYPTO 2003* (2003), 583–599.
- [7] DIERKS, T., AND ALLEN, C. The TLS Protocol Version 1.0. RFC 2246, Internet Engineering Task Force, Jan. 1999.
- [8] DIERKS, T., AND RESCORLA, E. The Transport Layer Security (TLS) Protocol Version 1.1. RFC 4346, Internet Engineering Task Force, Apr. 2006.
- [9] DIERKS, T., AND RESCORLA, E. The Transport Layer Security (TLS) Protocol Version 1.2. RFC 5246, Internet Engineering Task Force, Aug. 2008.
- [10] DUONG, T., AND RIZZO, J. Here come the \oplus Ninjas. Unpublished manuscript, 2011.

- [11] FLUHRER, S. R., MANTIN, I., AND SHAMIR, A. Weaknesses in the key scheduling algorithm of RC4. In *Selected Areas in Cryptography* (2001), S. Vaudenay and A. M. Youssef, Eds., vol. 2259 of *Lecture Notes in Computer Science*, Springer, pp. 1–24.
- [12] FLUHRER, S. R., AND MCGREW, D. Statistical analysis of the alleged RC4 keystream generator. In *FSE* (2000), B. Schneier, Ed., vol. 1978 of *Lecture Notes in Computer Science*, Springer, pp. 19–30.
- [13] ISOBE, T., OHIGASHI, T., WATANABE, Y., AND MORII, M. Full plaintext recovery attack on broadcast RC4. In *Preproceedings of FSE* (2013).
- [14] JAGANATHAN, K., ZHU, L., AND BREZAK, J. The RC4-HMAC Kerberos Encryption Types Used by Microsoft Windows. RFC 4757 (Informational), Dec. 2006.
- [15] MAITRA, S., PAUL, G., AND SENGUPTA, S. Attack on broadcast RC4 revisited. In *FSE* (2011), A. Joux, Ed., vol. 6733 of *Lecture Notes in Computer Science*, Springer, pp. 199–217.
- [16] MANTIN, I. Predicting and distinguishing attacks on rc4 keystream generator. In *EUROCRYPT* (2005), R. Cramer, Ed., vol. 3494 of *Lecture Notes in Computer Science*, Springer, pp. 491–506.
- [17] MANTIN, I., AND SHAMIR, A. A practical attack on broadcast RC4. In *FSE* (2001), M. Matsui, Ed., vol. 2355 of *Lecture Notes in Computer Science*, Springer, pp. 152–164.
- [18] MCGREW, D., AND BAILEY, D. AES-CCM Cipher Suites for Transport Layer Security (TLS). RFC 6655 (Proposed Standard), 2012.
- [19] MIRONOV, I. (Not so) random shuffles of RC4. In *CRYPTO* (2002), M. Yung, Ed., vol. 2442 of *Lecture Notes in Computer Science*, Springer, pp. 304–319.
- [20] PHP DOCUMENTATION GROUP. PHP manual, Feb 2013. <http://www.php.net/manual/en/session.configuration.php#ini.session.hash-bits-per-character>.
- [21] SALOWEY, J., CHOUDHURY, A., AND MCGREW, D. AES Galois Counter Mode (GCM) Cipher Suites for TLS. RFC 5288 (Proposed Standard), Aug. 2008.
- [22] SEN GUPTA, S., MAITRA, S., PAUL, G., AND SARKAR, S. Proof of empirical RC4 biases and new key correlations. In *Selected Areas in Cryptography* (2011), pp. 151–168.
- [23] SEN GUPTA, S., MAITRA, S., PAUL, G., AND SARKAR, S. (Non-) random sequences from (non-) random permutations – analysis of RC4 stream cipher. *Journal of Cryptology to appear* (2013).
- [24] SEPEHRDAD, P., VAUDENAY, S., AND VUAGNOUX, M. Discovery and exploitation of new biases in RC4. In *Selected Areas in Cryptography* (2010), A. Biryukov, G. Gong, and D. R. Stinson, Eds., vol. 6544 of *Lecture Notes in Computer Science*, Springer, pp. 74–91.
- [25] SEPEHRDAD, P., VAUDENAY, S., AND VUAGNOUX, M. Statistical attack on RC4 – distinguishing WPA. In *EUROCRYPT* (2011), K. G. Paterson, Ed., vol. 6632 of *Lecture Notes in Computer Science*, Springer, pp. 343–363.
- [26] VAUDENAY, S., AND VUAGNOUX, M. Passive-only key recovery attacks on RC4. In *Selected Areas in Cryptography* (2007), C. M. Adams, A. Miri, and M. J. Wiener, Eds., vol. 4876 of *Lecture Notes in Computer Science*, Springer, pp. 344–359.

Notes

¹The research of the third, fourth and fifth authors was supported by an EPSRC Leadership Fellowship, EP/H005455/1. The research of the second author was supported by the National Science Foundation under grant 1018836 and by the Netherlands Organisation for Scientific Research (NWO) under grant 639.073.005.

²SSL Pulse (<https://www.trustworthyinternet.org/ssl-pulse/>) reported in June 2013 that only 15.1% of 170,000 websites surveyed support TLS 1.2; most major browsers currently do not support TLS 1.2.

³For examples of RC4 being recommended in the face of CBC attacks, see advice at Qualys' website <https://community.qualys.com/blogs/securitylabs/2011/10/17/mitigating-the-beast-attack-on-tls>, Ivan Ristic's personal blog <http://blog.ivanristic.com/2009/08/is-rc4-safe-for-use-in-ssl.html>, PhoneFactor's blog <http://blog.phonefactor.com/2011/09/23/slaying-beast-mitigating-the-latest-ssltls-vulnerability>, and F5's suggested Lucky 13 mitigation at <http://support.f5.com/kb/en-us/solutions/public/14000/100/sol14190.html>. Other examples abound on discussion forums and vendor websites.

⁴<http://notary.icsi.berkeley.edu>

⁵Unfortunately, we do not currently know of a way to trigger TLS session resumption from JavaScript running in a browser.

⁶The Internet Message Access Protocol (IMAP) is a popular protocol for email retrieval.

⁷Note that when attacking secret URL parameters from HTTPS connections or passwords from IMAP sessions such limitations do not arise.

⁸In detail, instead of an internal permutation \mathcal{S} of 8-bit values, Fluhrer and McGrew consider variants of RC4 based on permutations of 3-bit, 4-bit, and 5-bit values, respectively. Note that in these versions of RC4, the internal variables i and j , as well as the output Z_r , will also be 3-bit, 4-bit and 5-bit values, respectively.

⁹Note that the internal state \mathcal{S} , which corresponds to a permutation over byte values, will not be distributed as a random permutation immediately after the key scheduling algorithm is run, even if the used key is picked uniformly at random. Furthermore, j will not be random, but initialized to 0. However, random \mathcal{S} and j will be a close approximation after keystream bytes have been generated a short period of time (see [12] for further discussion of this property).

¹⁰Our experiments showed that there is no significant difference in the recovery rate when running the attack on encryptions of the plaintext generated by a single key and encryptions generated by a small number of different keys.

¹¹This is made possible by Cross-Origin Resource Sharing (CORS), a mechanism developed to allow JavaScript to make requests to another domain than the domain the script originates from.

¹²<http://www.w3.org/TR/XMLHttpRequest/>

¹³<http://my.opera.com/securitygroup/blog/2013/03/20/on-the-precariousness-of-rc4>

PCF: A Portable Circuit Format For Scalable Two-Party Secure Computation

Ben Kreuter
Computer Science Dept.
U. Virginia

Benjamin Mood
Computer and Info. Science Dept.
U. Oregon

abhi shelat
Computer Science Dept.
U. Virginia

Kevin Butler
Computer and Info. Science Dept.
U. Oregon

Abstract

A secure computation protocol for a function $f(x, y)$ must leak no information about inputs x, y during its execution; thus it is imperative to compute the function f in a data-oblivious manner. Traditionally, this has been accomplished by compiling f into a boolean circuit. Previous approaches, however, have scaled poorly as the circuit size increases. We present a new approach to compiling such circuits that is substantially more efficient than prior work. Our approach is based on online circuit compression and lazy gate generation. We implemented an optimizing compiler for this new representation of circuits, and evaluated the use of this representation in two secure computation environments. Our evaluation demonstrates the utility of this approach, allowing us to scale secure computation beyond any previous system while requiring substantially less CPU time and disk space. In our largest test, we evaluate an RSA-1024 signature function with more than 42 billion gates, that was generated and optimized using our compiler. With our techniques, the bottleneck in secure computation lies with the cryptographic primitives, not the compilation or storage of circuits.

1 Introduction

Secure function evaluation (SFE) refers to several related cryptographic constructions for evaluating functions on unknown inputs. Typically, these constructions require an *oblivious* representation of the function being evaluated, which ensures that the control flow of the algorithm will not depend on its input; in the two party case, boolean circuits are most frequently seen. These oblivious representations are often large, with millions and in some cases billions of gates even for relatively simple functions, which has motivated the creation of software tools for producing such circuits. While there has been substantial work on the practicality of secure function

evaluation, it was only recently that researchers began investigating the practicality of compiling such oblivious representations from high-level descriptions.

The work on generating boolean circuits for SFE has largely focused on two approaches. In one approach, a library for a general purpose programming language such as Java is created, with functions for emitting circuits [13, 20]. For convenience, these libraries typically include pre-built gadgets such as adders or multiplexers, which can be used to create more complete functions. The other approach is to write a compiler for a high level language, which computes and optimizes circuits based on a high level description of the functionality that may not explicitly state how the circuit should be organized [18, 21]. It has been shown in previous work that both of these approaches can scale up to circuits with at least hundreds of millions of gates on modern computer hardware, and in some cases even billions of gates [13, 18].

The approaches described above were limited in terms of their practical utility. Library-based approaches like HEKM [13] or VMCrypt [20] require users to understand the organization of the circuit description of their function, and were unable to apply any optimizations across modules. The Fairplay compiler [21] was unable to scale to circuits with only millions of gates, which excludes many interesting functions that have been investigated. The poor scalability of Fairplay is a result of the compiler first unrolling all loops and inlining all subroutines, storing the results in memory for later compiler stages. The PALC system [23] was more resource efficient than Fairplay, but did not attempt to optimize functions, relying instead on precomputed optimizations of specific subcircuits. The KSS12 [18] system was able to apply some global optimizations and used less memory than Fairplay, but also had to unroll all loops and store the complete circuit description, which caused some functions to require days to compile. Additionally, the language used to describe circuits in the KSS12 system was

brittle and difficult to use; for example, array index values could not be arbitrary functions of loop indices.

1.1 Our Approach

In this work, we demonstrate a new approach to compiling, optimizing, and storing circuits for SFE systems. At a high level, our approach is based on representing the function to be evaluated as a program that computes the circuit representation of the function, similar to the circuit library approaches described in previous work. Our compiler then optimizes this program with the goal of producing a smaller circuit. We refer to our circuit representation as the *Portable Circuit Format* (PCF).

When the SFE system is run, it uses our interpreter to load the PCF program and execute it. As the PCF program runs, it interacts with the SFE system, managing information about gates internally based on the responses from the SFE system itself. In our system, the circuit is ephemeral; it is not necessary to store the entire circuit, and wires will be deleted from memory once they are no longer required.

The key insight of our approach is that it is not necessary to unroll loops until the SFE protocol runs. While previous compilers discard the loop structure of the function, ours emits it as part of the control structure of the PCF program. Rather than dealing directly with wires, our system treats wire IDs as *memory addresses*; a wire is “deleted” by overwriting its location in memory. Loop termination conditions have only one constraint: they must not depend on any secret wire values. There is no upper bound on the number of loop iterations, and the programmer is responsible for ensuring that there are no infinite loops.

To summarize, we present the following contributions:

- A new compiler that has the same advantages as the circuit library approach
- A novel, more general algorithm for translating conditional statements into circuits
- A new representation of circuits that is more compact than previous representations which scales to arbitrary circuit sizes.
- A portable interpreter that can be used with different SFE execution systems regardless of the security model.

Our compiler is a *back end* that can read the bytecode emitted by a *front end*; thus our compiler allows *any* language to be used for SFE. Instead of focusing on global optimizations of boolean functions, our optimization strategy is based on using higher-level information

from the bytecode itself, which we show to be more effective and less resource-intensive. We present comparisons of our compiler with previous work and show experimental results using our compiler in two complete SFE systems, one based on an updated version of the KSS12 system and one based on HEKM. In some of our test cases, our compiler produced circuits only 30% as large as previous compilers starting from the same source code. With the techniques presented in this work, we demonstrate that the RSA algorithm with a real-world key size and real-world security level can be compiled and run in a garbled circuit protocol using a typical desktop computer. To the best of our knowledge, the RSA-1024 circuit we tested is larger than any previous garbled circuit experiment, with more than 42 billion gates. We also present preliminary results of our system running on smartphones, using a modified version of the HEKM system.

For testing purposes, we used the LCC compiler [8] as a front-end to our system. A high-level view of our system, with the LCC front-end, is given in Figure 1.

The rest of this paper is organized as follows: Section 2 is a review of SFE and garbled circuits; Section 3 presents an overview of bytecode languages; Section 4 explains our compiler design and describes our representation; Section 5 discusses the possibility of using different bytecode and SFE systems; Section 6 details the experiments we performed to evaluate our system and results of those experiments; Section 7 details other work which is related to our own; and Section 8 presents future lines of research.

2 Secure Function Evaluation

The problem of secure two-party computation is to allow two mutually distrustful parties to compute a function of their two inputs without revealing their inputs to the opposing party (privacy) and with a guarantee that the output could not have been manipulated (correctness). Yao was the first to show that such a protocol can be constructed for any computable function, by using the *garbled circuits* technique [30]. In his original formulation, Yao proposed a system that would allow users to describe the function in a high level language, which would then be compiled into a circuit to be used in the garbled circuits protocol. The first complete implementation of this design was the Fairplay system given by Malkhi et al. [21].

Oblivious Transfer One of the key building blocks in Yao’s protocol is *oblivious transfer*, a cryptographic primitive first proposed by Rabin [25]. In this primitive, the “sender” party holds a database of n strings, and the “receiver” party learns exactly k strings with the guarantee that the sender will not learn which k strings were

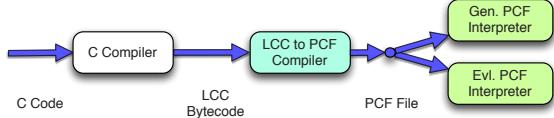


Figure 1: High-level design of our system. We take a C program and compile it down to the LCC bytecode. Our compiler then transforms the LCC bytecode to our new language PCF. Both parties then execute the protocol in their respective role in the SFE protocol. The interpreter could be any execution system.

sent and the receiver will not learn more than k strings; this is known as a k -out-of- n oblivious transfer. Given a public key encryption system it is possible to construct a 1-out-of-2 oblivious transfer protocol [7], which is the building block used in Yao’s protocol.

Garbled Circuits The core of Yao’s protocol is the construction of garbled circuits, which involves encrypting the truth table of each gate in a circuit description of the function. When the protocol is run, the truth values in the circuit will be represented as decryption keys for some cipher, with each gate receiving a unique pair of keys for its output wire. The keys for a gate’s input wires are then used to encrypt the keys for its output wires. Given a single key for each input wire of the circuit, the party that evaluates the circuit can decrypt a single key that represents a hidden truth value for each gate’s output wire, until the output gates are reached. Since this encryption process can be applied to any circuit, and since any computable function has a corresponding circuit family, this allows the construction of a secure protocol for any computable function.

The typical garbled circuit protocol has two parties though it can be expanded to more. Those two parties are Bob, the generator of the garbled circuit, and Alice, the evaluator of the garbled circuit. Bob creates the garbled circuit and therefore knows the decryption keys, but does not know which specific keys Alice uses. Alice will receive the input keys from Bob using an oblivious transfer protocol, and thus learns only one key for each input wire; if the keys are generated independent of Bob’s input, Alice will learn only enough to compute the output of the circuit.

Several variations on the Yao protocol have been published; a simple description of the garbling and evaluation process follows. Let $f : \{0,1\}^A \times \{0,1\}^B \rightarrow \{0,1\}^j \times \{0,1\}^k$ be a computable function, which will receive input bits from two parties and produce output bits for each party (not necessarily the same outputs). To garble the circuit, a block cipher $\langle E, D, G \rangle$ will be used.

For each wire in the circuit, Bob computes a pair of random keys $(k_0, k_1) \leftarrow (G(1^n), G(1^n))$, which represent

logical 0 and 1 values. For each of Alice’s outputs, Bob uses these keys to encrypt a 0 and a 1 and sends the pair of ciphertexts to Alice. Bob records the keys corresponding to his own outputs. The rest of the wires in the circuit are inputs to gates. For each gate, if the truth table is $[v_{0,0}, v_{0,1}, v_{1,0}, v_{1,1}]$, Bob computes the following ciphertext:

$$\begin{bmatrix} E_{k_{l,0}}(E_{k_{r,0}}(k_{v_{0,0}})), E_{k_{l,0}}(E_{k_{r,1}}(k_{v_{0,1}})) \\ E_{k_{l,1}}(E_{k_{r,0}}(k_{v_{1,0}})), E_{k_{l,1}}(E_{k_{r,1}}(k_{v_{1,1}})) \end{bmatrix}$$

where $k_{l,*}$ and $k_{r,*}$ are the keys for the left and right input wires (this can be generalized for gates with more than two inputs). The order of the four ciphertexts is then randomly permuted and sent to Alice.

Now that Alice has the garbled gates, she can begin evaluating the circuit. Bob will send Alice his input wire keys. Alice and Bob then use an oblivious transfer to give Alice the keys for her input wires. For each gate, Alice will only be able to decrypt one entry, and will receive one key for the gate’s output, and will continue to decrypt truth table entries until the output wires have been computed. Alice will then send Bob his output keys, and decrypt her own outputs.

Optimizations Numerous optimizations to the basic Yao protocol have been published [10, 13, 17, 24, 27]. Of these, the most relevant to compiling circuits is the “free XOR trick” given by Kolesnikov and Schneider [17]. This technique allows XOR gates to be evaluated without the need to garble them, which greatly reduces the amount of data that must be transferred and the CPU time required for both the generator and the evaluator. One basic way to take advantage of this technique is to choose subcircuits with fewer non-XOR gates; Schneider published a list of XOR-optimal circuits for even three-input functions [27].

Huang et al. noted that there is no need for the evaluator to wait for the generator to garble all gates in the circuit [13]. Once a gate is garbled, it can be sent to the evaluator, allowing generation and evaluation to occur in parallel. This technique is very important for large circuits, which can quickly become too large to store in RAM [18]. Our approach unifies this technique with the use of an optimizing compiler.

3 Bytecode

A common approach to compiler design is to translate a high level language into a sequence of instructions for a simple, abstract machine architecture; this is known as the *intermediate representation* or *bytecode*. Bytecode representations have the advantage of being machine-independent, thus allowing a compiler front-end to be used for multiple target architectures. Optimizations per-

formed on bytecode are machine independent as well; for example, dead code elimination is typically performed on bytecode, as removing dead code causes programs to run faster on all realistic machines.

For the purposes of this work, we focus on a commonly used bytecode abstraction, the *stack machine*. In this model, operands must be pushed onto an abstract stack, and operations involve popping operands off of the stack and pushing the result. In addition to the stack, a stack machine has RAM, which is accessed by instructions that pop an address off the stack. Instructions in a stack machine are partially ordered, and are divided into subroutines in which there is a total ordering. In addition to simple operations and operations that interact with RAM, a stack machine has operations that can modify the *program counter*, a pointer to the next instruction to be executed, either conditionally or unconditionally.

At a high level, our system translates bytecode programs for a stack machine into boolean circuits for SFE. At first glance, this would appear to be at least highly inefficient, if not impossible, because of the many ways such an input program could loop. We show, however, that imposing only a small set of restrictions on permissible sequences of instructions enables an efficient and practical translator, without significantly reducing the usability or expressive power of the high level language.

4 System Design

Our system divides the compiler into several stages, following a common compiler design. For testing, we used the LCC compiler front end to parse C source code and produce a bytecode intermediate representation (IR). Our back end performs optimizations and translates the bytecode into a description of a secure computation protocol using our new format. This representation greatly reduces the disk space requirements for large circuits compared to previous work, while still allowing optimizations to be done at the bit level. We wrote our compiler in Common Lisp, using the Steel Bank Common Lisp system.

4.1 Compact Representations of Boolean Circuits

In Fairplay and the systems that followed its design, the common pattern has been to represent Boolean circuits as adjacency lists, with each node in the graph being a gate. This introduces a scalability problem, as it requires storage proportional to the size of the circuit. Generating, optimizing, and storing circuits has been a bottleneck for previous compilers, even for relatively simple functions like RSA. Loading such large circuits into RAM

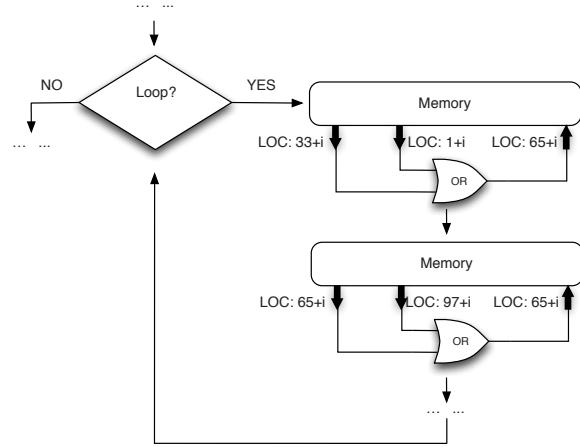


Figure 2: The high-level concept of the PCF design. It is not necessary to unroll loops at compile time, even to perform optimizations on the circuit. Instead, loops can be evaluated at runtime, with gates being computed on-the-fly, and loop indices being updated locally by each party. Wire values are stored in a table, with each gate specifying which two table entries should be used as inputs and where the output should be written; previous wire values in the table can be overwritten during this process, if they are no longer needed.

is a challenge, as even very high-end machines may not have enough RAM for relatively simple functions.

There have been some approaches to addressing this scalability problem presented in previous work. The KSS12 system reduced the RAM required for protocol executions by assigning each gate's output wire a reference count, allowing the memory used for a wire value to be deallocated once the gate is no longer needed. However, the compiler bottleneck was not solved in KSS12, as even computing the reference count required memory proportional to the size of the circuit. Even with the engineering improvements presented by Kreuter, shelat, and Shen, the KSS12 compiler was unable to compile circuits with more than a few billion gates, and required several days to compile their largest test cases [18].

The PAL system [23] also addresses memory requirements, by adding control structures to the circuit description, allowing parts of the description to be re-used. In the original presentation of PAL, however, a large circuit file would still be emitted in the Fairplay format when the secure protocol was run. An extension of this work presented by Mood [22] allowed the PAL description to be used directly at runtime, but this work sacrificed the ability to optimize circuits automatically.

Our system builds upon the PAL and KSS12 systems to solve the memory scalability problem without sacri-

ficing the ability to optimize circuits automatically. Two observations are key to our approach.

Our first observation is that it is possible to free the memory required for storing wire values without computing a reference count for the wire. In previous work, each wire in a circuit is assigned a unique global identifier, and gate input wires are specified in terms of these identifiers (output wires can be identified by the position of the gate in the gate list). Rather than using global identifiers, we observe that wire values are ephemeral, and only require a unique identity until their last use as the input to a gate.

We therefore maintain a table of “active” wire values, similar to KSS12, but change the gate description. In this format, wire values are identified by their index in the table, and gates specify the index of each input wire and an index for the output wire; in other words, a gate is a tuple $\langle t, i_1, i_2, o \rangle$, where t is a truth table, i_1, i_2 are the input wire indexes, and o is the output wire index. When a wire value is no longer needed, its index in the table can be safely used as an output wire for a gate.

Now, consider the following example of a circuit described in the above format, which accumulates the Boolean AND of seven wire values:

```
 $\langle AND_1, 1, 2, 0 \rangle$ 
 $\langle AND_2, 0, 3, 0 \rangle$ 
 $\langle AND_3, 0, 4, 0 \rangle$ 
 $\langle AND_4, 0, 5, 0 \rangle$ 
 $\langle AND_5, 0, 6, 0 \rangle$ 
 $\langle AND_6, 0, 7, 0 \rangle$ 
```

Our second observation is that circuits such as this can be described more compactly using a loop. This builds on our first observation, which allows wire values to be overwritten once they are no longer needed. A simple approach to allowing this would add a conditional branch operation to the description format. This is more general than the approach of PAL, which includes loops but allows only simple iteration. Additionally, it is necessary to allow the loop index to be used to specify the input or output wire index of the gates; as a general solution, we add support for indirection, allowing wire values to be copied.

This representation of Boolean circuits is a bytecode for a one-bit CPU, where the operations are the 16 possible two-arity Boolean gates, a conditional branch, and indirect copy. In our system, we also add instructions for function calls (which need not be inlined at compile time) and handling the parties’ inputs/outputs. When the secure protocol is run, a three-level logic is used for wire values: 0, 1, or \perp , where \perp represents an “unknown” value that depends on one of the party’s inputs. In the case of a Yao protocol, the \perp value is represented by a

garbled wire value. Conditional branches are not allowed to depend on \perp values, and indirection operations use a separate table of pointers that cannot be computed from \perp values (if such an indirection operation is required, it must be translated into a large multiplexer, as in previous work).

We refer to our circuit representation as the *Portable Circuit Format* or PCF. In addition to gates and branches, PCF includes support for copying wires indirectly, a function call stack, data stacks, and setting function parameters. These additional operations do not emit any gates and can therefore be viewed as “free” operations. PCF is modeled after the concept of PAL, but instead of using predefined sub-circuits for complex operations, a PCF file defines the sub-circuits for a given function to allow for circuit structure optimization. PCF includes lower level control structures compared to PAL, which allows for more general loop structures.

In Appendix A, we describe in detail the semantics of the PCF instructions. Example PCF files are available at the authors’ website.

4.2 Describing Functions for SFE

Most commonly used programming languages can describe processes that cannot be translated to SFE; for example, a program that does not terminate, or one which terminates after reading a specific input pattern. It is therefore necessary to impose some limitation on the descriptions of functions for SFE. In systems with domain specific languages, these limitations can be imposed by the grammar of the language, or can be enforced by taking advantage of particular features of the grammar. However, one goal of our system is to allow any programming language to be used to describe functionality for SFE, and so we cannot rely on the grammar of the language being used.

We make a compromise when it comes to restricting the inputs to our system. Unlike model checking systems [2], we impose no upper bound on loop iterations or on recursive function calls (other than the memory available for the call stack), and leave the responsibility of ensuring that programs terminate to the user. On the other hand, our system does forbid certain easily-detectable conditions that could result in infinite loops, such as unconditional backwards jumps, conditional backwards jumps that depend on input, and indirect function calls. These restrictions are similar to those imposed by the Fairplay and KSS12 systems [18, 21], but allow for more general iteration than incrementing the loop index by a constant. Although false positives, i.e., programs that terminate but which contain such constructs are possible, our hypothesis is that useful functions and typical compilers would not result in such instruction sequences, and

we observed no such functions in our experiments with LCC.

4.3 Algorithms for Translating Bytecode

Our compiler reads a bytecode representation of the function, which lacks the structure of higher-level descriptions and poses a unique challenge in circuit generation. As mentioned above, we do not impose any upper limit on loop iterations or the depth of the function call stack. Our approach to translation does not use any symbolic analysis of the function. Instead, we translate the bytecode into PCF, using conditional branches and function calls as needed and translating other instructions into lists of gates. For testing, we use the IR from the LCC compiler, which is based on the common stack machine model; we will use examples of this IR to illustrate our design, but note that none of our techniques strictly require a stack machine model or any particular features of the LCC bytecode.

In our compiler, we divide bytecode instructions into three classes:

Normal Instructions which have exactly one successor and which can be represented by a simple circuit. Examples of such instructions are arithmetic and bitwise logic operations, operations that push data onto the stack or move data to memory, etc.

Jump Instructions that result in an unconditional control flow switch to a specific label. This does not include function calls, which we represent directly in PCF. Such instructions are usually used for if/else constructs or preceding the entry to a loop.

Conditional Instructions that result in control flow switching to either a label or the subsequent instruction, depending on the result of some conditional statement. Examples include arithmetic comparisons.

In the stack machine model, all operands and the results of operations are pushed onto a global stack. For “normal” instructions, the translation procedure is straightforward: the operands are popped off the stack and assigned temporary wires, the subcircuit for the operation is connected to these wires, and the output of the operation is pushed onto the stack. “Jump” instructions appear, at first, to be equally straightforward, but actually require special care as we describe below.

“Conditional” instructions present a challenge. Conditional jumps whose targets precede the jump are assumed to be loop constructs, and are translated directly into PCF branch instructions. All other conditional jumps require the creation of multiplexers in the circuit to deal with

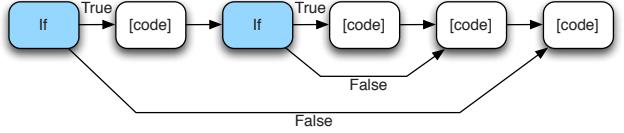


Figure 3: Nested if statements, which can be handled using the stack-based algorithm.

conditional assignments. Therefore, the branch targets must be tracked to ensure that the appropriate condition wires are used to control those multiplexers.

In the Fairplay and KSS12 compilers, the condition wire for an “if” statement is pushed onto a stack along with a “scope” that is used to track the values (wire assignments) of variables. When a conditional block is closed, the condition wire at the top of the stack is used to multiplex the value of all the variables in the scope at the top with the values from the scope second to the top, and then the stack is popped. This procedure relies on the grammar of “if/else” constructs, which ensures that conditional blocks can be arranged as a tree. An example of this type of “if/else” construct is in Figure 3. In a bytecode representation, however, it is possible for conditional blocks to “overlap” with each other without being nested.

In the sequence shown in Figure 4, the first branch’s target *precedes* the second branch’s target, and indirect loads and assignments exist in the overlapping region of these two branches. The control flow of such an overlap is given in Figure 5. A stack is no longer sufficient in this case, as the top of the stack will not correspond to the appropriate branch when the next branch target is encountered. Such instruction sequences are not uncommon in the code generated by production compilers, as they are a convenient way to generate code for “else” blocks and ternary operators.

To handle such sequences, we use a novel algorithm based on a priority queue rather than a stack, and we maintain a global condition wire that is modified as branches and branch targets are reached. When a branch instruction is reached, the global condition wire is updated by logically ANDing the branch condition with the global condition wire. The priority queue is updated with the branch condition and a scope, as in the stack-based algorithm; the priority is the target, with lower targets having higher priority. When an assignment is performed, the scope at the top of the priority queue is updated with the value being assigned, the location being assigned to, the old value, and a copy of the global condition wire. When a branch target is reached, multiplexers are emitted for each assignment recorded in the scope at the top of the priority queue, using the copy of the global condition wire that was recorded. After the

```

EQU4 A
INDIRI4 16
EQU4 B
INDIRI4 24
LABELV A
ASGNI4
LABELV B
ASGNI4

```

Figure 4: A bytecode sequence where overlapping conditional blocks are not nested; note that the target of the first branch, “A,” precedes the target of the second branch, “B.”

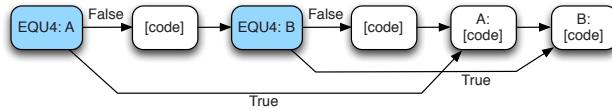


Figure 5: A control flow with overlapping conditional blocks.

multiplexers are emitted, the global condition wire is updated by ORing the *inverse* of the condition wire at the top of the priority queue, and then the top is removed.

Unconditional jumps are only allowed in the forward direction, i.e., only if the jump precedes its target. When such instructions are encountered, they are translated into conditional branches whose condition wire is the inverse of the conjunction of the condition wires of all enclosing branches. In the case of a jump that is not in any conditional block, the condition wire is set to false; this does not necessarily mean that subsequent assignments will not occur, as the multiplexers for these assignments will be emitted and will depend on a global control line that may be updated as part of a loop construct. The optimizer is responsible for determining whether such assignments can occur, and will rewrite the multiplexers as direct assignments when possible.

Finally, it is possible that the operand stack will have changed in the fall-through path of a conditional jump. In that case, the stack itself must be multiplexed. For simplicity, we require that the depth of the stack not change in a fall-through path. We did not observe any such changes to the stack in our experiments with LCC.

4.4 Optimization

One of the shortcomings of the KSS12 system was the amount of time and memory required to perform optimizations on the computed circuit. In our system, optimization is performed before loops are unrolled but after the functionality is translated into a PCF representation. This allows optimizations to be performed on a smaller

representation, but increases the complexity of the optimization process somewhat.

The KSS12 compiler bases its optimization on a rudimentary dataflow analysis, but without any conditional branches or loops, and with single assignments to each wire. In our system, loops are not eliminated and wires may be overwritten, but conditional branches are eliminated. As in KSS12, we use an approach based on dataflow analysis, but we must make multiple passes to find a fixed point solution to the dataflow equations. Our dataflow equations take advantage of the logical rules of each gate, allowing more gates to be identified for elimination than the textbook equations identify.

We perform our dataflow analysis on individual PCF instructions, which allows us to remove single gates even where entire bytecode instructions could not be removed, but which carries the cost of somewhat longer compilation time, on the order of minutes for the experiments we ran. Currently, our framework only performs optimization within individual functions, without any interprocedural analysis. Compile times in our system can be reduced by splitting a large procedure into several smaller procedures.

Optimization	128 mult.	5x5 matrix	256 RSA
None	707,244	260,000	904,171,008
Const. Prop.	296,960	198,000	651,504,495
Dead Elim.	700,096	255,875	883,307,712
Both	260,073	131,875	573,156,735

Table 1: Effects of constant propagation and dead code elimination on circuit size, measured with simulator that performs no simplification rules. For each function, the number of non-XOR gates are given for all combinations of optimizations enabled.

4.4.1 Constant Propagation

The constant propagation framework we use is straightforward, similar to the methods used in typical compilers. However, for some gates, simplification rules can result in constants being computed even when the inputs to a gate are not constant; for example, XORing a variable with itself. The transfer function we use is augmented with a check against logic simplification rules to account for this situation, but remains monotonic and so convergence is still guaranteed.

4.4.2 Dead Gate Removal

The last step of our optimizer is to remove gates whose output wires are never used. This is a standard bit vector dataflow problem that requires little tailoring for our system. As is common in compilers, performing this step

Function	With	Without	Ratio
16384-bit Comp.	32,228	49,314	65%
128-bit Sum	345	508	67%
256-bit Sum	721	1,016	70%
1024-bit Sum	2,977	4,064	73%
128-bit Mult.	76,574	260,073	20%
256-bit Mult.	300,634	1,032,416	20%
1024-bit Mult.	8,301,962	19,209,120	21%

Table 2: Non-XOR gates in circuits computed by the interpreter with and without the application of simplification rules by the runtime system.

last yields the best results, as large numbers of gates become dead following earlier optimizations.

4.5 Externally-Defined Functions

Some functionality is difficult to describe well in bytecode formats. For example, the graph isomorphism experiment presented in Section 6 uses AES as a PRNG building block, but the best known description of the AES S-box is given at the bit-level [4], whereas the smallest width operation supported by LCC is a single byte. To compensate for this difficulty, we allow users to specify functions with the same language used internally to translate bytecode operations into circuits; an example of this language is shown in Section 5.1. This allows for possible combinations of our compiler with other circuit generation and optimization tools.

4.6 PCF Interpreter

To use a PCF description of a circuit in a secure protocol, an interpreter is needed. The interpreter simulates the execution of the PCF file for a single-bit machine, emitting gates as needed for the protocol. Loops are not explicitly unrolled; instead, PCF branch instructions are conditionally followed, based on the logic value of some wire, and each wire identifier is treated as an address in memory. This is where the requirement that loop bounds be independent of both parties’ inputs is ultimately enforced: the interpreter cannot determine whether or not to take a branch if it cannot determine the condition wire’s value.

For testing purposes, we wrote two PCF interpreters: one in C, which is packaged as a reusable library, and one in Java that was used for tests on smartphones. The C library can be used as a simulator or for full protocol execution. As a simulator it simply evaluates the PCF file without any garbling to measure the size of the circuit that would have been garbled in a real protocol. This interpreter was used for the LAN tests, using an updated version of the KSS12 protocol. The Java interpreter was

Function	With (s)	Without (s)
16384-bit Comp.	$4.41 \pm 0.3\%$	$4.44 \pm 0.3\%$
128-bit Sum	$0.0581 \pm 0.3\%$	$0.060 \pm 2\%$
256-bit Sum	$0.103 \pm 0.3\%$	$0.105 \pm 0.3\%$
1024-bit Sum	$0.365 \pm 0.3\%$	$0.367 \pm 0.2\%$
128-bit Mult.	$0.892 \pm 0.1\%$	$0.894 \pm 0.1\%$
256-bit Mult.	$3.02 \pm 0.1\%$	$3.04 \pm 0.1\%$
1024-bit Mult.	$39.7 \pm 0.2\%$	$39.9 \pm 0.06\%$

Table 3: Simulator time with simplification rules versus without, using the C interpreter. Times are averaged over 50 samples, with 95% confidence intervals, measured using the *time* function implemented by SBCL.

incorporated into the HEKM system for the smartphone experiments, and can also be used in a simulator mode.

4.7 Threat Model

The PCF system treats the underlying secure computation protocol as a black box, without making any assumptions about the threat model. In Section 6, we present running times for smaller circuits in the malicious model version of the KSS12 protocol. This malicious model implementation simply invokes multiple copies of the same PCF interpreter used for the semi-honest version, one for each copy of the circuit needed in the protocol.

4.8 Runtime Optimization

Some optimizations cannot be performed without unrolling loops, and so we defer these optimizations until the PCF program is interpreted. As an example, logic simplification rules that eliminate gates whose output values depend on no more than one of their input wires can only be partially applied at compile time, as some potential applications of these rules might only be possible for some iterations of a loop. While it is possible to compute this information at compile time, in the general case this would involve storing information about each gate for every iteration of every loop, which would be as expensive as unrolling all loops at compile time.

A side effect of applying such logic simplification rules is copy propagation. A gate that always takes on the same value as one of its inputs is equivalent to a copy operation. The application of logic simplification rules to such a gate results in the interpreter simply copying the value of the input wire to the output wire, without emitting any gate. As there is little overhead resulting from the application of simplification rules at runtime, we are able to reduce compile times further by not performing this optimization at compile time.

Function	This Work	KSS12	HFKV
16384 Comp.	32,229	49,149	-
RSA 256	235,925,023	332,085,981	-
Hamming 160	880	-	3,003
Hamming 1600	9,625	-	30,318
3x3 Matrix	27,369	160,949	47,871
5x5 Matrix	127,225	746,177	221,625
8x8 Matrix	522,304	3,058,754	907,776
16x16 Matrix	4,186,368	24,502,530	7,262,208

Table 4: Comparisons between our compiler’s output and the output of the KSS12 and Holzer et al. (HFKV) compilers, in terms of non-XOR gates.

For each gate, the interpreter checks if the gate’s value can be statically determined, i.e., if its output value does not rely on either party’s input bits. This is critical, as some of the gates in a PCF file are used for control flow, e.g., to increment a loop index. Additionally, logic simplification rules are applied where possible in the interpreter. This allows the interpreter to not emit gates that follow an input or which have static outputs even when their inputs cannot be statically determined. As shown in Table 2, we observed cases where up to 80% of the gates could be removed in this manner. Even in a simulator that performs no garbling, applying this runtime optimization not only shows no performance overhead, but actually a very slight performance gain, as shown in Table 3. The slight performance gain is a result of the transfer of control that occurs when a gate is emitted, which has a small but non-trivial cost in the simulator. In a garbled circuit protocol, this cost would be even higher, because of the time spent garbling gates.

5 Portability

5.1 Portability Between Bytecodes

Our compiler can be given a description of how to translate bytecode instructions into boolean circuits using a special internal language. An example, for the LCC instruction “ADDU,” is shown in Figure 6. The first line is specific to LCC, and would need to be modified for use with other front-ends. The second line assumes a stack machine model: this instruction reads two instructions from the stack. Following that is the body of the translation rule, which can be used in general to describe circuit components and how the input variables should be connected to those components.

The description follows an abstraction similar to VM-Crypt, in which a unit gadget is “chained” to create a larger gadget. It is possible to create chains of chains, e.g., for a shift-and-add multiplier as well. For more complex operations, Lisp source code can be embedded,

```
(``ADDU'' nil second normal nil nil
(two-stack-arg (x y) (var var)
(chain [o1 = i1 + i2 + i3,
        o2 = i1 + (i1 + i2) * (i1 + i3)]
(o2 -> i3
  x -> i1
  y -> i2
  o1 -> stack)
(0 -> i3))))
```

Figure 6: Code used in our compiler to map the bytecode instruction for unsigned integer addition to the subcircuit for that operation.

which can interact directly with the compiler’s internal data structures.

5.2 Portability Between SFE Systems

Both the PCF compiler and the interpreter can treat the underlying secure computation system as a black box. Switching between secure computation systems, therefore, requires work only at the “back end” of the interpreter, where gates are emitted. We envision two possible approaches to this, both of which we implemented for our tests:

1. A single function should be called when a gate should be used in the secure computation protocol. The Java implementation of PCF uses this approach, with the HEKM system.
2. Gates should be generated as if they are being read from a file, with the secure computation system calling a function. The secure computation system may need to provide “callback” functions to the PCF interpreter for copying protocol-specific data between wires. The C implementation we tested uses this abstraction for the KSS12 system.

6 Evaluation

We compiled a variety of functions to test our compiler, optimizer, and PCF interpreter. For each circuit, we tested the performance of the KSS12 system on a LAN, described below. For the KSS12 timings, we averaged the runtime for 50 runs, alternating which computer acted as the generator and which as the evaluator to account for slight configuration differences between the systems. Compiler timings are based on 50 runs of the compiler on a desktop PC with an Intel Xeon 5560 processor, 8GB of RAM, a 7200 RPM hard disk, Scientific Linux 6.3 (kernel version 2.6.32, SBCL version 1.0.38).

Function	Total Gates	non-XOR Gates	Compile Time (s)	Simulator Time (s)
16384-bit Comp.	97,733	32,229	3.40 ± 4%	4.40 ± 0.2%
Hamming 160	4,368	880	9.81 ± 1%	0.0810 ± 0.3%
Hamming 1600	32,912	6,375	11.0 ± 0.4%	0.52 ± 8%
Hamming 16000	389,312	97,175	10.8 ± 0.2%	4.83 ± 0.5%
128-bit Sum	1,443	345	4.70 ± 3%	0.0433 ± 0.4%
256-bit Sum	2,951	721	4.60 ± 3%	0.0732 ± 0.4%
1024-bit Sum	11,999	2,977	4.60 ± 3%	0.250 ± 0.5%
64-bit Mult.	105,880	24,766	71.7 ± 0.2%	0.332 ± 0.4%
128-bit Mult.	423,064	100,250	74.9 ± 0.1%	0.903 ± 0.3%
256-bit Mult.	1,659,808	400,210	79.5 ± 0.9%	3.07 ± 0.2%
1024-bit Mult.	25,592,368	6,371,746	74.0 ± 0.2%	40.9 ± 0.4%
256-bit RSA	673,105,990	235,925,023	381. ± 0.2%	980. ± 0.3%
512-bit RSA	5,397,821,470	1,916,813,808	350. ± 0.2%	7,330 ± 0.2%
1024-bit RSA	42,151,698,718	15,149,856,895	564. ± 0.2%	56,000 ± 0.3%
3x3 Matrix Mult.	92,961	27,369	306. ± 1%	0.256 ± 0.5%
5x5 Matrix Mult.	433,475	127,225	343. ± 0.7%	0.94 ± 2%
8x8 Matrix Mult.	1,782,656	522,304	109. ± 0.1%	3.14 ± 0.3%
16x16 Matrix Mult.	14,308,864	4,186,368	109. ± 0.1%	23.7 ± 0.3%
4-Node Graph Iso.	482,391	97,819	684. ± 0.2%	3.63 ± 0.5%
16-Node Graph Iso.	10,908,749	4,112,135	1040 ± 0.1%	47.0 ± 0.1%

Table 5: Summary of circuit sizes for various functions and the time required to compile and interpret the PCF files in a protocol simulator. Times are averaged over 50 samples, with 95% confidence intervals, except for RSA-1024 simulator time, which is averaged over 8 samples. Run times were measured using the *time* function implemented in SBCL.

Source code for our compiler, our test systems, and our test functions is available at the authors’ website.

6.1 Effect of Array Sizes on Timing

Some changes in compile time can be observed as some of the functions grow larger. The dataflow analysis deals with certain pointer operations by traversing the entire local variable space of the function and all global memory, which in functions with large local arrays or programs with large global arrays is costly as it increases the number of wires that optimizer must analyze. Reducing this cost is an ongoing engineering effort.

6.2 Experiments

We compiled and executed the circuits described below to evaluate our compiler and representation. Several of these circuits were tested in other systems; we present the non-XOR gate counts of the circuits generated by our compiler and other work in Table 4. The sizes, compile times, and interpreter times required for these circuits are listed in Table 5. By comparison, we show compile times and circuit sizes using the KSS12 and HFKV compilers in Table 6. As expected, the PCF compiler outperforms

these previous compilers as the size of the circuits grow, due to the improved scalability of the system.

Arbitrary-Width Millionaire’s Problem As a simple sanity check for our system, we tested an arbitrary-width function for the millionaire’s problem; this can be viewed as a string comparison function on 32 bit characters. It outputs a 1 to the party which has the larger input. We found that for this simple function, our performance was only slightly better than the performance of the KSS12 compiler on the same circuit.

Matrix Multiplication To compare our system with the work of Holzer et al. [12], we duplicated some of their experiments, beginning with matrix multiplication on 32-bit integers. We found that our system performed favorably, particularly due to the optimizations our compiler and PCF interpreter perform. On average, our system generated circuits that are 60% smaller. We tested matrices of 3x3, 5x5, 8x8, and 16x16, with 32 bit integer elements.

Hamming Distance Here, we duplicate the Hamming distance experiment from Holzer et al. [12]. Again, we found that our system generated substantially smaller circuits. We tested input sizes of 160, 1600, and 16000 bits.

Integer Sum We implemented a basic arbitrary-width integer addition function, using ripple-carry addition. No

Function	HFKV			KSS12		
	Total Gates	non-XOR gates	Time (s)	Total Gates	non-XOR gates	Time (s)
16384-bit Comp.	330,784	131,103	105. \pm 0.1%	98,303	49,154	4.66 \pm 0.5%
3x3 Matrix Mult.	172,315	47,871	2.2 \pm 4%	424,748	160,949	10.5 \pm 0.5%
5x5 Matrix Mult.	797,751	221,625	8.40 \pm 0.3%	1,968,452	746,177	48.2 \pm 0.2%
8x8 Matrix Mult.	3,267,585	907,776	59.4 \pm 0.3%	8,067,458	3,058,754	210 \pm 2%
16x16 Matrix Mult.	26,140,673	7,262,208	2,600 \pm 7%	64,570,969	24,502,530	2,200 \pm 1%
32-bit Mult.	65,121	26,624	6.43 \pm 0.3%	15,935	5,983	0.55 \pm 5%
64-bit Mult.	321,665	126,529	71.4 \pm 0.3%	64,639	24,384	1.6 \pm 2%
128-bit Mult.	1,409,025	546,182	999. \pm 0.1%	260,351	97,663	6.10 \pm 0.6%
256-bit Mult.	5,880,833	2,264,860	16,000 \pm 2%	1,044,991	391,935	24.5 \pm 0.2%
512-bit Mult.	-	-	-	4,187,135	1,570,303	105. \pm 0.2%
1024-bit Mult.	-	-	-	16,763,518	6,286,335	430. \pm 0.3%

Table 6: Times of HFKV and KSS12 compilers with circuit sizes. The Mult. program uses a Shift-Add implementation. All times are averaged over 50 samples with the exception of the HFKV 256-bit multiplication, which was run for 10 samples; times are given with 95% confidence intervals.

array references are needed, and so our compiler easily handles this function even for very large input sizes. We tested input sizes of 128, 256, and 1024 bits.

Integer Multiplication Building on the integer addition function, we tested an integer multiplication function that uses the textbook shift-and-add algorithm. Unlike the integer sum and hamming distance functions, the multiplication function requires arrays for both input and output, which slows the compiler down as the problem size grows. We tested bit sizes of 64, 128, 256, and 1024.

RSA (Modular Exponentiation) In the KSS12 system [18], it was possible to compile an RSA circuit for toy problem sizes, and it took over 24 hours to compile a circuit for 256-bit RSA. This lengthy compile time and large memory requirement stems from the fact that all loops are unrolled before any optimization is performed, resulting in a very large intermediate representation to be analyzed. As a demonstration of the improvement our approach represents, we compiled not only toy RSA sizes, but also an RSA-1024 circuit, using only modest computational resources. We tested bit sizes of 256, 512, and 1024.

Graph Isomorphism We created a program that allows two parties to jointly prove the zero knowledge proof of knowledge for graph isomorphism, first presented by Goldreich et al. [9]. In Goldreich et al.’s proof system, the prover has secret knowledge of an isomorphism between two graphs, g_1 and g_2 . To prove this, the prover sends the verifier a random graph g_3 that is isomorphic to g_1 and g_2 , and the verifier will then choose to learn either the $g_1 \rightarrow g_3$ isomorphism or the $g_2 \rightarrow g_3$ isomorphism. We modify this protocol so that Alice and Bob must jointly act as the prover; each is given shares of an isomorphism between graphs g_1 and g_2 , and will use the online protocol to compute g_3 and shares of the two isomorphisms.

Our implementation works as follows: the program takes in XOR shares of the isomorphism between g_1 and g_2 and a random seed from both participants. It also takes the adjacency matrix representation of g_1 as input by a single party. The program XORs the shares together to create the $g_1 \rightarrow g_2$ isomorphism. The program then creates a random isomorphism from $g_1 \rightarrow g_3$ using AES as the PRNG (to reduce the input sizes and thus the OT costs), which effectively also creates g_3 .

Once the random isomorphism $g_1 \rightarrow g_3$ is created, the original isomorphism, $g_1 \rightarrow g_2$, is inverted to get an isomorphism from $g_2 \rightarrow g_1$. Then the two isomorphisms are “followed” in a chain to get the g_2 to g_3 isomorphism, i.e., for the i^{th} instance in the isomorphic matrix, $iso_{2 \rightarrow 3}[i] = iso_{1 \rightarrow 3}[iso_{2 \rightarrow 1}[i]]$. The program outputs shares of both the isomorphism from g_1 to g_3 and the isomorphism from g_2 to g_3 to both parties.

An adjacency matrix of g_3 is also an output for the party which input the adjacency matrix g_1 . This is calculated by using g_1 and the $g_1 \rightarrow g_3$ isomorphism.

6.3 Online Running Times

To test the online performance of our new format, we modified the KSS12 protocol to use the PCF interpreter. Two sets of tests were run: one between two computers with similar specifications on the University of Virginia LAN, a busy 100 megabit Ethernet network, and one between two smartphones communicating over a wifi network.

For the LAN experiments, we used two computers running ScientificLinux 6.3, a four core Intel Xeon E5506 2.13GHz CPU, and 8GB of RAM. No time limit on computation was imposed on these machines, so we were able to run the RSA-1024 circuit, which requires a little less than two days. To compensate for slight con-

Function	CPU (s)	Network (s)	CPU (s)	Network (s)
	Generator		Evaluator	
16384-bit Comp.	99.8 ± 0.2%	5.63 ± 0.6%	26.0 ± 0.6%	79.4 ± 0.2%
Hamming 1600	9.13 ± 0.4%	0.64 ± 4%	2.9 ± 4%	6.87 ± 2%
Hamming 16000	91.2 ± 0.2%	5.67 ± 0.7%	28. ± 3%	69. ± 2%
64-bit Mult.	0.749 ± 0.3%	0.158 ± 0.7%	0.409 ± 0.3%	0.494 ± 0.6%
128-bit Mult.	2.04 ± 0.3%	0.52 ± 1%	1.25 ± 0.2%	1.31 ± 0.6%
256-bit Mult.	5.74 ± 0.5%	1.2 ± 2%	4.2 ± 2%	2.7 ± 3%
1024-bit Mult.	72.7 ± 0.2%	28. ± 4%	60. ± 2%	40. ± 3%
256-bit RSA	1940 ± 0.2%	767. ± 0.7%	1620 ± 2%	1080 ± 3%
1024-bit RSA	$1.15 \times 10^5 \pm 0.5\%$	$4.4 \times 10^4 \pm 4\%$	$9.5 \times 10^4 \pm 5\%$	$6.5 \times 10^4 \pm 7\%$
3x3 Matrix Mult.	5.33 ± 0.4%	0.403 ± 0.6%	1.45 ± 0.8%	4.28 ± 0.6%
5x5 Matrix Mult.	24.4 ± 0.2%	1.81 ± 0.4%	6.75 ± 0.9%	19.5 ± 0.4%
8x8 Matrix Mult.	100. ± 0.2%	7.39 ± 0.4%	26.8 ± 0.7%	81.1 ± 0.3%
4-node ISO	10.1 ± 0.1%	1.05 ± 0.7%	4.96 ± 0.3%	6.15 ± 0.4%
16-node ISO	116. ± 0.2%	15.7 ± 0.6%	71.6 ± 0.3%	60.3 ± 0.6%

Table 7: Total running time, including PCF operations and protocol operations such as oblivious transfer, for online protocols using the PCF interpreter and the KSS12 two party computation system, on two computers communicating over the University of Virginia LAN. With the exception of RSA-1024, all times are averaged over 50 samples; RSA-1024 is averaged over 8 samples. Running time is divided into time spent on computation and time spent on network operations (including blocking).

figuration differences between the two systems, we alternated between each machine acting as the generator and acting as the evaluator.

We give the results of this experiment in Table 7. We note that while the simulator times given in Table 5 are more than half the CPU time measured, they are also on par with the time spent waiting on the network. Non-blocking I/O or a background thread for the PCF interpreter may improve performance somewhat, which is an ongoing engineering task in our implementation.

6.4 Malicious Model Tests

The PCF system is not limited to the semi-honest model. We give preliminary results in the malicious model version of KSS12. These experiments were run on the same test systems as above, using two cores for each party. We present our results in Table 9. The increased running times are expected, as we used only two cores per party. In the case of 16384-bit comparison, the increase is very dramatic, due to the large amount of time spent on oblivious transfer (as both parties have long inputs).

6.5 Phone Execution

We created a PCF interpreter for use with the HEKM execution system and ported it to the Android environment. We then ran it on two Galaxy Nexus phones where one

phone was the generator and another phone was the evaluator. These phones have dual core 1.2Ghz processors and were linked over Wi-Fi using an Apple Airport.

6.6 Phone Trials

As seen in Table 8, we were able to run the smaller programs directly on two phones. Since the interpreter executes slower on a phone and what would have taken a week of LAN trials would have taken years of phone time, we did not complete trials of the larger programs. Not all of the programs had output for the generator, allowing the generator to finish before the evaluator. This leads to a noticeable difference in total running time between the two parties.

Mood’s work on designing SFE applications for mobile devices [22] found that allocation and deallocation was a bottleneck to circuit execution. This issue was addressed by substituting the standard *BigInteger* type for a custom class that reduced the amount of allocation required for numeric operations, resulting in a four-fold improvement in execution time. The lack of this optimization in our mobile phone experiments may contribute to the reduced performance that we observed.

In future work, we will port the C interpreter and KSS12 system to Android and run the experiment with that execution system. Since overhead appears to be tied to Android’s Dalvik Virtual Machine (DVM), running programs natively should reduce overhead and hence re-

Function	CPU (s)	Network (s)	CPU (s)	Network (s)
	Generator		Evaluator	
16384-bit Comp.	163. \pm 0.5%	12. \pm 3%	142. \pm 0.5%	68. \pm 1%
128-bit Sum	5.8 \pm 8.2%	1. \pm 30%	5.6 \pm 8%	3. \pm 20%
256-bit Sum	7.3 \pm 5.0%	1. \pm 30%	6. \pm 5%	4. \pm 20%
1024-bit Sum	16. \pm 3.1%	2. \pm 20%	16. \pm 3%	6.4 \pm 7%
64-bit Mult.	63.3 \pm 0.5%	1. \pm 10%	66.3 \pm 0.6%	5. \pm 10%
128-bit Mult.	257. \pm 0.2%	3.8 \pm 5%	280. \pm 0.3%	12. \pm 6%
3x3 Matrix Mult.	76.9 \pm 0.4%	12. \pm 2%	82.0 \pm 0.5%	8.5 \pm 4%
5x5 Matrix Mult.	352. \pm 0.3%	49. \pm 2%	371. \pm 0.3%	32. \pm 4%
8x8 Matrix Mult.	1,588. \pm 0.1%	82. \pm 3%	1,550. \pm 0.1%	120. \pm 1%

Table 8: Execution results from the phone interpreter using the HEKM execution system on two Galaxy Nexus phones. Times are averages of 50 samples, with 95% confidence intervals.

Function	CPU (s)	Network (s)	CPU (s)	Network (s)
	Generator		Evaluator	
16384-bit comp.	3900 \pm 3%	76 \pm 4%	2820 \pm 2%	1200 \pm 10%
128-bit sum	23. \pm 2%	21 \pm 2%	33.3 \pm 0.5%	11.2 \pm 0.2%
256-bit sum	63.0 \pm 0.4%	10 \pm 20%	49. \pm 6%	27. \pm 4%
1024-bit sum	260 \pm 10%	16 \pm 6%	187. \pm 2%	100 \pm 40%
128-bit mult.	192. \pm 0.3%	47.2 \pm 0.6%	168. \pm 0.4%	70.1 \pm 1%
256-bit mult.	637. \pm 0.5%	160 \pm 1%	577. \pm 0.3%	210 \pm 2%

Table 9: Online running time in the malicious model for several circuits. Times are averaged over 50 samples, with 95% confidence intervals.

duce the performance differential between the phone and PC environments. Additionally, the KSS12 system uses more efficient cryptographic primitives, potentially further improving performance.

7 Related Work

Compiler approaches to secure two-party computation have attracted significant attention in recent years. The TASTY system presented by Henecka et al. [11] combines garbled circuit approaches with homomorphic encryption, and includes a compiler that emits circuits that can be used in both models. As with Fairplay and KSS12, TASTY requires functions to be described in a domain-specific language. The TASTY compiler performs optimizations on the abstract syntax tree for the function being compiled. Kruger et al. developed an ordered BDD compiler to test the performance of their system relative to Fairplay [19]. Mood et al. focused on compiling secure functions on mobile devices with the PALC system, which involved a modification to the Fairplay compiler [23].

Recently, a compiler approach based on bounded model checking was present by Holzer et al. [12]. In that

work, the CBMC system [5] was used to construct circuits, which were then rewritten to have fewer non-XOR gates. This approach had several advantages over previous approaches, most prominent being that functions could be described in the widely used C programming language, and that the use of CBMC allows for more advanced software engineering techniques to be applied to secure computation protocols. Like KSS12, however, this approach unrolls all loops (up to some fixed number of iterations), and converts a high level description directly to a boolean circuit which must then be optimized.

In addition to SFE, work on efficient compilers for proof systems has also been presented. Almeida et al. developed a zero-knowledge proof of knowledge compiler for Σ -protocols, which converts a protocol specification given in a domain-specific language into a program for the prover and the verifier to run [1]. Setty et al. presented a system for verifiable computation that uses a modification of the Fairplay compiler, which computes a system of quadratic constraints instead of boolean circuits, and emits executables for the prover and verifier [28, 29]. Our system is somewhat similar to these approaches, in that the circuit representation we present can be viewed as a program that is executed by the par-

ties in the SFE system; however, our approach is unique in its handling of control flow and iterative constructs.

Closely related to our work is the Sharemind system [3, 14], which uses secure computation as a building block for privacy-preserving distributed applications. As in our approach, the circuits used in the secure computation portions of Sharemind are not fully unrolled until the protocol is actually run. Functions in Sharemind are described using a domain-specific language called SecreC. Although there has been work on static analysis for SecreC [26], the SecreC compiler does not perform automatic optimizations. By contrast, our approach is focused on allowing circuit optimizations at the bit-level to occur without having to unroll an entire circuit.

Kerschbaum has presented work on automatically optimizing secure computation at the protocol level, with an approach based on term and expression rewriting [15, 16]. This approach is based on maximizing the use of offline computation by inferring what each party can compute without knowledge of the other party’s input, and does not treat the underlying secure computation primitives as a black box. It therefore requires additional work to remain secure in the malicious model. Our techniques could conceivably be combined with Kerschbaum’s to reduce the overhead of online components.

8 Future Work

Our compiler can conceivably read any bytecode representation as input; one immediate future direction is to write translations for the instructions of another bytecode format, such as LLVM or the JVM, which would allow functions to be expressed in a broader range of languages. Additionally, we believe that our techniques could be combined with Sharemind, by having our compiler read the bytecode for the Sharemind VM and compute optimized PCF files for cases where garbled circuit computations are used in a Sharemind protocol.

The PCF format does not convey high-level information about data operations or types. Such information may further reduce the size of the circuits that are computed. Static analysis of such information by compilers has been widely studied, and it is possible that our compiler could be extended to support further reductions in the sizes of circuits emitted by the PCF interpreter. High-level information about data structures could also be used to improve the generation of circuits prior to optimization, using techniques recently presented by Evans and Zahur [6].

Our system and techniques can likely be generalized to the multiparty case, and to other representations of functions, such as arithmetic circuits. This would require significant changes to the optimization strategies and goals in our compiler, but fewer changes would be necessary

for the PCF interpreter. Similar modifications to support homomorphic encryption systems are also possible.

9 Conclusion

We have presented an approach to compiling and storing circuits for secure computation systems that requires substantially lower computational resources than previous approaches. Empirical evidence of the improvement and utility of our approach is given, using a variety of functions with different circuit sizes and control flow structures. Additionally, we have presented a compiler for secure computation that reads bytecode as an input, rather than a domain-specific language, and have explored the challenges associated with such an approach. We also presented interpreters, which evaluate our new language on both PCs and phones.

The code for the compiler, PCF interpreters, and test cases will be available on the authors’ website.

Acknowledgments We would like to thank Elaine Shi for her helpful advice. We also thank Chih-hao Shen for his help with porting KSS12 to use PCF. This material is based on research sponsored by the Defense Advanced Research Projects Agency (DARPA) and the Air Force Research Laboratory (AFRL) under contract 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] 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 15th European conference on Research in computer security, ESORICS’10*, pages 151–167, Berlin, Heidelberg, 2010. Springer-Verlag.
- [2] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu. Symbolic Model Checking without BDDs. In *Proceedings of the 5th International Conference on Tools and Algorithms for Construction and Analysis of Systems, TACAS ’99*, pages 193–207, London, UK, UK, 1999. Springer-Verlag.
- [3] D. Bogdanov, S. Laur, and J. Willemson. Sharemind: A Framework for Fast Privacy-Preserving Computations. In *Proceedings of the 13th European Symposium on Research in Computer Security - ESORICS’08*, 2008.
- [4] J. Boyar and R. Peralta. A New Combinational Logic Minimization Technique with Applications to Cryptology. In P. Festa, editor, *Experimental Algorithms*, volume 6049 of *Lecture Notes in Computer Science*, pages 178–189. Springer Berlin / Heidelberg, 2010.

- [5] E. Clarke, D. Kroening, and F. Lerda. A Tool for Checking ANSI-C Programs. In K. Jensen and A. Podelski, editors, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2004)*, volume 2988 of *Lecture Notes in Computer Science*, pages 168–176. Springer, 2004.
- [6] D. Evans and S. Zahur. Circuit structures for improving efficiency of security and privacy tools. In *IEEE Symposium on Security and Privacy (to appear)*, 2013.
- [7] S. Even, O. Goldreich, and A. Lempel. A randomized protocol for signing contracts. *Commun. ACM*, 28(6):637–647, June 1985.
- [8] C. W. Fraser and D. R. Hanson. *A Retargetable C Compiler: Design and Implementation*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.
- [9] O. Goldreich, S. Micali, and A. Wigderson. Proofs that yield nothing but their validity or all languages in np have zero-knowledge proof systems. *J. ACM*, 38(3):690–728, July 1991.
- [10] V. Goyal, P. Mohassel, and A. Smith. Efficient Two Party and Multi Party Computation Against Covert Adversaries. In *Proceedings of 27th annual international conference on Advances in cryptology, EUROCRYPT’08*, pages 289–306, Berlin, Heidelberg, 2008. Springer-Verlag.
- [11] W. Henecka, S. Kögl, A.-R. Sadeghi, T. Schneider, and I. Wehrenberg. TASTY: Tool for Automating Secure Two-partY computations. In *ACM Conference on Computer and Communications Security*, 2010.
- [12] A. Holzer, M. Franz, S. Katzenbeisser, and H. Veith. Secure Two-Party computations in ANSI C. In *Proceedings of the 2012 ACM conference on Computer and communications security, CCS ’12*, pages 772–783, New York, NY, USA, 2012. ACM.
- [13] Y. Huang, D. Evans, J. Katz, and L. Malka. Faster Secure Two-Party Computation Using Garbled Circuits. In *USENIX Security Symposium*, 2011.
- [14] R. Jagomägis. SecreC: a Privacy-Aware Programming Language with Applications in Data Mining. Master’s thesis, University of Tartu, 2010.
- [15] F. Kerschbaum. Automatically optimizing secure computation. In *Proceedings of the 18th ACM conference on Computer and communications security, CCS ’11*, pages 703–714, New York, NY, USA, 2011. ACM.
- [16] F. Kerschbaum. Expression rewriting for optimizing secure computation. In *Conference on Data and Application Security and Privacy*, 2013.
- [17] V. Kolesnikov and T. Schneider. Improved Garbled Circuit: Free XOR Gates and Applications. In L. Aceto, I. Damgård, L. Goldberg, M. Halldórrson, A. Ingólfssdóttir, and I. Walukiewicz, editors, *ALP 2008*, volume 5126 of *LNCS*, pages 486–498. Springer, 2008.
- [18] B. Kreuter, A. Shelat, and C.-H. Shen. Billion-gate secure computation with malicious adversaries. In *Proceedings of the 21st USENIX conference on Security symposium, Security’12*, pages 14–14, Berkeley, CA, USA, 2012. USENIX Association.
- [19] L. Kruger, S. Jha, E.-J. Goh, and D. Boneh. Secure function evaluation with ordered binary decision diagrams. In *Proceedings of the 13th ACM conference on Computer and communications security (CCS’06)*, Alexandria, VA, Oct. 2006.
- [20] L. Malka. VMCrypt: modular software architecture for scalable secure computation. In *ACM Conference on Computer and Communications Security*, pages 715–724, 2011.
- [21] D. Malkhi, N. Nisan, B. Pinkas, and Y. Sella. Fairplay: A Secure Two-Party Computation System. In *13th Conference on USENIX Security Symposium*, volume 13, pages 287–302. USENIX Association, 2004.
- [22] B. Mood. Optimizing Secure Function Evaluation on Mobile Devices. Master’s thesis, 2012, University of Oregon.
- [23] B. Mood, L. Letaw, and K. Butler. Memory-Efficient Garbled Circuit Generation for Mobile Devices. In *Financial Cryptography and Data Security*, volume 7397. Springer Berlin Heidelberg, 2012.
- [24] B. Pinkas, T. Schneider, N. Smart, and S. Williams. Secure Two-Party Computation Is Practical. In M. Matsui, editor, *Asiacrypt*, volume 5912 of *LNCS*, pages 250–267. Springer, 2009.
- [25] M. Rabin. How to Exchange Secrets by Oblivious Transfer. Technical Report TR-81, Harvard Aiken Computation Laboratory, 1981.
- [26] J. Ristioja. An analysis framework for an imperative privacy-preserving programming language. Master’s thesis, Institute of Computer Science, University of Tartu, 2010.
- [27] T. Schneider. *Engineering Secure Two-Party Computation Protocols - Design, Optimization, and Applications of Efficient Secure Function Evaluation*. Springer, 2012.
- [28] S. Setty, R. McPherson, A. J. Blumberg, and M. Walfish. Making Argument Systems for Outsourced Computation Practical (Sometimes). In *NDSS*, 2012.
- [29] S. Setty, V. Vu, N. Panpalia, B. Braun, A. J. Blumberg, and M. Walfish. Taking proof-based verified computation a few steps closer to practicality. In *Proceedings of the 21st USENIX conference on Security symposium*, Berkeley, CA, USA, 2012.
- [30] A. Yao. Protocols for Secure Computations. In *23rd Symposium on Foundations of Computer Science*, pages 160–164. IEEE Computer Society, 1982.

A PCF Semantics

The PCF file format consists of a header section that declares the input size, followed by a list of operations that are divided into subroutines. At runtime, these operations manipulate the internal state of the PCF interpreter, causing gates to be emitted when necessary. The internal state of the PCF interpreter consists of an instruction pointer, a call stack, an array of wire values, and an array of pointers. The pointers are positive integers. Wire values are 0, 1, or \perp , where \perp represents a value that depends on input data, which is supplied by the code that invokes the interpreter. Each position in the wire table can be treated as a stack.

Each PCF instruction can take up to 3 arguments. The instructions and their semantics are as follows:

CLABEL/SETLABELC Appears only in the header, used for setting the input size for each party. CLABEL declares the bit width of a value, SETLABELC sets the value.

FUNCTION Denotes the beginning of a subroutine. When the subroutine is called, the instruction pointer is set to the position following this instruction.

GADGET Denotes a branch target

BRANCH Takes two arguments: a target, declared with GADGET, and a location in the wire table. In the wire value is 0, the instruction pointer is set to the instruction following the target. If the wire value is 1, the instruction pointer is incremented. If the wire value is \perp , evaluation halts with an error.

FUNC Calls a subroutine, pushing the current instruction pointer onto the call stack.

PUSH Pushes a copy of the wire value at a specified position onto the stack at that position.

POP Pops a stack at a specified position. If there is only one value on that stack, evaluation halts with an error.

ALICEIN32/BOBIN32 Fetches 32 input bits from one party, beginning at a specified *bit* position in that party's input. The bit position is specified by an array of 32 values in the wire table. If any of the values is \perp , evaluation halts with an error. The input values will all have the value \perp , and will be stored in the wire table at positions 0 through 31.

SHIFT OUT Outputs a single bit for a given party

RETURN Return from a subroutine. The instruction pointer is repositioned to the value popped from the top of the call stack.

STORECONSTPTR Sets a value in the pointer table

OFFSETPTR Adds a value to a pointer, specified by an array of 32 wire values starting at a position in the wire table. If any value in the array is \perp , evaluation halts with an error.

PTRTOWIRE Saves a pointer value as a 32 bit unsigned integer. Each of the bits is pushed onto the stack at a location in the wire table.

PTRTOPTR Copies a value from one position in the pointer table to another.

CPY121 Copy a wire value from a position specified by a pointer to a statically specified position.

CPY32 Copy a wire value from a statically specific position to a position specified by a pointer.

g0,0g0,1g1,0g1,1 Compute a gate with the specified truth table on two input values from the wire table, with output stored at a specified position. Logic simplification rules are applied when one or both of the input values is \perp . If no simplification is possible, then the output will be \perp and the interpreter will emit a gate. This is used for both local computations such as updating a loop index, and for computing the gates used by the protocol.

A.1 Example PCF Description

Below is an example of a PCF file. It iterates over a loop several times times, XORing the two parties' inputs with a bit from the internal state.

```
GADGET: main
CLABEL ALICEINLENGTH 32
CLABEL BOBINLEGNTH 32
CLABEL xxx 32
SETLABELC ALICEINLENGTH 128
SETLABELC ALICEINLENGTH 128
FUNCTION: main
1111 32 0 0
0000 33 0 0
0000 34 0 0
0000 35 0 0
GADGET: L
0110 36 35 34
0001 35 36 36
0110 36 34 33
0001 34 36 36
0110 36 33 32
0001 33 36 36
ALICEINPUT32 0 0
0001 36 0 0
BOBINPUT32 0 0
0001 37 0 0
0110 38 37 36
0110 39 33 38
SHIFT OUT ALICE 39
BRANCH L 35
RETURN xxx
```

Control Flow Integrity for COTS Binaries *

Mingwei Zhang and R. Sekar

*Stony Brook University
Stony Brook, NY, USA.*

Abstract

Control-Flow Integrity (CFI) has been recognized as an important low-level security property. Its enforcement can defeat most injected and existing code attacks, including those based on Return-Oriented Programming (ROP). Previous implementations of CFI have required compiler support or the presence of relocation or debug information in the binary. In contrast, we present a technique for applying CFI to stripped binaries on x86/Linux. Ours is the first work to apply CFI to complex shared libraries such as glibc. Through experimental evaluation, we demonstrate that our CFI implementation is effective against control-flow hijack attacks, and eliminates the vast majority of ROP gadgets. To achieve this result, we have developed robust techniques for disassembly, static analysis, and transformation of large binaries. Our techniques have been tested on over 300MB of binaries (executables and shared libraries).

1 Introduction

Since its introduction by Abadi et. al. [1, 2], Control-Flow Integrity (CFI) has been recognized as an important low-level security property. Unlike address-space randomization [24, 5] and stack cookies [12, 17], CFI’s control-flow hijack defense is not vulnerable to the recent spate of information leakage and guessing attacks [40, 37, 16]. Unlike code injection defenses such as DEP (data execution prevention), CFI can protect from existing code attacks such as return-oriented programming (ROP) [38, 9, 49] and jump-oriented programming (JOP) [10, 7]. In addition to exploit defense, CFI provides a principled basis for building other security mechanisms that are robust against low-level code attacks, as evidenced by its application in software fault isolation [27, 47] and sandboxing of untrusted code [15, 46].

An important feature of CFI is that it can be meaning-

fully enforced on binaries. Indeed, some applications of CFI, such as sandboxing untrusted code, explicitly target binaries. Most existing CFI implementations, including those in Native Client [46], Pittsfield [27], Control-flow locking [6] and many other works [22, 3, 42, 4, 36] are implemented within compiler tool chains. They rely on information that is available in assembly code or higher levels, but unavailable in COTS binaries. The CFI implementation of Abadi et al [2] relies on relocation information. Although this information is included in Windows libraries that support ASLR, UNIX systems (and specifically, Linux systems) rely on position-independent code for randomization, and hence do not include relocation information in COTS binaries. We therefore develop a new approach for enforcing CFI on COTS binaries without relocation or other high-level information.

Despite operating with less information, the security and performance provided by our approach are comparable to that of the existing CFI implementations. Moreover, our implementation is robust enough to handle complex executables as well as shared libraries. We begin by summarizing our approach and results.

1.1 CFI for COTS Binaries

We present the first practical approach for CFI enforcement that scales to large binaries as well as shared libraries without requiring symbol, debug, or relocation information. We have developed techniques that cope with the challenges presented by static analysis and transformation of large programs, including those of Firefox, Adobe Acrobat 9, GIMP-2.6 and glibc. In our experiments, we have transformed and tested over 300MB of binaries. Some of the key features of our design are:

- *Modularity:* Each shared library and executable is instrumented independently to enforce CFI. Our technique ensures that when an executable is loaded and run, CFI property is enforced globally across the executable and all the shared libraries used by it.

*This work was supported in part by AFOSR grant FA9550-09-1-0539, NSF grant CNS-0831298, and ONR grant N000140710928.

- *Transparency*: If our instrumentation made even the smallest changes to (stack, heap or static) memory used by a program, it can cause complex programs to fail or function differently. As an example, consider saved return addresses on the program stack. Since code rewriting causes instruction locations to change, a straight-forward implementation would change these saved return addresses. Unfortunately, programs use this information in several ways:

- Position-independent code (PIC) computes the locations of static variables from return address.
- C++ exception handler uses return addresses to identify the function (or more specifically, the try-block within the function) to which the exception needs to be dispatched.
- A program may use the return address (and any other code pointer) to read constant data stored in the midst of code, or more generally, its own code.

Changes to saved return address would cause these uses to break, thus leading to application failure. For this reason, our instrumentation has been designed to provide full transparency.

The principal challenge in achieving full transparency is one of performance. To address this challenge, we have developed new optimization techniques.

- *Compiler independence and support for hand-coded assembly*: Our approach does not make strong assumptions regarding the compiler used to generate a binary, such as the the conventions for generating jump tables. Indeed, our code has been tested with hand-written assembly, such as that found in low-level libraries (e.g., glibc). It has been tested with the two popular compilers on Linux, GCC and LLVM.

1.2 Quality of Protection

An ideal CFI implementation will restrict program execution to exactly the set of program paths that can be taken. In practice, due to the fact that targets of indirect control-flow (ICF) transfers are difficult to predict, CFI implementations enforce a conservative approximation of ideal CFI. Different techniques enforce different approximations, so a natural and important question concerns the relative strengths of these techniques. To answer this question, we propose a simple metric, called average indirect target reduction (AIR) which quantifies the fraction of possible indirect targets eliminated by a CFI technique. To compute AIR, we start with the fraction of possible targets eliminated by a CFI technique for each ICF transfer instruction, and average this number across all ICF transfer instructions. (See Definition 1 on Page 6.)

AIRs of several types of CFI are shown in Figure 1. For the base case of an unprotected program, every byte

CFI type	Description	AIR (%)
null	no CFI protection	0.00
instr	Restrict ICFs to valid instruction boundaries	79.27
bundle	Instructions grouped into 32-byte bundles [46]. All ICFs must target the start of a bundle.	96.04
reloc	CFI based on relocation information. Indirect calls/jumps to target any location present in relocation table, returns to target a location immediately following a call.	99.13
strict	Enforces property closely matching reloc-CFI but does not require relocation info.	99.08
bin	Generalizes strict-CFI to avoid special treatment of threads and exceptions	98.86

Figure 1: CFI flavors and strengths on SPEC CPU2006.

address in the code is a possible ICF target, and the AIR is 0%. We then define a coarse form of CFI called instr-CFI that limits ICF transfers to instruction boundaries. It eliminates attacks that jump to the middle of instructions. Bundle-CFI is another coarse form of CFI used in PittsField [27] and Native Client [46]. It limits ICF transfers to addresses that are multiples of 16 (PittsField) or 32 (Native Client).

The next version, reloc-CFI, captures the strength of CFI implementation described by Abadi et al [2]. It relies on relocation information in binaries. (See Section 4.2 for more discussion).

Large and complex binaries contain several exceptions to the simple model of calls, returns and indirect jumps embodied in many CFI works:

- *Returns used as jumps*. Return instructions are some times used to jump to functions by pushing their address on the stack and returning. Examples include code for thread context switching, signal handling, etc.
- *Returns to caller function, but not a return address*. Some times, returns go back to a caller, but don't target a return address, e.g., due to C++ exceptions.
- *Jumps to return addresses*. Functions such as `longjmp` use an indirect jump that targets a return address.
- *Runtime generation of new ICF targets*. Some applications create ICF targets on the fly using `dlopen` to add additional libraries at any point during runtime.
- *Indirect jumps using arithmetic operations*. Low-level assembly code can contain ICF targets that are computed using multiple arithmetic operations.

To cope with these exceptions, our approach, called bin-CFI, avoids making any of the common assumptions regarding ICF targets in general. Instead, it relies on static analysis and a very conservative set of assumptions so that it can scale to large executables and libraries.

Note that bin-CFI eliminates about 99% of possible indirect targets. Moreover, it experiences only a small

decrease in AIR as compared to reloc-CFI. This provides evidence that *our approach achieves compatibility with COTS binaries without incurring a major reduction in its quality of protection*.

To further pinpoint the sources of the slight decrease in AIR, we implemented a stricter version of bin-CFI called strict-CFI. It uses the same binary analysis techniques as bin-CFI, but instead of providing a general way to handle exceptions and threads, it simply uses a relaxed policy for a few specific instructions in system libraries that perform thread switching or exception unwinding. Note that the strict-CFI has an AIR very close to that of reloc-CFI, pointing out that the sources of AIR decrease are the exceptions that need to be made in order to support large and complex binaries. Effective precision loss incurred by our static analysis is very small (0.05%) as compared to the use of relocation information.

1.2.1 Experimental Evaluation

We present a detailed experimental evaluation of our technique. Key points include:

- **Good performance:** Techniques for achieving transparency and modularity can exact a price in terms of performance. We describe several optimization techniques in Section 6 that have reduced the overhead to about 8.54% across the SPEC CPU benchmark suite.
- **ROP and JOP defense:** As our AIR measurements indicate, about 99% of possible ICF targets have been eliminated by bin-CFI. Moreover, on the SPEC CPU 2006 benchmark, our technique also eliminated about 93% of ROP gadgets that were found by the popular ROP gadget discovery tool ROPGadget [35].
- **Control-flow hijack detection.** Our results show that bin-CFI defeats the vast majority of control-flow hijack attacks from the RIPE benchmark [45].

2 Disassembly

2.1 Background

There are two basic techniques for disassembly: *linear disassembly* and *recursive disassembly*. Linear disassembly starts by disassembling the first instruction in a given segment. Once an instruction at an address l is disassembled, and is determined to have a length of k bytes, disassembly proceeds to the instruction starting at address $l + k$. This process continues to the end of the segment.

Linear disassembly can be confused by “gaps” in code that consist of data or alignment-related padding. These gaps will be interpreted by linear disassembly as instructions and decoded, resulting in erroneous disassembly. With variable-length instruction sets such as those of x86, incorrect disassembly of one instruction

can cause misidentification of the start of the next instruction; hence these errors can cascade even past the end of gaps.

Recursive disassembly uses a different strategy, one that is similar to a depth-first construction of program’s control-flow graph (CFG). It starts with a set of code entry points specified in the binary. For an executable, there may be just one such entry point specified, but for shared libraries, the beginning of each exported functions is specified as well. The technique starts by disassembling the instruction at an entry point. Subsequent instructions are disassembled in a manner similar to linear disassembly. The difference with linear disassembly occurs when control-flow transfer instructions are encountered. Specifically, (a) each target identified by a direct control-flow transfer instruction is added to the list of entry points, and (b) disassembly stops at unconditional control-flow transfers.

Unlike linear disassembly, recursive disassembly does not get confused by gaps in code, and hence does not produce incorrect disassembly¹. However, it fails to disassemble code that is reachable only via ICF transfers.

Incompleteness of recursive disassembly can be mitigated by providing it a list of all targets that are reachable via ICF transfers. This list can be computed from *relocation information*. However, in stripped binaries, which typically do not contain relocation information, recursive disassembly can fail to disassemble significant parts of the code.

2.2 Our Disassembly Technique

The above discussion on using relocation information to complete recursive disassembly suggests the following strategy for disassembly:

- Develop a static analysis to compute ICF targets.
- Modify recursive disassembly to make use of these as possible entry points.

Unfortunately, the first step will typically result in a superset of possible ICF targets: some of these locations don’t represent code addresses. Thus, blindly following ICF targets computed by static analysis can lead to incorrect disassembly. We therefore use a different strategy, one that combines linear and recursive disassembly techniques, and uses static analysis results as positive (but not definitive) evidence about correctness of disassembly.

Our approach starts by eagerly disassembling the entire binary using linear disassembly, which is then checked for errors. The error checking step primarily relies on the steps used in recursive disassembly. Finally,

¹This does rely on some assumptions: (a) calls must return to the instruction following the call, (b) all conditional branches are followed by valid code, and (c) all targets of (conditional as well as unconditional) direct control-flow transfers represent legitimate code. These assumptions are seldom violated, except in case of obfuscated code.

an error correction step identifies and marks regions of disassembled code as representing gaps. The error detection step relies on the following checks:

- *Invalid opcode*: Some byte patterns do not correspond to any instruction, so attempts to decode them will result in errors. This is relatively rare because x86 machine code is very dense. But when it occurs, it is a definitive indicator of a disassembly error.
- *Direct control transfers outside the current module*. Cross-module transfers need to use special structures called program-linkage table (PLT) and global offset table (GOT), and moreover, need to use ICF transfers. Thus, any direct control transfer to an address outside the current module indicates erroneous disassembly.
- *Direct control transfer to the middle of an instruction*: This can happen either because of incorrect disassembly of the target, or incorrect disassembly of the control-flow transfer instruction. Detection of additional errors near the source or target will increase our confidence regarding which of the two has been incorrectly disassembled. In the absence of additional information, our approach considers both possibilities.

Since errors in linear disassembly arise due to gaps, our error correction step relies on identifying and marking these gaps. An incorrectly disassembled instruction signifies the presence of a gap, and we need to find its beginning and end. To find the beginning of the gap, we simply walk backward from the erroneously disassembled instruction to the closest preceding unconditional control-flow transfer. If there are additional errors within a few bytes preceding the gap, the scan is continued for the next preceding unconditional control-flow transfer. To find the end of the gap, we rely on static analysis results (Section 3). Specifically, the smallest ICF target larger than the address of the erroneously disassembled instruction is assumed to be the end of the gap. Once again, if there are disassembly errors in the next few bytes, we extend the gap to the next larger ICF target.

After the error correction step, all identified disassembly errors are contained within gaps. At this point, the binary is disassembled again, this time avoiding the disassembly of the marked gaps. If no errors are detected this time, then we are done. Otherwise, the whole process needs to be repeated. While it may seem that repetition of disassembly is an unnecessarily inefficient measure, we have used it because of its simplicity, and because disassembly errors have been so rare in our implementation that no repetition was needed for the vast majority of our benchmarks.

3 Indirect Control Flow Analysis

In this section, we describe a static analysis for discovering possible ICF targets. We classify ICF targets into

several categories, and devise distinct analyses to compute them:

- *Code pointer constants (CK)* consists of code addresses that are computed at compile-time.
- *Computed code addresses (CC)* include code addresses that are computed at runtime.
- *Exception handling addresses (EH)* include code addresses that are used to handle exceptions.
- *Exported symbol addresses (ES)* include export function addresses.
- *Return addresses (RA)* include the code addresses next of a call.

Our static analysis results are filtered to retain only those addresses that represent valid instruction boundaries in disassembled code.

3.1 Identifying Code Pointer Constants (CK)

In general, there is no way to distinguish a code pointer from other types of constants in code. So, we take a conservative approach: any constant that “looks like a code pointer,” as per the following tests, is included in CK:

- it falls within the range of code addresses in the current module.
- it points to an instruction boundary in disassembled code.

Note that a module has no compile-time knowledge of addresses in another module, and hence it suffices to check for constants that fall within the range of code addresses in the current module. For shared libraries, absolute addresses are unknown, so we check if the constant represents a valid offset from the base of the code segment. It is also possible that the offset may be with respect to the GOT of the shared library, so our validity check takes that into account as well.

The entire code and data segments are scanned for possible code constants as determined by the procedure in the preceding paragraph. Since 32-bit values need not be aligned on 4-byte boundaries on x86, we use a 4-byte sliding window over the code and data to identify all potential code pointer constants.

3.2 Identifying Computed Code Pointers (CC)

Whereas our CK analysis was very conservative, it is difficult to bring the same level of conservativeness to the analysis of computed code pointers. This is because, in general, arbitrary computations may be performed on a constant before it is used as an address, and it would be impossible to estimate the results of such operations with any degree of accuracy. However, these general cases are just a theoretical possibility. The vast majority of code is generated from high-level languages where arbitrary

pointer arithmetic on code pointers isn't meaningful². Even for hand-written assembly, considerations such as maintainability, reliability and portability lead programmers to avoid arbitrary arithmetic on code pointers. So, rather than supporting arbitrary code pointer computation, we support computed code pointers in a limited set of contexts where they seem to arise in practice. Indeed, the only context in which we have observed code pointer arithmetic is that of jump tables³.

The most common case of jump tables arise from compiling switch statements in C and C++ programs. If these were the only sources of CC, then a simple approach could be developed that is based on typical conventions used by compilers for translating switch statements. However, this approach isn't feasible in our case since we wish to handle many low-level libraries that contain hand-written assembly code. So, we begin by identifying properties that we believe are generic to jump tables:

- Jump table targets are intra-procedural: the ICF transfer instruction and ICF target are in the same function. (We don't require function boundaries — we estimate them conservatively, as described below.)
- The target address is computed using simple arithmetic operations such as additions and multiplication.
- Other than one quantity that serves as an index, all other quantities involved in the computation are constants in the code or data segment.
- All of the computation takes place within a fixed size window of instructions, currently set to 50 instructions in our implementation.

Based on these characteristics, we have developed a static analysis technique to compute possible CC targets. It uses a three-step process. The first step is the identification of function boundaries and the construction of a control-flow graph. In the absence of full symbol table information, it is difficult to identify all function boundaries, so we fall back to the following approach that uses information about exported function symbols. We treat the region between two successive exported function symbols as an approximation of a function. (Note that this approximation is conservative, as there may be non-exported functions in between.) We then construct a control-flow graph for each region.

In the second step, we identify instructions that perform an indirect jump. We perform a backward walk from these instructions using the CFG. All backward paths are followed, and for each path, we trace the

²This is true even in languages that are notorious for pointer arithmetic, such as C.

³C++ exception handling also involved address arithmetic on return addresses, but we can rely on exception handler information that must be included in binaries rather than the CC analysis.

chain of data dependences to compute an expression for the indirect jump target. This expression has the form $*(CE_1 + Ind) + CE_2$, where CE_1 and CE_2 denote expressions consisting of only constants, Ind represents the index variable, and $*$ denotes memory dereferencing. In some cases, it is possible to extend the static analysis to identify the range of values that can be taken by Ind . However, we have not implemented such an analysis, especially because the index value may come from other functions. Instead, we make an assumption that valid Ind values will start around 0.

In the third step, we enumerate possible values for the index variable, compute the jump target for each value, and check if it falls within the current region. Specifically, we check if $CE_1 + Ind$ falls within the data or code segment of the current module, and if so, retrieve the value stored at this location. It is then added with CE_2 and the result checked to determine if it falls within the current region. If so, the target is added to the set CC. If either of these checks fail, Ind value is deemed invalid.

We start from Ind value of 1, and explore values on either side until we reach values for which the computed target is invalid.

We point out that the backward walk through the CFG can cross function boundaries, e.g., traversing into the body of a called function. It may also go backwards through indirect jumps. To support this case, we extend the CFG to capture indirect jumps discovered by the analysis. The maximum extent of backward pass is bounded by the window size specified above.

The above procedure can fail in some cases, e.g., if CC computation is dispersed beyond the 50-instruction window used in the analysis, or if the computation does not have the form $*(CE_1 + Ind) + CE_2$. In such cases, we can conservatively add every instruction address within the region to CC.

3.3 Identifying Other Code Addresses

Below, we describe the computation of the three remaining types of code pointers: exception handlers (EH), exported symbols (ES), and return addresses (RA).

In ELF binaries, exception handlers are also valid ICF targets. They are constructed by adding a base address with an offset. The base addresses and offsets are stored in ELF sections `.eh_frame` and `.gcc_except_table` respectively. Both these sections are in DWARF [26] format. We use an existing tool, katana [29, 30], to parse these DWARF sections and get both base addresses and offsets, and thus compute the set EH. (We point out that the CC analysis mentioned above won't be able to discover these EH targets because DWARF format permits variable length numeric encoding such as LEB128, and hence the simple technique of scanning for 32-bit constant values won't work.)

Exported symbol (ES) addresses are listed in the dynamic symbol table, which is found in the `.dynamic` section of an ELF file.

Return addresses (RA) are simply the set of locations that follow a call instruction in the binary. Thus, they can be computed following the disassembly step.

4 Defining and Assessing CFI for Binaries

4.1 A Metric for Measuring CFI Strength

Previous works on CFI have relied on analysis of higher level code to effectively narrow down ICF targets. Since binary analysis is generally weaker than analyses on higher-level code, our CFI enforcement is likely to be less precise. It is natural to ask how much protection is lost as a result. To answer this question, we define a simple metric for quality of protection offered by a CFI technique.

Definition 1 (Average Indirect target Reduction (AIR))

Let i_1, \dots, i_n be all the ICF transfers in a program and S be the number of possible ICF targets in an unprotected program. Suppose that a CFI technique limits possible targets of ICF transfer i_j to the set T_j . We define AIR of this technique as the quantity

$$\frac{1}{n} \sum_{j=1}^n \left(1 - \frac{|T_j|}{S} \right)$$

where the notation $|T|$ denotes the size of set T .

On x86, where branches can target any byte offset, S is the same as the size of code in a binary.

4.2 A Simple CFI Property based on Relocation

CFI techniques are generally based on the following model of how ICF transfers are used in source code:

1. *Indirect call (IC)*: An indirect call can go to any function whose address is taken, including those addresses that are implicitly taken and stored in tables, such as virtual function tables in C++.
2. *Indirect jump (IJ)*: Since compiler optimizations⁴ can replace an indirect call (IC) with indirect jump (IJ), the same policy is often applied to indirect jumps as well.
3. *Return (RET)*: Returns should go back to any Return Address (RA), i.e., an instruction following a call.

It is theoretically possible to further constrain each of these sets, and moreover, use different sets for each ICF transfer. However, implementations typically don't use this option, as increased precision comes with certain drawbacks. For instance, the callers of functions in shared libraries (or dynamically linked libraries in the

case of Microsoft Windows) are not known before runtime, and hence it is difficult to constrain their returns more narrowly than described above. Moreover, some techniques rely on relocation information, which does not distinguish between targets reachable by IC from those reachable by IJ, or between the targets reachable by any two ICs. Hence they do not refine over the above property. For this reason, we refer to the above CFI property as reloc-CFI.

The description of implementation in Abadi et al [2] indicates their use of relocation information, and confirms the above policy regarding ICs. No specifics are provided regarding IJs and returns, but for reasons described above, we believe that they support the reloc-CFI policy described above. We also note that indexed hooks [22] uses a single table for ICs and IJs, and another for returns, enforcing reloc-CFI but in a kernel environment.

4.3 Strict-CFI: A CFI Property for Binaries Closely Matching Reloc-CFI

Strict-CFI is derived from reloc-CFI, except that it uses ICF targets computed by our ICF target analysis rather than relocation information. In addition, strict-CFI incorporates an extension needed to handle features such as exception handling and multi-threading. Specifically, these features are used by a handful of instructions in system libraries, and we simply relax the above policy for these instructions:

- Instructions performing exception related stack unwinding are permitted to go to any exception handler landing pad (EH).
- Instructions performing context switches are permitted to use any type of ICF transfer to transfer to a function address.

Since they apply to a very small fraction of ICF transfers in a program, their overall effect on AIR is negligible. Thus, the difference in AIR between reloc-CFI and strict-CFI will pinpoint the precision loss due to the use of static analysis in place of relocation information.

4.4 Bin-CFI: CFI for Complex Binaries

Complex binaries can contain exceptions to the simple model of ICF transfers outlined earlier. To define a suitable CFI property for such binaries, we introduce a category of ICF transfer in addition to RET, IC and IJ described earlier. This category, called PLT, includes all ICF transfers in the program linkage table, a section of code used in dynamic linking⁵.

We are now ready to define bin-CFI as shown in Figure 2.

⁴Specifically, a tail call optimization that replaces a call occurring at the very end of a function with a jump.

⁵Specifically, for each function belonging to another module, a stub routine is created by the compiler in this section.

	Returns (RET), Indirect Jumps (IJ)	PLT targets, Indirect Calls (IC)
Return addresses (RA)	Y	
Exception handling addresses (EH)	Y	
Exported symbol addresses (ES)		Y
Code pointer constants (CK)	Y	Y
Computed code addresses (CC)	Y	Y

Figure 2: Bin-CFI Property Definition

It is easy to see that strict-CFI is stricter than bin-CFI. The reasons for relaxing strict-CFI are as follows. In general, there is no easy way to distinguish between returns used for purposes such as stack unwinding, longjmp, thread context switch, and function dispatch from (the more common) use of returning from functions. We therefore permit returns to go to any of the valid targets corresponding to each of these uses. Returns are sometimes broken up into a pop and jump, so all possible targets of RET are permissible targets of IJ. This explains the first column of the table.

Since the purpose of PLT stubs is to dispatch cross-module calls, it would seem that the targets can only be exported symbols from other modules. However, recent versions of gcc support a new function type called `gnu_indirect_function`, which allows a function to have many different implementations, with the most suitable one selected at runtime based on factors such as the CPU type. Currently, many glibc low level functions such as `memcpy`, `strcmp` and `strlen` use this feature. To support this feature, a library exports a chooser function that selects at runtime which of the many implementations is going to be used. These implementation functions may not be exported at all. To avoid breaking such programs, the policy for PLT should be relaxed to include code pointers in the target library. This is what we have done on the second column of Figure 2.

Indirect calls should go to the targets in one of the sets CC or CK. Since these two sets are usually much larger than ES, we chose to merge IC and PLT to use the same table of valid targets.

5 Implementation

Although our design is largely applicable to most architectures, our implementation targets 32-bit x86 processors running Linux. For this reason, some implementation aspects discussed below are specific to this platform.

5.1 Disassembly

Binaries on Linux (and most other UNIX systems) use the ELF (Executable and Linkable Format) [25] format. We support binaries that represent executables and shared libraries. The ELF format divides a binary into several sections, each of which may contain code, read-only data, initialized data, and so on. While our approach utilizes the data in read-only data sections, it is mainly concerned with the code sections.

Our implementation utilizes `objdump` to perform linear disassembly. We have built our disassembly error detection and correction components on top of `objdump`. In our experience, disassembly errors occurred primarily due to insertion of null padding generated by legacy code or linker script. In addition, we discovered jump table data in the middle of code in `libffi.so` and `libxul.so`.

There were also several instances where conditional jumps targeted the middle of an instruction. Further analysis revealed that these errors occurred with instructions that had optional prefixes, such as the “lock” prefix. We eliminated this error by treating these prefixes as independent instructions, so that jumps could target the instruction with or without the prefix.

5.2 Instrumentation and Regeneration of Binary

After disassembly, the resulting code is instrumented to enforce CFI. The specifics of this instrumentation are described in Section 5.3. Below we describe the generation of a binary from instrumented code, since a general understanding of this process will enable a fuller understanding of the instrumentation steps.

Instrumentation is performed on assembly representation. This simplifies our implementation since it does not need to be concerned with details such as encoding and decoding of instructions. Moreover, it can use labels instead of addresses. In particular, for each instruction location A in the disassembler output of `objdump`, we associate a symbolic label L_A as follows:

```
L_8040930: movl %ecx, %eax
```

These symbolic labels are used as targets of direct branch instructions, which means that the assembler will take care of fixing up the branch offsets. (These offsets will typically change since we are inserting additional code during instrumentation.)

After rewriting, the instrumented assembly file is processed using the system assembler (in our case, the GNU assembler `gas`) to produce an object file. We extract the code from this object file and then use the `objcopy` tool to inject it into the original ELF file. Note that the original code sections are not overwritten. This ensures that any attempt by the instrumented program to read its own code will produce the same results as the original program.

The final step prepares the ELF file produced by objcopy for execution. This step requires relocation actions on the newly added segment, and updating the ELF header to set its entry point to the segment containing instrumented code. The original code segments are made unexecutable. For shared libraries, it is also necessary to update the dynamic symbol sections.

5.3 Instrumentation for CFI

As described above, instrumented code resides in a different code segment (and hence a different memory location) from the original code. This means that function pointer values, which will typically appear in the code as constants, will have incorrect values. Unfortunately, it is not possible to fix them up automatically, since we cannot distinguish constants representing code addresses from other types of constants. It would obviously be unsound to modify a constant value that does not represent a code pointer.⁶

The typical way to deal with this uncertainty, employed in dynamic binary translation (DBT) [8], is to wait until a value is used as the target of an ICF transfer. At that point, this target value is translated into the corresponding location in the instrumented code. This translation is performed using a table that consists of pairs of the form

$\langle \text{original address}, \text{new address} \rangle$

At runtime, `addr_trans`, a piece of trampoline code, performs address translation. (In fact, there are two such trampolines, one corresponding to each column of Figure 2.) Instrumentation is inserted at the site of the original indirect control-flow transfer instruction as shown in Figure 3.

060c0: call *%ecx	L_060c0: push \$060c2
060c2:	movl %eax, %gs:0x44
	movl %ecx, %eax
	jmp addr_trans
L_060c2:	

Figure 3: Original (left) and Instrumented code (right) for ICF transfer

This code saves the register (eax) used by the instrumentation, and moves the target address into it.⁷ Then the original indirect jump (or call) is replaced with a direct jump to the trampoline routine, `addr_trans`. Note the use of labels such as L_060c0 that are used to associate locations in the instrumented code with the corresponding

⁶Here again, relocation information can address this uncertainty, but in our case, this is unavailable.

⁷Note that %gs points to the base of thread-local storage, and %gs:0x44 is not used by existing system software.

original address, namely, 060c0. As a result, the translation table can consist of entries of the form

$\langle A, L_A \rangle$

for each valid ICF target A. As noted earlier, there are two address translation routines, one corresponding to each column of Figure 2. The valid ICF targets for each table consists of the subset of ICF targets computed by the static analysis described in Section 3 that appear in the corresponding column of Figure 2.

The details of `addr_trans` are as follows: After saving registers and flags needed for its operation, `addr_trans` performs an address range check to determine if the target is within the current module. If not, this represents a cross-module control transfer that is described later in this section. After the range check, `addr_trans` performs address translation. Our implementation relies on closed hashing [44] to perform an efficient lookup of the table described above. Rather than storing just the target address L_A in the table, our implementation stores code that transfers control to L_A . For instance, the hash table entry to translate a code address 0x060c2 looks as follows.

0x060c2	movl %gs:0x44, %eax; jmp L_060c2
---------	----------------------------------

If no translation is found for the target address, `addr_trans` will set an error code to help in debugging, and terminate the program.

Note that, for shared libraries, translation table only contains the offsets rather than absolute addresses. Consequently, the base address of the module needs to be subtracted from the runtime address given to the translation routine. We rely on the dynamic linker to patch the routine with the module’s base address when the module is loaded.

In order to preserve the functionality of original code, it is necessary to ensure that the instrumentation does not modify any of the registers or memory used by the program. It is relatively easy to avoid changes to memory, or registers other than the program counter (PC). Since instrumentation changes code locations (as described earlier), it is not possible to preserve the PC register. So, what we need to do is to add a compensation for any operation that uses the PC for any purpose other than fetching the next instruction. Fortunately, on x86, there are only two instructions that use PC this way: `call` and `return`. A `call X` is translated into a `push next; jmp X`, where `next` denotes the address of the instruction following `call` in the original program. Similarly, a `return` is translated into a `pop` followed by a direct jump. Note that after this transformation, none of the instructions in the original program involve movement of data between PC and other registers or memory⁸, thus ensuring that

⁸In x86-64 architecture, any PC-relative data addressing needs to

program behavior is unaffected by our instrumentation.

Modularity. Support for shared libraries is achieved as follows. Our technique rewrites a single module (an executable or a shared library) at a time. There is exactly one version of a transformed shared library, regardless of the context (or the executable) in which it is used. Note that we transform all shared libraries, including `glibc` and `ld.so`.

As described before, `addr_trans` already handles intra-module control transfers. Inter-module transfers rely on a two-stage process. In the first stage, a *global translation table (GTT)* is used to map an ICF target to the translation routine address in the target module. This table is constructed as follows. Since shared libraries must begin at page boundaries, any two modules have to be apart by at least 4KB, the page size on 32-bit Linux systems. Thus, it is enough to use the leading 20 bits of the ICF target in this lookup table. We use a simple array implementation for GTT since there are only $2^{20} = 1M$ entries in this table. This array is made read-only in order to protect it. The second stage performs a lookup in the destination module, using the address translation table for that module. We use the term *module translation table (MTT)* for the translation table that specifies translations for addresses within the module.

Changes to the Loader. Note that the GTT needs to be updated as and when modules are loaded. Naturally, the best place to do this is the dynamic linker. We modified the source code of `ld.so` to accomplish this. Our change uniformly handles the typical case of the loader mapping all of shared libraries referenced by an executable (or another shared library loaded by the loader), as well as the less common case of an application using `dlopen` and `dlclose` primitives to load and unload libraries at runtime. Our changes relate to about 300 lines of the source code of `ld.so`.

Our loader modification also addressed two other idiosyncrasies of `ld.so`. First, note that our approach modifies the entry point of a binary. Thus, any program that uses the entry point for purposes other than jumping to it may not work any more. As it turns out, `ld.so` does make use of this information when it is invoked to load a program, as in `ld.so <binary>`. We changed the loader so that it compensates for the change in the entry point, and hence works correctly in all cases.

The second idiosyncrasy concerns the use of return instructions for lazy symbol resolving. Lazy symbol resolving is implemented by the `_dl_runtime_resolve` function (or `_dl_runtime_profile` if profiling is enabled) in `ld.so`. This function computes the target address corresponding to the symbol, pushes this address on the top of stack, and returns. For this to work correctly,

be translated too. This can be done easily by modifying the offset value.

```
060b1: call 060c0          L_060b1: call S_060b1
      ....                  ....
      .....                 S_060b1: add $offset, (%esp)
                           jmp L_060c0
```

Figure 4: Optimized instrumentation of calls

turns should be permitted to target exported symbols, further decreasing the accuracy of our CFI implementation. Instead, we chose to modify the loader to use indirect jumps instead of returns, and restricted the target of these jumps with the policy shown in Figure 2 for PLT entries.

Signals. Signal is another mechanism to redirect program control flow. If a program registers its signal handlers, once again we will have the problem that the program will specify the location of the handler in original code, whereas we want the signal to be delivered to the instrumented code. (This problem arises because signals are delivered by the kernel, which is not aware of the address translations used to correctly handle code pointers.)

Our implementation intercepts `sigaction` and `signal` system calls, and stores the address of the signal handlers specified by these calls in a table. The signal handler argument is then changed so that control will be transferred to a wrapper function, which contains code that jumps to the user-specified handler. Since this wrapper will be instrumented as usual, instrumented version of the user-specified handler will be invoked.

6 Optimizations

6.1 Improving Branch Prediction (BP)

Modern processors use very deep pipelines, so branch prediction misses can greatly decrease performance. Unfortunately, our translation of returns (into a combination of `pop` and `jmp`) leads to misses. When a return instruction is used, the processor is able to predict the target by maintaining a stack that keeps track of calls. When it is replaced by an indirect jump, especially one that is always made from a single trampoline routine, prediction fails.

To address this problem, we modified the transformation of calls and returns as shown in Figures 4 and 5. The original call is transformed into another call into stub code that is part of the instrumentation. There is a unique stub for each call site. The code in the stub adjusts the return address on the stack so that it will have the same value as in the untransformed program. This requires addition of a constant that represents the offset between the call instructions in the original and transformed code. Similarly, at the time of return, the return address on the stack is translated from its original value to the corresponding value in the transformed program, after which a normal return can be executed.

```

060d1:  ret      .... #address translation
          add $4, %esp
          mov %edx, (%esp)
          ret

```

Figure 5: Optimized instrumentation of returns

The key point about this transformation is that the processor sees a return in Figure 5 that returns from the call it executed (Figure 4, label L_060b1). Although the address on the program stack was adjusted (Figure 4, label S_060b1), this is reversed by address translation in Figure 5. As a result, the processor’s predicted return matches the actual return address on the stack.

6.2 Avoiding Address Translation (AT)

We explored three optimizations aimed at eliminating address translation overheads in the following cases:

AT.1 jump tables

AT.2 PIC translation

AT.3 return target speculation

For the first optimization, instead of computing an original code address and then translating it into new addresses, we create a new table that contains translated addresses. The content of the table is copied from the original table, and then each value is translated (at instrumentation time) into the corresponding new address. A catch here is that we don’t know the size of the original table. Note, however, that we have a good guess, based on the CC computation technique from Section 3.2. We first check that the index variable is within this range, and if so, use the new table. Otherwise, we use the old table, and translate the jump address at runtime.

PIC has several code patterns, including a call to `get_pc_thunk` and a call to the next instruction. The basic function of the pattern is getting the current PC and copying it into a general purpose register. In the translated code, however, `get_pc_thunk` introduces an address lookup for return. This extra translation could be avoided by translating this version into a call of the next instruction. No returns are used in this case, thereby avoiding address translation overhead. (It is worth noting that using a call/pop combination does not affect branch prediction for return instructions. The processor is able to correct for minor violations of call/return discipline.

In the third case, if a particular ICF transfer tends to target the same location most of the time, we can speed it up by avoiding address translation for this location. Instead, a comparison is introduced to determine if the target is this location, and if so, introducing a direct jump. In our implementation, we choose to apply it only to return instruction. We used profiling to determine if the return frequently targets the same location.

6.3 Violating Transparency (VT)

Using static analysis results, we can safely avoid some of the overheads associated with full transparency. The following are two optimizations we use:

VT.1 no saving of eflags

VT.2 use non-transparent calls

To achieve, VT.1, we analyze all potential indirect and direct control targets. If there is no instruction that uses eflags prior to all instructions that define it, then we can safely use VT.1. In fact, we discover that eflags is live only in a few jump tables.

When VT.2 is enabled, all return addresses are within the new code. Note that VT.2 is always enabled on PIC patterns, i.e., call of `get_pc_thunk` and call of next instruction. This is because it is simple to analyze this pattern and determine that non-transparent mode will not lead to any problems, as long as the offset added to obtain data address is appropriately adjusted.

7 Evaluation

We first evaluate functionality of our system, focusing on disassembly, and compatibility with different compilers. Next, we evaluate its effectiveness in terms of the AIR metric and attack defense. Then, we evaluate its runtime and memory overheads. Finally, we summarize the limitations of the approach and its current implementation.

Module	Package	Size	# of Instructions	# of Errors
libxul.so	firefox-5.0	26M	4.3M	0
gimp-console-2.6	gimp-2.6.5	7.7M	385K	0
libc.so	glibc-2.13	8.1M	301K	0
libnss3.so	firefox-5.0	4.1M	235K	0
libmozsqlite3.so	firefox-5.0	1.8M	128K	0
libfreebl3.so	firefox-5.0	876K	66K	0
libsoftokn3.so	firefox-5.0	756K	50K	0
libnsspr4.so	firefox-5.0	776K	41K	0
libssl3.so	firefox-5.0	864K	40K	0
libm.so	glibc-2.13	620K	35K	0
libnssdbm3.so	firefox-5.0	570K	34K	0
libsmime3.so	firefox-5.0	746K	30K	0
ld.so	glibc-2.13	694K	28K	0
gimpressionist	gimp-2.6.5	403K	21K	0
script-fu	gimp-2.6.5	410K	21K	0
libnssckbi.so	firefox-5.0	733K	19K	0
libtestcrasher.so	firefox-5.0	676K	17K	0
gfig	gimp-2.6.5	442K	17K	0
libpthread.so	glibc-2.13	666K	15K	0
libnsl.so	glibc-2.13	448K	15K	0
map-object	gimp-2.6.5	257K	15K	0
libresolv.so	glibc-2.13	275K	13K	0
libnssutil3.so	firefox-5.0	311K	13K	0
Total		58M	5.84M	0

Figure 6: Disassembly Correctness

Application Name	Experiment
Wireshark v1.6.2	capture packets on LAN for 20 minutes
gedit v3.2.3	open multiple files; edit; print; save
lyx v2.0.0	open a large report; edit; convert to pdf/dvi/ps
acroread9	open 20 pdf files; scroll;print;zoom in/out
mp3player 4.6.1	play an mp3 file
firefox 5 (no JIT)	open web pages
perl	execute a complex script, compare the output
vim	open file, copy/paste, search, edit
gimp-2.6	load jpg picture, crop, blur, sharpen, etc.
lynx 2.8.8dev	open web pages
ssh 5.8p1	login to a remote server
evince 3.2.1	open a large pdf file

Figure 7: Real World Program Functionality Test

7.1 Functionality

Testing transformed code. We tested the SPEC CPU2006 programs (Figure 8). This benchmark comes with scripts to verify outputs, thus simplifying functionality testing.

We also tested many real world programs including coreutils-8.16 and binutils-2.22, and medium to large programs such ssh, scp, wireshark, gedit, mp3player, perl, gimp, firefox, acroread, lyx as well as all the shared libraries used by them including libc.so.6, libpthread.so.0, libQtGui.so.4, libQtCore.so.4.

Altogether, we had to transform 786 shared libraries during testing. The total code transformed was over 300 MB, of which the libraries were about 240MB and executables were about 60MB. We tested each of these programs and ensured that they worked correctly. A subset of these tests is shown in Figure 7.

Correctness of Disassembly. Since testing explores only a fraction of program paths, we undertook a more complete evaluation of disassembly correctness. For this, we recompiled several large programs, including Firefox 5, GIMP-2.6 and glibc-2.13 to obtain the assembly code generated by the compiler. Specifically, we turned on the option `--listing-lhs-width=4 -alcdn` of GNU assembler to generate listing files containing both machine code and assembly. This was then compared with disassembly.

Note that multiple object files are combined by the linker to produce an executable or library. We intercept the linker `ld` to record address ranges in the code that correspond to each object file. This information is used to compare compiler-produced assembly for each object file with the corresponding part of the disassembler output.

Figure 6 shows the results of our disassembly testing. About 58MB of executable files including code and data, corresponding to a total of about 6M instructions have been tested, with no errors reported.

Testing Code Generated by Alternative Compilers. We applied our instrumentation to two programs compiled using LLVM. In particular, we used Clang 2.9 to compile two programs in the OpenSSH project, `ssh` and `scp`. Experiments shows that both LLVM generated `ssh` and `scp` function correctly when we used them to login to a remote server and copy a large file to/from the server.

7.2 CFI Effectiveness Evaluation

Figure 8 compares the AIR metric for bin-CFI with strict-CFI, reloc-CFI, bundle-CFI and instr-CFI. To calculate AIR of reloc-CFI, we recompiled SPEC2006 programs using “-g” and a linker option “-Wl,-emit-relocs” to retain all the relocations in executables. We can now calculate AIR from the description of reloc-CFI in Section 4.2 and Definition 1.

To calculate AIR for bundle-CFI, we recompiled SPEC2006 using the Native Client provided `gcc` and `g++` compilers. Since bundle-CFI restricts ICF targets to 32-byte boundaries, 31/32 of the compiled binary code is eliminated as ICF targets. However, the AIR number is smaller because the base is the original program size; programs compiled using Native Client tool-chain are larger due to reasons such as the need to introduce padding to align indirect targets at 32-byte boundaries.

Name	Reloc CFI	Strict CFI	Bin CFI	Bundle CFI	Instr CFI
perlbench	98.49%	98.44%	97.89%	95.41%	67.33%
bzip2	99.55%	99.49%	99.37%	95.65%	78.59%
gcc	98.73%	98.71%	98.34%	95.86%	80.63%
mcf	99.47%	99.37%	99.25%	95.91%	79.35%
gobmk	99.40%	99.40%	99.20%	97.75%	89.08%
hmmer	98.90%	98.87%	98.61%	95.85%	79.01%
sjeng	99.32%	99.30%	99.10%	96.22%	83.18%
libquantum	99.14%	99.07%	98.89%	95.96%	76.53%
h264ref	99.64%	99.60%	99.52%	96.25%	80.71%
omnetpp	98.26%	98.08%	97.68%	95.72%	82.03%
astar	99.18%	99.13%	98.95%	96.02%	78.00%
milc	98.89%	98.86%	98.65%	96.03%	79.74%
namd	99.65%	99.64%	99.59%	95.81%	76.37%
soplex	99.19%	99.10%	98.86%	95.50%	77.37%
povray	99.01%	98.99%	98.67%	95.87%	78.03%
lbm	99.60%	99.50%	99.46%	96.79%	80.92%
sphinx3	98.83%	98.80%	98.64%	96.06%	80.75%
average	99.13%	99.08%	98.86%	96.04%	79.27%

Figure 8: AIR metrics for SPEC CPU 2006.

7.3 Security Evaluation

7.3.1 Control-Flow Hijack Attacks

To evaluate control flow hijack defense, we used the RIPE [45] test suite. RIPE is a benchmark consisting of 850 distinct exploits including code injection, return-to-libc and ROP attacks. RIPE illustrated these attacks by building vulnerabilities into a small program. Ex-

	DEP disabled	DEP enabled
Original	520	140
CFI	90	90

Figure 9: Security Evaluation using RIPE

ploit code is also built into this program, so some of the challenges of developing exploits, e.g., knowing the right jump addresses, are not present. As such, techniques such as ASLR have no impact on RIPE. So, the only change we can experiment with is that of enabling or disabling DEP.

Originally, on Ubuntu 11.10 platform, 520 attacks survive with data execution prevention (DEP) disabled. With DEP enabled, 140 attacks survive. All of these attacks are return-to-libc attacks.

The 2nd row in Figure 9 shows bin-CFI could defeat 430 attacks including 380 code injection attacks and 50 return-to-libc attacks, even when DEP is disabled. In both scenarios, when DEP is enabled or disabled, however there are 90 function pointer overwrite attacks that survive in CFI.

Code injection attacks are defeated by CFI because global data, stack and heap are not allowed targets of ICF transfers. 50 out of 140 return-to-libc attacks are defeated because they overflow return addresses and try to redirect control flow to the libc functions and violate the policy of bin-CFI. Those attacks are defeated.

The function pointer overwrite attacks that succeed are somewhat of an artifact of RIPE design that includes exploit code within the victim program. Since pointers to exploit code are already taken in the program, they are identified as legitimate targets and permitted by our approach. If the same attacks were to be carried out against real programs, only a subset of them will succeed: those that overwrite function pointers with pointers to other local functions. In this subset of cases, previous CFI implementations (although not necessarily their formulations) would fail too, as they too permit any indirect call to reach any function whose address is taken.

7.3.2 ROP Attacks

We use the tool ROPGadget-v3.3[35], an ROP gadget generator/compiler, as our testing tool. It scans binaries to find useful gadgets for ROP attacks.

Figure 10 shows that CFI enforcement is effective, resulting in the elimination of the vast majority (93%) of gadgets in the original program. Moreover, there is little diversity in the gadgets found — the tool was able to find only the following gadgets:

- `mov constant, %eax; ret` (32.26%)
- `add offset, %esp; pop %ebx; ret` (27.42%)
- `add offset, %esp; ret` (19.35%)

- `mov (%esp), %ebx; ret` (14.52%)
- `xor %eax, %eax; ret` (5.65%)
- `pop %edx; pop %ecx; pop %ebx; ret` (0.81%)

There is little variety in these gadgets. Among other missing features, note the complete lack of useful arithmetic operations in the identified gadgets. As a result, the tool was unable to build even a single exploit using these gadgets

Name	Reloc CFI	Strict CFI	Bin CFI	Instr CFI
perlbench	96.62%	96.24%	93.23%	58.65%
bzip2	97.78%	95.56%	93.33%	44.44%
gcc	97.69%	97.69%	91.42%	66.67%
mcf	95.45%	90.91%	90.91%	36.36%
gobmk	98.84%	98.27%	97.69%	70.52%
hmmer	97.00%	96.00%	96.00%	58.00%
sjeng	92.75%	92.75%	91.30%	47.83%
libquantum	93.18%	90.91%	86.36%	40.91%
h264ref	98.26%	97.39%	96.52%	60.87%
omnetpp	97.12%	97.12%	93.42%	74.07%
astar	95.35%	93.02%	93.02%	46.51%
milc	95.77%	94.37%	90.14%	57.75%
namd	94.87%	92.31%	92.31%	53.85%
soplex	94.64%	93.75%	93.75%	54.46%
povray	96.75%	96.75%	95.45%	61.69%
lmb	94.12%	88.24%	88.24%	23.53%
sphinx3	95.00%	93.75%	92.50%	52.50%
average	95.95%	94.41%	92.68%	53.45%

Figure 10: Gadget elimination in different CFI implementation

7.4 Performance Evaluation

Our testbed consists of an Intel core-i5 2410m CPU with 4GB memory, running Ubuntu 11.10 (32-bit version), with glibc version 2.13. We used the SPEC 2006 CPU benchmark to evaluate both the runtime overhead and space overhead.

7.4.1 Runtime Overhead

Figure 11 shows the runtime overheads of CFI enforcement on SPEC CPU 2006 benchmarks. The average overhead for C programs is 4.29%. Due to C++ exception handling, VT.2 (Section 6.3) cannot be applied to C++ programs. As a result, the overhead for C++ programs increases to an average of 8.54%. omnetpp, soplex, and povray are particular contributors to this increased overhead. One way to bring these overheads down (to match the overhead for C-programs) is to update the exception handling metadata to use code addresses within instrumented code.

7.4.2 Space and Memory Overhead

Our instrumentation introduces a new code section that is on average 1.2 times the original code size. The new

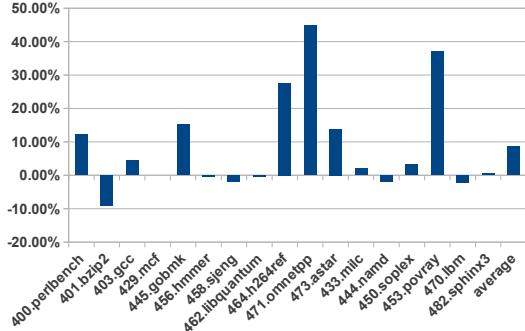


Figure 11: SPEC CPU2006 Benchmark Performance

data section introduced contains address translation table for indirect branch instructions. In total, the space overhead for bin-CFI is 139% over the original file size. Note that although the file size has increased, execution will be confined to the new code. Except in the case of programs that store read-only data in their code, other programs don't access their code even once. Hence the runtime memory overhead is unaffected by the presence of the original copy of code. Indeed, our measurements showed a very small increase in resident memory use (about 2.2% on average).

7.5 Limitations

Dynamic code. Since we rely on static transformation of binaries, any usage of dynamic code such as just-in-time compilation cannot be handled by bin-CFI. This also applies to any binary that modifies itself. These limitations are shared by most previous implementations of CFI.

Obfuscated code. Reliable static disassembly of obfuscated code is a challenging problem without satisfactory solutions. However, obfuscation is typically used on malware, whereas our target consists of benign (but possibly vulnerable) programs.

Return-into-libc attack. In general, CFI does not eliminate the threat of all return-to-libc attacks, a fact that holds true in our implementation as well.

Most return-into-libc fall into one of the two following types. The first type chains a sequence of library function calls, and relies on the semantics of these functions to perform attacks [28]. The second type relies on the side effects of library functions to realize Turing-complete ROP [41]. Both types rely heavily on returning to exported functions in glibc, and hence are defeated by bin-CFI. (Note that exported functions are excluded from allowable return targets by our policy.) However, it may be possible to construct return-to-libc attacks that make use of code pointers in glibc (or other shared libraries), or more generally, any address computed by our static anal-

ysis. These attacks could be mitigated by further tightening the policy for returns, improving the precision of static analysis, or both. We point out that even without these improvements, bin-CFI degrades return-to-libc attacks in much the same way as it degrades ROP attacks: it reduces the number of possible functions that can be used in an attack.

8 Related Work

8.1 ROP Attacks and Defenses

Return Oriented Programming (ROP) [38] is a powerful code reuse attack. It has become a very popular means to carry out successful attacks in spite of DEP. Although ROP was originally thought to be applicable primarily to CISC processors such as the x86, subsequent work has demonstrated their effectiveness on RISC architectures as well [9]. ROP attacks can target user programs as well as the kernel [19]. The introduction of JOP [10, 7] eliminates the need to use return instructions to effect ICF transfers, thereby defeating defenses that rely on the use of (repeated) returns [11, 14, 32].

Some of ROP defenses [31, 23] modify the code generation process to ensure that there are no useful gadgets in a generated binary. As they work at the level of code generation, they require source code. Rather than eliminating gadgets, some recent works [18, 43, 33] rely on fine-grained randomization that makes it difficult to find the location of useful gadgets. Instruction Location Randomization (ILR) [18] randomizes instruction locations, thereby making ROP hard. A benefit of their approach is that they can randomize return addresses, which significantly reduces the number of valid ICF targets, as return addresses constitute a majority of them. But this randomization can cause problems in large and complex binaries where a return instruction may be used for purposes other than returning from a call, e.g., PIC code data access, or to implement context-switching-like functionality.

A drawback of ILR is high space overhead. Binary Stirring (STIR) [43] solves this issue by randomizing basic blocks at load time using static rewriting. It achieves better runtime performance and reasonable space overhead. However, neither ILR nor Binary Stirring apply their work on libraries or large binaries. [33] uses static in-place randomization (IPR) to eliminate gadgets. The runtime overhead is almost zero, though the effectiveness depends on the target binary layout. In particular, a significant fraction of gadgets remain, thus limiting protection against ROP attacks.

While strong randomization could confuse attackers at runtime, and further reduce the number of usable gadgets, we have refrained from adding randomization to our technique for several reasons. First and foremost, we believe that one of principal reasons behind the success of

CFI is that it provides deterministic protection, thus laying a solid foundation for other protection mechanisms such as SFI or policy enforcement on untrusted code. Second, randomization defenses are already widely deployed in the form of ASLR and stack cookies. To the extent their randomization isn't defeated, they can provide excellent protection in conjunction with our CFI. If, on the other hand, we assume that randomization of ASLR can be defeated, then there is no good reason to believe that a randomization component added to a CFI technique won't be defeated either. Thirdly, the utility of randomization is increasingly called into question by advances in information leakage attacks. Recent exploits [37, 16] show that strong information leakage attack could help bypass ASLR with high entropy. Moreover, just-in-time code reuse attacks [39] discover gadgets using repeated information leakage attacks and are able to defeat even fine-grained code randomization.

8.2 Control Flow Integrity

Control-flow integrity (CFI) was introduced by Abadi et al [1]. The basic idea was to use a static analysis to compute a control-flow graph, and enforce it at runtime. Enforcement was based on matching constants (called IDs) between the source and target of each ICF transfer. However, due to difficulties in performing accurate points-to analysis, and because of so-called destination equivalence problem, their implementation resorts to coarse granularity enforcement, wherein any indirect call is permitted to target any function whose address is taken. Li et al. [22] implement a compiler based CFI that uses a similar policy for coarse-grained CFI. While they can also support finer-granularity CFI, this requires runtime profiling to compute possible targets of indirect calls, and can hence be prone to false positives.

Control-flow locking (CFL) [6] improves significantly on the performance of Abadi et al, while simultaneously tightening the policy, especially for returns. But this tighter policy poses challenges in the presence of indirect tail calls. Another difference between their work and ours is that they operate on assembly code generated by the compiler, whereas our work targets binaries.

MoCFI [13] presents a design and implementation of CFI for mobile platforms. The mobile environment presents a unique set of challenges, including an instruction set that does not have explicit returns, a closed platform (iOS), and so on. An important characteristic of their approach is that they aggressively prune possible targets of each ICF transfer. While this can provide better protection, it leads to false positives in some cases (e.g., when large jump tables are involved). In contrast, our approach emphasizes handling of large binaries, including shared libraries, that are not handled by their approach. We discussed how this requirement dictates the

use of coarser granularity CFI in our technique.

CCFIR [48], like the work presented in this paper, targets binaries. The main insight in their work is that most binaries on Windows support ASLR, which requires relocation information to be included in the binary. They leverage this information for accurate disassembly and static rewriting. Moreover, since relocation information effectively identifies all code pointers, they can avoid runtime address translation, which enables them to achieve better performance. The flipside of this performance improvement is that the technique can't be used on most UNIX systems, as UNIX binaries rarely contain the requisite relocations.

CFI has been used as the basis for untrusted code sandboxing. PittSFIeld [27] implements SFI on top of instruction bundling, a weaker CFI model. XFI [15] presents techniques that are based on CFI and SFI to confine untrusted code in shared-memory environments. Zeng et al [47] improve the performance of SFI using CFI and static analysis. Native client [46] is aimed at running native binaries securely in a browser context, and relies on instruction bundling. PittSFIeld, Native Client, and many other works [22, 3, 4, 42, 21, 36, 34, 20] that enforce CFI rely on compiler-provided information and even hardware support. In contrast, bin-CFI operates on COTS binaries without support from compiler, OS or hardware.

9 Conclusions

In this paper, we developed a notion of control-flow integrity that can be effectively enforced on binaries. We developed analysis techniques to compute possible ICF targets, and instrumentation techniques that limit ICF transfers to these targets. The resulting implementation defeats most common control-flow hijack attacks, and greatly reduces the number of possible gadgets for ROP attacks. We presented a robust implementation that scales to large binaries as well as complex, low-level libraries that include hand-coded assembly. Our technique is modular, supporting independent transformation of shared libraries. It also provides very good performance.

Our results realize one of central benefits of the CFI property, i.e., it can be applied to protect low-level code that is available only in the form of binaries. Although the lack of high-level information can degrade the precision of static analysis, our results demonstrate that the reduction is small; and overall, there is only a modest reduction in the strength of protection as compared to previous techniques that required source code, relocation information, or relied on compiler-based implementations.

10 Acknowledgements

We are very grateful to the developers of Katana, especially James Oakley for his quick and very helpful responses to our questions. Also we thank Edward Schwartz for his technique support.

References

- [1] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-flow integrity. In *the 12th ACM conference on Computer and communications security (CCS)*, 2005.
- [2] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-flow integrity principles, implementations, and applications. *ACM Transactions on Information and System Security (TISSEC)*, (1), Nov. 2009.
- [3] P. Akrítidis, C. Cadar, C. Raiciu, M. Costa, and M. Castro. Preventing memory error exploits with WIT. In *the 29th IEEE Symposium on Security and Privacy (Oakland)*, 2008.
- [4] J. Ansel, P. Marchenko, U. Erlingsson, E. Taylor, B. Chen, D. L. Schuff, D. Sehr, C. L. Biffle, and B. Yee. Language-independent sandboxing of just-in-time compilation and self-modifying code. In *the 32nd ACM SIGPLAN conference on Programming language design and implementation (PLDI)*, 2011.
- [5] S. Bhatkar, D. C. DuVarney, and R. Sekar. Address obfuscation: an efficient approach to combat a board range of memory error exploits. In *the 12th conference on USENIX Security Symposium*, 2003.
- [6] T. Bletsch, X. Jiang, and V. Freeh. Mitigating code-reuse attacks with control-flow locking. In *the 27th Annual Computer Security Applications Conference (AC-SAC)*, 2011.
- [7] T. Bletsch, X. Jiang, V. W. Freeh, and Z. Liang. Jump-oriented programming: a new class of code-reuse attack. In *the 6th ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, 2011.
- [8] D. L. Bruening. *Efficient, transparent, and comprehensive runtime code manipulation*. PhD thesis, MIT, 2004.
- [9] E. Buchanan, R. Roemer, H. Shacham, and S. Savage. When good instructions go bad: generalizing return-oriented programming to RISC. In *the 15th ACM conference on Computer and communications security (CCS)*, 2008.
- [10] S. Checkoway, L. Davi, A. Dmitrienko, A.-R. Sadeghi, H. Shacham, and M. Winandy. Return-oriented programming without returns. In *the 17th ACM conference on Computer and communications security (CCS)*, 2010.
- [11] P. Chen, H. Xiao, X. Shen, X. Yin, B. Mao, and L. Xie. DROP: Detecting return-oriented programming malicious code. In *the 5th International Conference on Information Systems Security (ICISS)*, 2009.
- [12] C. Cowan, C. Pu, D. Maier, H. Hintony, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang. StackGuard: automatic adaptive detection and prevention of buffer-overflow attacks. In *the 7th conference on USENIX Security Symposium*, 1998.
- [13] L. Davi, R. Dmitrienko, M. Egele, T. Fischer, T. Holz, R. Hund, S. Nrnberger, and A. reza Sadeghi. MoCFI: A framework to mitigate control-flow attacks on smartphones. In *the 19th Network and Distributed System Security Symposium (NDSS)*, 2012.
- [14] L. Davi, Ahmad-Reza Sadeghi, and M. Winandy. ROPdefender: a detection tool to defend against return-oriented programming attacks. In *the 6th ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, 2011.
- [15] U. Erlingsson, S. Valley, M. Abadi, M. Vrable, M. Budiu, and G. C. Necula. XFI: software guards for system address spaces. In *the 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2006.
- [16] C. Evans. Exploiting 64-bit linux like a boss. <http://scarybeastsecurity.blogspot.com/2013/02/exploiting-64-bit-linux-like-boss.html>.
- [17] M. Frantzen and M. Shuey. StackGhost: Hardware facilitated stack protection. In *the 10th conference on USENIX Security Symposium*, 2001.
- [18] J. Hiser, A. Nguyen-Tuong, M. Co, M. Hall, and J. W. Davidson. ILR: Where'd my gadgets go? In *the 33th IEEE Symposium on Security and Privacy (Oakland)*, 2012.
- [19] R. Hund, T. Holz, and F. C. Freiling. Return-oriented rootkits: bypassing kernel code integrity protection mechanisms. In *the 18th conference on USENIX security symposium*, 2009.
- [20] M. Kayaalp, M. Ozsoy, N. Abu-Ghazaleh, and D. Ponomarev. Branch regulation: low-overhead protection from code reuse attacks. In *the 39th Annual International Symposium on Computer Architecture (ISCA)*, 2012.
- [21] V. Kiriansky, D. Bruening, and S. P. Amarasinghe. Secure execution via program shepherding. In *the 11th conference on USENIX Security Symposium*, 2002.
- [22] J. Li, Z. Wang, T. Bletsch, D. Srinivasan, M. Grace, and X. Jiang. Comprehensive and efficient protection of kernel control data. *IEEE Transactions on Information Forensics and Security*, (4), Dec. 2011.
- [23] J. Li, Z. Wang, X. Jiang, M. Grace, and S. Bahram. Defeating return-oriented rootkits with "return-less" kernels. In *the 5th European conference on Computer systems (EuroSys)*, 2010.
- [24] the PaX team. Address space layout randomization. <http://pax.grsecurity.net/docs/aslr.txt>, 2001.
- [25] Tool Interface Standard. Executable and linking format (ELF) specification. <http://www.uclibc.org/docs/elf.pdf>, 1995.
- [26] UNIX International Programming Languages SIG. DWARF debugging information format. <http://www.dwarfstd.org/doc/dwarf-2.0.0.pdf>, 1993.

- [27] S. McCamant and G. Morrisett. Evaluating SFI for a CISC architecture. In *the 15th conference on USENIX Security Symposium*, 2006.
- [28] Nergal. The advanced return-into-libc exploits: PaX case study. *Phrack Magazine*, 2001.
- [29] J. Oakley and S. Bratus. Exploiting the hard-working DWARF: trojan and exploit techniques with no native executable code. Technical report, Computer Science Department, Dartmouth College, 2011.
- [30] J. Oakley and S. Bratus. Exploiting the hard-working DWARF: trojan and exploit techniques with no native executable code. In *the 5th USENIX conference on Offensive technologies (WOOT)*, 2011.
- [31] K. Onarlioglu, L. Bilge, A. Lanzi, D. Balzarotti, and E. Kirda. G-Free: defeating return-oriented programming through gadget-less binaries. In *the 26th Annual Computer Security Applications Conference (ACSAC)*, 2010.
- [32] V. Pappas. kBouncer: Efficient and transparent ROP mitigation. Technical report, Columbia University, 2012.
- [33] V. Pappas, M. Polychronakis, and A. D. Keromytis. Smashing the gadgets: Hindering return-oriented programming using in-place code randomization. In *the 33th IEEE Symposium on Security and Privacy (Oakland)*, 2012.
- [34] A. Prakash, H. Yin, and Z. Liang. Enforcing system-wide control flow integrity for exploit detection and diagnosis. In *the 8th ACM SIGSAC symposium on Information, computer and communications security (ASIACCS)*, 2013.
- [35] J. Salwan. ROPGadget. <http://shell-storm.org/project/ROPgadget>.
- [36] D. Sehr, R. Muth, C. Biffle, V. Khimenko, E. Pasko, K. Schimpf, B. Yee, and B. Chen. Adapting software fault isolation to contemporary cpu architectures. In *the 19th conference on USENIX Security Symposium*, 2010.
- [37] F. J. Serna. CVE-2012-0769, the case of the perfect info leak, 2012.
- [38] H. Shacham. The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86). In *the 14th ACM conference on Computer and communications security (CCS)*, 2007.
- [39] K. Z. Snow, F. Monroe, L. Davi, A. Dmitrienko, C. Liebchen, and A.-R. Sadeghi. Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization. In *the 34th IEEE Symposium on Security and Privacy*, 2013.
- [40] R. Strackx, Y. Younan, P. Philippaerts, F. Piessens, S. Lachmund, and T. Walter. Breaking the memory secrecy assumption. In *the 2nd European Workshop on System Security (EUROSEC)*, 2009.
- [41] M. Tran, M. Etheridge, T. Bleisch, X. Jiang, V. Freeh, and P. Ning. On the expressiveness of return-into-libc attacks. In *the 14th international conference on Recent Advances in Intrusion Detection (RAID)*, 2011.
- [42] Z. Wang and X. Jiang. HyperSafe: A lightweight approach to provide lifetime hypervisor control-flow integrity. In *the 31th IEEE Symposium on Security and Privacy (Oakland)*, 2010.
- [43] R. Wartell, V. Mohan, K. W. Hamlen, and Z. Lin. Binary stirring: self-randomizing instruction addresses of legacy x86 binary code. In *the 19th ACM conference on Computer and communications security (CCS)*, 2012.
- [44] wikipedia. Open addressing hashing. http://en.wikipedia.org/wiki/Open_addressing, 2012.
- [45] J. Wilander, N. Nikiforakis, Y. Younan, M. Kamkar, and W. Joosen. RIPE: runtime intrusion prevention evaluator. In *the 27th Annual Computer Security Applications Conference (ACSAC)*, 2011.
- [46] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar. Native client: A sandbox for portable, untrusted x86 native code. In *the 30th IEEE Symposium on Security and Privacy (Oakland)*, 2009.
- [47] B. Zeng, G. Tan, and G. Morrisett. Combining control-flow integrity and static analysis for efficient and validated data sandboxing. In *the 18th ACM conference on Computer and communications security (CCS)*, 2011.
- [48] 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 *the 34th IEEE Symposium on Security and Privacy*, 2013.
- [49] D. D. Zovi. Practical return-oriented programming. Technical report, SOURCE, 2010.

Native x86 Decompilation using Semantics-Preserving Structural Analysis and Iterative Control-Flow Structuring

Edward J. Schwartz

Carnegie Mellon University

Maverick Woo

Carnegie Mellon University

JongHyup Lee

Korea National University of Transportation

David Brumley

Carnegie Mellon University

Abstract

There are many security tools and techniques for analyzing software, but many of them require access to source code. We propose leveraging *decompilation*, the study of recovering abstractions from compiled code, to apply existing source-based tools and techniques to compiled programs. A decompiler should focus on two properties to be used for security. First, it should recover abstractions as much as possible to minimize the complexity that must be handled by the security analysis that follows. Second, it should aim to recover these abstractions correctly.

Previous work in control-flow structuring, an abstraction recovery problem used in decompilers, does not provide either of these properties. Specifically, existing structuring algorithms are not *semantics-preserving*, which means that they cannot safely be used for decompilation without modification. Existing structural algorithms also miss opportunities for recovering control flow structure. We propose a new structuring algorithm in this paper that addresses these problems.

We evaluate our decompiler, Phoenix, and our new structuring algorithm, on a set of 107 real world programs from GNU coreutils. Our evaluation is an order of magnitude larger than previous systematic studies of end-to-end decompilers. We show that our decompiler outperforms the *de facto* industry standard decompiler Hex-Rays in correctness by 114%, and recovers 30 \times more control-flow structure than existing structuring algorithms in the literature.

1 Introduction

Security analyses are often faster and easier when performed on source code rather than on binary code. For example, while the runtime overhead introduced by *source-based* taint checkers can be as low as 0.65% [12], the overhead of the fastest *binary-based* taint checker is over 150% [8]. In addition, many security analyses described

in the literature assume access to source code. For instance, there are numerous source-based static vulnerability finding tools such as KINT [40], RICH [9], and Coverity [6], but equivalent binary-only tools are scarce.

In many security scenarios, however, access to source code is simply not a reasonable assumption. Common counterexamples include analyzing commercial off-the-shelf software for vulnerabilities and reverse engineering malware. The traditional approach in security has been to directly apply some form of low-level binary analysis that does not utilize source-level abstractions such as types and functions [5, 7, 10, 24]. Not surprisingly, reasoning at such a low level causes binary analysis to be more complicated and less scalable than source analysis.

We argue that *decompilation* is an attractive alternative to traditional low-level binary-based techniques. At its surface, decompilation is the recovery of a program’s source code given only its binary. Underneath, decompilation consists of a collection of *abstraction recovery* mechanisms such as indirect jump resolution, control flow structuring, and data type reconstruction, which recover high-level abstractions that are not readily available in the binary form. Our insight is that by reusing these mechanisms, we can focus our research effort on designing security analyses that take advantage of such abstractions for accuracy and efficiency. In fact, when taken to an extreme, we may even use decompilation to leverage an existing source-based tool—be it a vulnerability scanner [27], a taint engine [12], or a bug finder [6]—by applying it to the decompiled program code.

Of course, decompilation is also extremely beneficial in situations where manual analysis is required. For example, practitioners often reverse-engineer program binaries to understand proprietary file formats, study vulnerabilities fixed in patches, and determine the exploitability of crashing inputs. Arguably, any one of these tasks becomes easier when given access to source code.

Unfortunately, current research in decompilation does not directly cater to the needs of many security applica-

tions. A decompiler should focus on two properties to be used for security. First, it should recover abstractions as much as possible to minimize the complexity that must be handled by the actual security analysis that follows. Second, it should aim to recover these abstractions correctly. As surprising as it may sound, previous work on decompilation almost never evaluated correctness. For example, Cifuentes et al.’s pioneering work [13] and numerous subsequent works [11, 14, 16, 39] all measured either how much smaller the output C code was in comparison to the input assembly, or with respect to some subjective readability metric.

In this paper, we argue that source can be recovered in a principled fashion. As a result, security analyses can better take advantage of existing source-based techniques and tools both in research and practice. Security practitioners can also recover correct, high-level source code, which is easier to reverse engineer. In particular, we propose techniques for building a correct decompiler that effectively recovers abstractions. We implement our techniques in a new end-to-end binary-to-C decompiler called *Phoenix*¹ and measure our results with respect to correctness and high-level abstraction recovery.

Phoenix makes use of existing research on principled abstraction recovery where possible. Source code reconstruction requires the recovery of two types of abstractions: data type abstractions and control flow abstractions. Recent work such as TIE [28], REWARDS [29], and Howard [38] have largely addressed principled methods for recovering data types. In this paper, we investigate new techniques for recovering high-level control structure.

1.1 The Phoenix Structural Analysis Algorithm

Previous work has proposed mechanisms for recovering high-level control flow based on the structural analysis algorithm and its predecessors [20, 23, 39]. However, they are problematic because they (1) do not feature a correctness property that is necessary to be safely used for decompilation, and (2) miss opportunities for recovering control flow structure. Unfortunately, these problems can cause a security analysis using the recovered control structures to become unsound or scale poorly. These problems motivated us to create our own control flow structuring algorithm for *Phoenix*. Our algorithm is based on structural analysis, but avoids the problems we identified in earlier

work. In particular, we identify a new property that structural analysis algorithms should have to be safely used for decompilation, called *semantics-preservation*. We also propose *iterative refinement* as a strategy for recovering additional structure.

Semantics Preservation Structural analysis [32, p. 203] is a control flow structuring algorithm that was originally invented to help accelerate data flow analysis. Later, decompiler researchers adapted this algorithm to reconstruct high-level control flow structures such as if-then-else and do-while from a program’s control flow graph (see §2.1). We propose that structuring algorithms should be *semantics-preserving* to be safely used in decompilers. A structuring algorithm is semantics-preserving if it always transforms the input program to a functionally equivalent program representation. Semantics-preservation is important for security analyses to ensure that the analysis of the structured program also applies to the original binary. Surprisingly, we discovered that common descriptions of structural analysis algorithms are *not* semantics-preserving. For example, in contrast to our natural loop schema in Table 4, other algorithms employ a schema that permits out-going edges (e.g., see [20, Figure 3]). This can lead to incorrect decompilation, such as the example in Figure 3. We demonstrate that fixing this and other schemas to be semantics-preserving increases the number of utilities that *Phoenix* is able to correctly decompile by 30% (see §4).

Iterative Refinement When structural analysis algorithms encounter unstructured code, they stop recovering structure in that part of the program. Our algorithm instead iteratively refines the graph to continue making progress. The basic idea is to select an edge from the graph that is preventing the algorithm from making progress, and represent it using a goto in the decompiled output. This may seem counter-intuitive, since more gotos implies less structure recovered. However, by removing the edge from the graph the algorithm can make more progress, and recover more structure. We also show how refinement enables the recovery of switch structures. In our evaluation, we demonstrate that iterative refinement recovers 30× more structure than structural analysis algorithms that do not employ iterative refinement (see §4). Missed structure is problematic in security applications because it can hamper syntax-based deductions—such as the fact that body will execute ten times in `for (i=0; i<10; i++) {body;}`. Control flow structure is also used to explicitly accelerate some analyses (e.g., data flow analysis [2, 17]), and failure to recover structure can undermine the performance of these

¹*Phoenix* is named in honor of the famous “Dragon Book” [1] on compilers. According to Chinese mythology, the phoenix is a supreme bird that complements the dragon (compilation). In Greek mythology, the phoenix can be reborn from the ashes of its predecessor. Similarly, a decompiler can recover source code and abstractions from the compiled form of a binary, even when these artifacts seem to have been destroyed.

algorithms. Unfortunately, even recent structuring algorithms such as the one in [20, Algorithm 2] do not employ refinement in their descriptions, and thus can fail to recover structure on problematic program sections.

Contributions:

1. We propose a new structural analysis algorithm that addresses two shortcomings of existing structural analysis algorithms: (1) they can cause incorrect decompilation, and (2) they miss opportunities to recover control flow structure. Our algorithm uses *iterative refinement* to recover additional structure, including switches. We also identify a new property, *semantics-preservation*, that control flow structuring algorithms should have to be safely used in decomilers. We implement and test our algorithm in our new end-to-end binary-to-C decompiler, Phoenix.
2. We demonstrate that our proposed structural analysis algorithm recovers $30\times$ more control-flow structure than existing research in the literature [20, 32, 36], and 28% more than the *de facto* industry standard decompiler Hex-Rays [23]. Our evaluation uses the 107 programs in GNU coreutils as test cases, and is an order of magnitude larger than any other systematic end-to-end decompiler evaluation to date.
3. We propose *correctness* as a new metric for evaluating decomilers. Although previous work has measured the correctness of individual decompiler components (e.g., type recovery [28] and structure recovery [20]), surprisingly the correctness of a decompiler as a whole has never been measured. We show in our evaluation that Phoenix successfully decompiled over $2\times$ as many programs that pass the coreutils test suite as Hex-Rays.

2 Overview

Any end-to-end decompiler such as Phoenix is necessarily a complex project. This section aims to give a high-level description of Phoenix. We will start by reviewing several background concepts and then present an overview of each of the four stages of Phoenix. The remainder of the paper focuses on our novel structural analysis algorithm, which is Phoenix’s third stage.

2.1 Background

Control Flow Analysis A *control flow graph* (CFG) of a program P is a directed graph $G = (N, E, n_s, n_e)$. The node set N contains basic blocks of program statements in P . Each basic block must have exactly one entrance at the beginning and one exit at the end. Thus, each time the

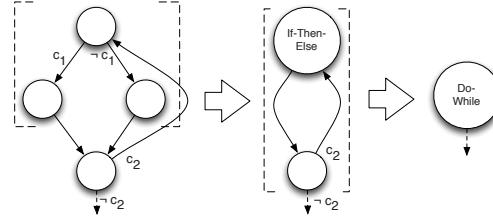


Figure 1: Example of structural analysis.

first instruction of a basic block is executed, the remaining instructions must also be executed in order. The nodes $n_s \in N$ and $n_e \in N$ represent the entrance and the exit basic blocks of P respectively. An edge (n_i, n_j) exists in the edge set E if $n_i \in N$ may transfer control to $n_j \in N$. Each edge (n_i, n_j) has a label ℓ that specifies the logical predicate that must be satisfied for n_i to transfer control to n_j .

Domination is a key concept in control flow analysis. Let n be any node. A node d dominates n , denoted $d \mathbf{dom} n$, iff every path in G from n_s to n includes d . Furthermore, every node dominates itself. A node p post-dominates n , denoted $p \mathbf{pdom} n$, iff every path in G from n to n_e includes p . For any node n other than n_s , the immediate dominator of n is the unique node d that strictly dominates n (i.e., $d \mathbf{dom} n$ and $d \neq n$) but does not strictly dominate any other node that strictly dominates n . The immediate post-dominator of n is defined similarly.

Loops are defined through domination. An edge (s, d) is a *back edge* iff $d \mathbf{dom} s$. Each back edge (s, d) defines a *natural loop*, whose header is d . The natural loop of a back edge (s, d) is the union of d and the set of nodes that can reach s without going through d .

Structural Analysis *Structural analysis* is a control flow structuring algorithm for recovering high-level control flow structure such as if-then-else constructs and loops. Intriguingly, such an algorithm has uses in both compilation (during optimization) and decompilation (to recover abstractions). At a high level, structural analysis matches a set of region *schemas* over the CFG by repeatedly visiting its nodes in post-order. Each schema describes the shape of a high-level control structure such as if-then-else. When a match is found, all nodes matched by the schema are *collapsed* or *reduced* into a single node that represents the schema matched. For instance, Figure 1 shows the progression of structural analysis on a simple example from left to right, assuming that the topmost node is being visited. In the initial (leftmost) graph, the top three nodes match the shape of an if-then-else. Structural analysis therefore reduces these nodes into a single node that is explicitly labeled as an if-then-else region in the middle graph. This graph is then further reduced into

a do-while loop. A decompiler would use this sequence of reductions and infer the control flow structure: `do { if (c1) then {...} else {...} } while (c2).`

Once no further matches can be found, structural analysis starts reducing acyclic and cyclic subgraphs into *proper regions* and *improper regions*, respectively. Intuitively, both of these regions indicate that no high-level structure can be identified in that subgraph and thus `goto` statements will be emitted to encode the control flow. A key topic of this paper is how to build a modern structural analysis algorithm that can *refine* such regions so that more high-level structure can be recovered.

SESS Analysis and Tail Regions Vanilla structural analysis cannot recognize loops containing common C constructs such as `break` and `continue`. For instance, structural analysis would fail to structure the loop

```
while (...) { if (...) { body; break; } }.
```

Engel et al. [20] proposed the SESS (single exit single successor) analysis to identify regions that have multiple exits (using `break` and `continue`) but share a unique successor. Such exits can be converted into a *tail region* that represents the equivalent control flow construct. In the above example, `body` would be reduced to a `break` tail region. Without tail regions, structural analysis stops making progress when reasoning about loops containing multiple exits.

Although the SESS analysis was proposed to help address this problem, the core part of the algorithm, the detection of tail regions, is left unspecified [20, Algorithm 2, Line 15]. We implemented SESS analysis as closely to the paper as possible, but noticed that our implementation often stopped making progress *before* SESS analysis was able to produce a tail region. This can occur when regions do not have an unambiguous successor, or when loop bodies are too complex. Unfortunately, no structure is recovered for these parts of the program. This problem motivated the iterative refinement technique of our algorithm, which we describe in §3.

2.2 System Overview

Figure 2 shows the high-level overview of the approach that Phoenix takes to decompile a target binary. Like most previous work, Phoenix uses a number of stages, where the output of stage i is the input to stage $i+1$. Phoenix can fail to output decompiled source if any of its four stages fails. For this reason we provide an overview of each stage in this section. The first two stages are based on existing implementations. The last two use novel techniques and implementations developed specifically for Phoenix.

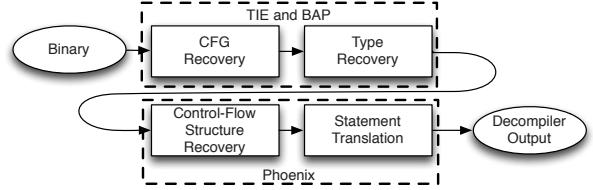


Figure 2: Decompilation flow of Phoenix. Phoenix contains new implementations for control flow recovery and statement translation.

<i>edge</i>	$::=$	<i>exp</i>
<i>vertex</i>	$::=$	<i>stmt</i> [*]
<i>stmt</i>	$::=$	<i>var := exp assert exp addr address</i>
<i>exp</i>	$::=$	<i>load(exp, exp, exp, τ_{reg})</i> <i> store(exp, exp, exp, exp, τ_{reg})</i> <i> exp op exp var lab(string) integer</i> <i> cast(cast kind, τ_{reg}, exp)</i>

Table 1: An abbreviated syntax of the BAP IL used to label control flow graph vertices and edges.

2.3 Stages I and II—Existing Work

Control Flow Graph Recovery The first stage parses the input binary’s file format, disassembles the binary, and creates a control flow graph (CFG) for each function. At a high level, a control flow graph is a program representation in which vertices represent basic blocks, and edges represent possible control flow transitions between blocks. (See §2.1 for more detail.) While precisely identifying binary code in an executable is known to be hard in the general case, current algorithms have been shown to work well in practice [4, 5, 24, 25].

There are mature platforms that already implement this step. We use the CMU Binary Analysis Platform (BAP) [10]. BAP lifts sequential x86 assembly instructions in the CFG into an intermediate language called BIL, whose syntax is shown in Table 1 (see [10]). As we will see, the end goal of Phoenix is to decompile this language into the high-level language shown in Table 2.

Variable and Type Recovery The second stage recovers individual variables from the binary code, and assigns them types. Phoenix uses TIE [28] to perform this task. TIE runs Value Set Analysis (VSA) [4] to recover variable locations. TIE then uses a static, constraint-based type inference system similar to the one used in the ML programming language [31]. Roughly speaking, each statement imposes some constraints on the type of variables involved. For example, an argument passed to a function that expects an argument of type T should be of type T , and the denominator in a division must be an

integer and not a pointer. The constraints are then solved to assign each variable a type.

2.4 Stage III—Control-Flow Structure Recovery

The next stage recovers the high-level control flow structure of the program. The input to this stage is an assembly program in CFG form. The goal is to recover high-level, structured control flow constructs such as loops, if-then-else and switch constructs from the graph representation. A program or construct is *structured* if it does not utilize *gos*. Structured program representations are preferred because they help scale program analysis [32] and make programs easier to understand [19]. The process of recovering a structured representation of the program is sometimes called *control flow structure recovery* or *control flow structuring* in the literature.

Although *control flow structure recovery* is similar in name to *control flow graph recovery* (stage I), the two are very different. Control flow graph recovery starts with a binary program, and produces a control flow graph representation of the program as output. Control flow structure recovery takes a control flow graph representation as input, and outputs the high-level control flow structure of the program, for instance:

```
while (...) { if (...) {...} }.
```

The rest of this paper will only focus on control flow structuring and not control flow graph reconstruction.

Structural analysis is a control flow structuring algorithm that, roughly speaking, matches predefined graph schemas or patterns to the control flow constructs that create the patterns [32]. For example, if a structural analysis algorithm identifies a diamond-shape in a CFG, it outputs an if-then-else construct, because if-then-else statements create diamond-shaped subgraphs in the CFG.

However, using structural analysis in a decompiler is not straightforward. We initially tried implementing the most recent algorithm in the literature [20] in Phoenix. We discovered that this algorithm, like previous algorithms, can (1) cause incorrect decompilation, and (2) miss opportunities for recovering structure. These problems motivated us to develop a new structural analysis algorithm for Phoenix which avoids these pitfalls. Our algorithm has two new features. First, our algorithm employs iterative refinement to recover more structure than previous algorithms. Our algorithm also features semantics-preserving schemas, which allows it to be safely used for decompilation. These topics are a primary focus of this paper, and we discuss them in detail in §3.

```

prog ::= (varinfo*, func*)
func ::= (string, varinfo, varinfo, stmt*)
stmt ::= var := exp | Goto(exp) | If exp then stmt else stmt
       | While(exp, stmt) | DoWhile(stmt, exp)
       | For(stmt, exp, stmt)
       | Sequence(stmt*)
       | Switch(exp, stmt*)
       | Case(exp, stmt)
       | Label(string)
       | Nop

```

Table 2: An abbreviated syntax of the HIL.

2.5 Stage IV—Statement Translation and Outputting C

The input to the next stage of our decompiler is a CFG annotated with structural information, which loosely maps each vertex in the CFG to a position in a control construct. What remains is to translate the BIL statements in each vertex of the CFG to a high-level language representation called HIL. Some of HIL’s syntax is shown in Table 2.

Although most statements are straightforward to translate, some require information gathered in prior stages of the decompiler. For instance, to translate function calls, we use VSA to find the offset of the stack pointer at the call site, and then use the type signature of the called function to determine how many arguments should be included. We also perform optimizations to make the final source more readable. There are two types of optimizations. First, similar to previous work, we perform optimizations to remove redundancy such as dead-code elimination [13]. Second, we implement optimizations that improve readability, such as untiling.

During compilation a compiler uses a transformation called *tiling* to reduce high-level program statements into assembly statements. At a high level, tiling takes as input an abstract syntax tree (AST) of the source language and produces an assembly program by covering the AST with semantically equivalent assembly statements. For example, given:

```
x = (y+z)/w
```

Tiling would first cover the expression $y + z$ with the add instruction, and then the division with the `div` instruction. Tiling will typically produce many assembly instructions for a single high-level statement.

Phoenix uses an *untiling* algorithm to improve readability. Untiling takes several statements and outputs an equivalent high-level source statement. For instance, at a low-level, `High1[a&b]` means to extract the most significant bit from bitwise-anding a with b . This may not seem like a common operation used in C, but it is equivalent to

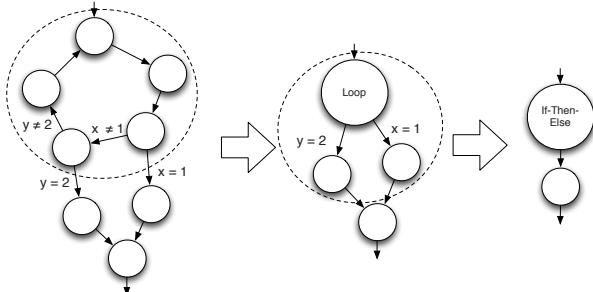


Figure 3: An example of how structural analysis can fail without semantics-preservation.

the high-level operation of computing $a <_s 0 \&\& b <_s 0$ (i.e., both a and b are less than zero when interpreted as signed integers). Phoenix uses about 20 manually crafted untiring patterns to simplify instructions emitted by gcc’s code generator. These patterns only improve the readability of the source output, and do not influence correctness or control-flow structure recovery. The output of the statement translation phase is a HIL program.

The final stage in Phoenix is an analysis that takes the HIL representation of the program as input. In this paper, we use an analysis that translates HIL into C, in order to test Phoenix as a binary-to-C decompiler.

3 Semantics-Preserving Structural Analysis and Iterative Control-Flow Structuring

In this section we describe our proposed structural analysis algorithm. Our algorithm builds on existing work by adding iterative refinement and semantics-preserving schemas. Before we discuss the details of our algorithm, we highlight the importance of these additions.

Semantics Preservation Structural analysis was originally invented to scale data flow analysis by summarizing the reachability properties of a program’s CFG. Later, decompiler researchers adapted structural analysis and its predecessor, interval analysis, to recover the control flow structure of decompiled programs [15, 23].

Unfortunately, structural analysis can identify control flow that is consistent with a graph’s reachability, but is inconsistent with the graph’s semantics.

Such an error from structural analysis is demonstrated in Figure 3. Structural analysis would identify the loop in the leftmost graph and reduce it to a single node representing the loop, thus producing the diamond-shaped graph shown in the middle. This graph matches the schema for an if-then-else region, which would also be reduced to a single node. Finally, the two remaining nodes would

then be reduced to a sequence node (not shown), at which point structural analysis is finished. This would be correct for data flow analysis, which only depends on reachability. However, the first node reduction is *not* semantics-preserving. This is easy to see for the case when both $x = 1$ and $y = 2$ hold. In the original graph, the first loop exit would be taken, since $x = 1$ matches the first exit edge’s condition. However, in the middle graph, both exit edges can be taken.

Such discrepancies are a problem in security, because they can unintentionally cause unsoundness in analyses. For example, an otherwise sound bug checker, when applied to the program in Figure 3, could state that a bug is present, even if the original program had no bugs.

To avoid unintentional unsoundness, a structural analysis algorithm should preserve the semantics of a CFG during each reduction. Otherwise the recovered control flow structure can become inconsistent with the actual control flow in the binary. Most schemas in structural analysis [32, p. 203] preserve semantics, but the natural loop schema is one that does not. A natural loop is a generalized definition of a single-entrance loop that can have multiple exits. The loop in Figure 3 is a natural loop, for example, because it has one entrance and two exits. We demonstrate that fixing the schemas in our algorithm to be semantics-preserving increases the number of utilities Phoenix correctly decompiles by 30% (see §4). We describe these modifications in the upcoming sections.

Iterative Refinement At a high level, *refinement* is the process of removing an edge from a CFG by emitting a *goto* in its place, and *iterative refinement* refers to the repeated application of refinement until structuring can progress. This may seem counter-intuitive, since adding a *goto* seems like it would *decrease* the amount of structure recovered. However, the removal of a *carefully-chosen* edge can potentially allow a schema to match the refined CFG, thus enabling the recovery of additional structure. (We describe which edges are removed in the following sections.) The alternative to refinement is to recover no structure for problematic parts of the CFG. We show that Phoenix emits 30× more *gositos* (from 40 to 1,229) when iterative refinement is disabled.

Recovering structure is important for two reasons. First, structuredness has been shown to help scale program analysis in general [32]. In addition, some analyses use syntactic patterns to find facts, which relies on effective structure recovery. For example, a bug checker might conclude that there is no buffer overflow in

```
char b[10];
int i = 0;
while (i < 10) {
    b[i] = 0;
    i++;
}
```

by syntactically discovering the induction variable i and the loop invariant $i < 10$. If the structuring algorithm does not recover the while loop, and instead represents this loop using gotos, the bug checker could be unable to reason that the loop is safe, and output a false positive.

3.1 Algorithm Overview

We focus on the novel aspects of our algorithm in this paper and refer readers interested in any structural analysis details elided to standard sources [32, p. 203].

Like vanilla structural analysis, our algorithm visits nodes in post-order in each iteration. Intuitively, this means that all descendants of a node will be visited (and hence had the chance to be reduced) before the node itself. The algorithm’s behavior when visiting node n depends on whether the region at n is cyclic (has a loop) or not. For an acyclic region, the algorithm tries to match the subgraph at n to one of the acyclic schemas (§3.2). If there is no match, and the region is a switch candidate, then it attempts to refine the region at n into a switch region (§3.4). If n is cyclic, the algorithm compares the region at n to the cyclic schemas (§3.5). If this fails, it refines n into a loop (§3.6). If neither matching or refinement make progress, the current node n is then skipped for the current iteration of the algorithm. If there is an iteration in which *all* nodes are skipped, i.e., the algorithm makes no progress, then the algorithm employs a last resort refinement (§3.7) to ensure that progress can be made.

3.2 Acyclic Regions

The acyclic region types supported by Phoenix correspond to the acyclic control flow operators in C: sequences, ifs, and switches. The schemas for these regions are shown in Table 3. For example, the $\text{Seq}[n_1, \dots, n_k]$ region contains k regions that always execute in the listed sequence. $\text{IfThenElse}[c, n, n_t, n_f]$ denotes that n_t is executed after n when condition c holds, and otherwise n_f is executed.

Our schemas match both shape and the boolean predicates that guard execution of each node, to ensure semantics preservation. These conditions are implicitly described using meta-variables in Table 3, such as c and $\neg c$. The intuition is that shape alone is not enough to distinguish which control structure should be used in decompilation. For instance, a switch for cases $x = 2$ and $x = 3$ can have the diamond shape of an if-then-else, but we would not want to mistake a switch for an if-then-else because the semantics of if-then-else requires the outgoing conditions to be inverses.

3.3 Tail Regions and Edge Virtualization

When no subgraphs in the CFG match known schemas, the algorithm is stuck and the CFG must be refined before more structure can be recovered. The insight behind *refinement* is that removing an edge from the CFG may allow a schema to match, and *iterative refinement* refers to the repeated application of refinement until a match is possible. Of course, each edge in the CFG represents a possible control flow, and we must represent this control flow in some other way to preserve the program semantics. We call removing the edge in a way that preserves control flow *virtualizing* the edge, since the decompiled program behaves as if the edge was present, even though it is not.

In Phoenix, we virtualize an edge by collapsing the source node of the edge into a tail region (see §2.1). Tail regions explicitly denote that there should be a control transfer at the end of the region. For instance, to virtualize the edge (n_1, n_2) , we remove the edge from the CFG, insert a fresh label l at the start of n_2 , and collapse n_1 to a tail region that denotes there should be a `goto l` statement at the end of region n_1 . Tail regions can also be translated into `break` or `continue` statements when used inside a switch or loop. Because the tail region explicitly represents the control flow of the virtualized edge, it is safe to remove the edge from the graph and ignore it when doing future pattern matches.

3.4 Switch Refinement

If the subgraph at node n fails to match a known schema, it may be a switch candidate. *Switch candidates* are regions that would match a switch schema in Table 3 but contain extra edges. A switch candidate can fail to match the switch schema if it has extra incoming edges or multiple successors. For instance, the nodes in the `IncSwitch[·]` box in Figure 4 would not be identified as an `IncSwitch[·]` region because there is an extra incoming edge to the default case node.

A switch candidate is refined by first virtualizing incoming edges to any node other than the switch head. The next step is to ensure there is a single successor of all nodes in the switch. The immediate post-dominator of the switch head is selected as the successor if it is the successor of any of the case nodes. Otherwise, the node that (1) is a successor of a case node, (2) is not a case node itself, and (3) has the highest number of incoming edges from case nodes is chosen as the successor. After the successor has been identified, any outgoing edge from the switch that does not go to the successor is virtualized.

After refinement, a switch candidate is usually collapsed to a `IncSwitch[·]` region. For instance, a common implementation strategy for switches is to redirect inputs handled by the default case (e.g., $x > 20$) to a default

	Seq[n_1, \dots, n_k]: A block of sequential regions that have a single predecessor and a single successor.
	IfThenElse[c, n, n_t, n_f]: If-then-else region.
	IfThen[c, n, n_t]: If-then region.
	IncSwitch[$n, (c_1, n_1), \dots, (c_k, n_k)$]: Incomplete switch region. The outgoing conditions are pairwise disjoint and satisfy $\bigvee_{i \in [1, k]} c_i \neq \text{true}$.
	Switch[$n, (c_1, n_1), \dots, (c_k, n_k)$]: Complete switch region. The outgoing conditions are pairwise disjoint and satisfy $\bigvee_{i \in [1, k]} c_i = \text{true}$.

Table 3: Acyclic regions.

node, and use a jump table for the remaining cases (e.g., $x \in [0, 20]$). This relationship is depicted in Figure 4, along with the corresponding region types. Because the jump table only handles a few cases, it is recognized as an IncSwitch[·]. However, because the default node handles all other cases, together they constitute a Switch[·].

3.5 Cyclic Regions

If the subgraph at node n is cyclic, the algorithm tries to match a loop at n to one of the cyclic loop patterns. It is possible for a node to be the loop header of multiple loops. For instance, nested do-while loops share a common loop header. Distinct loops at node n can be identified by finding back edges pointing to n (see §2.1). Each back edge (n_b, n) defines a loop body consisting of the nodes that can reach n_b without going through the loop header, n . The loop with the smallest loop body is reduced first. This must happen before the larger loops can match the cyclic region patterns, because there is no schema for nested loops.

As shown in Table 4, there are three types of loops. While[·] loops test the exit condition before executing the loop body, whereas DoWhile[·] loops test the exit condition after executing the loop body. If the exit condition occurs in the middle of the loop body, the region is a nat-

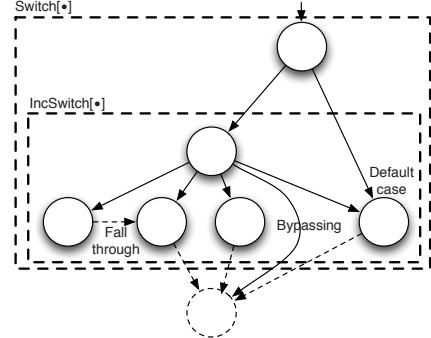


Figure 4: Complete and incomplete switches.

	While[h, b, s, b]: A while loop.
	DoWhile[c, h, b]: A do-while loop.
	NatLoop[$h, b, e_1 \dots e_k$]: A natural loop. Note that there are no edges leaving the loop; outgoing edges must be virtualized during refinement to match this schema.

Table 4: Cyclic regions.

ural loop. Natural loops do not represent one particular C looping construct, but can be caused by code such as

```
while (1) { body1; if (e) break; body2; }
```

Notice that our schema for natural loops contains no outgoing edges from the loop. This is not a mistake, but is required for semantics-preservation. Because NatLoop[·] regions are decompiled to

```
while (1) { ... },
```

which has no exits, the body of the loop must trigger any loop exits. In Phoenix, the loop exits are represented by a tail region, which corresponds to a `goto`, `break`, or `continue` in the decompiled output. These tail regions are added during loop refinement, which we discuss next.

3.6 Loop Refinement

If any loops were detected with loop header n that did not match a loop schema, loop refinement begins. Cyclic regions may fail to match loop schemas because (1) there

```

1 int f(void) {
2     int a = 42;
3     int b = 0;
4     while (a) {
5         if (b) {
6             puts("c");
7             break;
8         } else {
9             puts("d");
10    }
11    a--;
12    b++;
13 }
14 puts("e");
15 return 0;
16 }
```

(a) Original source code

```

1 t_reg32 f (void) {
2     t_reg32 var_20 = 42;
3     t_reg32 var_24;
4     for (var_24 = 0; var_20 != 0; var_24 = var_24 + 1) {
5         if (var_24 != 0) {
6             puts("c");
7             break;
8         }
9         puts("d");
10    var_20 = var_20 - 1;
11 }
12 puts("e");
13 return 0;
14 }
```

(b) Phoenix decompiled output of (a)
with new loop membership definition

```

1 t_reg32 f (void)
2 {
3     t_reg32 var_20 = 42;
4     t_reg32 var_24;
5     for (var_24 = 0; var_20 != 0; var_24 = var_24 + 1) {
6         if (var_24 != 0) goto lab_1;
7         puts("d");
8         var_20 = var_20 - 1;
9     }
10 lab_2:
11     puts("e");
12     return 0;
13 lab_1:
14     puts("c");
15     goto lab_2;
16 }
```

(c) Phoenix decompiled output of (a)
without new loop membership definition

Figure 5: Loop refinement with and without new loop membership definition.

are multiple entrances to the loop, (2) there are too many exits from the loop, or (3) the loop body cannot be collapsed (i.e., is a proper region).

The first step of loop refinement is to ensure the loop has a single entrance (nodes with incoming edges from outside the loop). If there are multiple entrances to the loop, the one with the most incoming edges is selected, and incoming edges to the other entrances are virtualized.

The next step is to identify the type of loop. If there is an exit edge from the loop header, the loop is a While[-] candidate. If there is an outgoing edge from the source of the loop’s back edge (see §2.1), it is a DoWhile[-] candidate. Otherwise, any exit edge is selected and the loop is considered a NatLoop[-] candidate. The exit edge determines the successor of the loop, i.e., the statement that is executed immediately after the loop. The successor in turn determines which nodes are lexically contained in the loop.

Phoenix virtualizes any edge leaving the lexically contained loop nodes other than the exit edge. Edges to the loop header use the continue tail regions, while edges to the loop successor use the break regions. Any other virtualized edge becomes a goto.

In our first implementation, we considered the lexically contained nodes to be the loop body defined by the loop’s back edge [32]. However, we found this definition introduced goto statements when the original program had break statements, as in Figure 5(a). The `puts("c")` statement is *not* in the loop body according to the standard definition, because it cannot reach the loop’s back edge, but it *is* lexically contained in the loop. Obviously, a `break` statement must be lexically contained inside the loop body, or there would be no loop to break out of.

Our observation is that the nodes lexically contained in the loop should intuitively consist of the loop body *and*

any nodes that execute after the loop body but before the successor. More formally, this corresponds to the loop body, and the nodes that are dominated by the loop header, excluding any nodes reachable from the loop’s successor without going through the loop header. For example, `puts("c")` in Figure 5(b) is considered as a node that executes between the loop body and the successor, and thus Phoenix places it lexically inside the loop. When Phoenix uses the standard loop membership definition used in structural analysis, Phoenix outputs gotos, as in Figure 5(c). In our evaluation (§4), we show that enabling the new loop membership definition decreased the numbers of gotos Phoenix emitted by 45% (73 to 40).

The last loop refinement step is to remove edges that may prevent the loop body from being collapsed. This can happen, for instance, when a `goto` was used in the input program. This step is only performed if the prior loop refinement steps did not remove any edges during the latest iteration of the algorithm. For this, we use the last resort refinement on the loop body.

3.7 Last Resort Refinement

If the algorithm does not collapse any nodes or perform any refinement during an iteration, Phoenix removes an edge in the graph to allow it to make progress. We call this process the last resort refinement, because it has the lowest priority, and always allows progress to be made. Last resort refinement prefers to remove edges whose source does not dominate its target, nor whose target dominates its source. These edges can be thought of as cutting across the dominator tree. By removing them, the edges that remain reflect more structure.

4 Evaluation

In this section, we describe the results of our experiments on Phoenix. At a high level, these results demonstrate that Phoenix is suitable for use in program analysis. Specifically, we show that the techniques employed by Phoenix lead to significantly more correct decompilation and more recovered structure than the *de facto* industry standard Hex-Rays. Phoenix was able to decompile 114% more utilities that passed the entire `coreutils` test suite than Hex-Rays (60 vs 28). Our results show that employing semantics-preserving schemas increased correctness by 30% (from 46 to 60). We attribute most remaining correctness errors in Phoenix to type recovery (see §5). Phoenix was able to structure the control flow for 8,676 functions using only 40 `goto`s. This corresponds to recovering 30× more structure (40 `goto`s vs 1,229) than structural analysis without iterative refinement.

4.1 Phoenix Implementation

Our implementation of Phoenix consists of an extension to the BAP framework. We implemented it in OCaml, to ease integration with BAP, which is also implemented in OCaml. Phoenix alone consists of 3,766 new lines of code which were added to BAP. Together, the decompiler and TIE comprise 8,443 lines of code. For reference, BAP consisted of 29,652 lines of code before our additions. We measured the number of lines of code using David A. Wheeler’s SLOCCount utility.

4.2 Metrics

We propose two *quantitative* dimensions for evaluating the suitability of decompilers for program analysis, and then evaluate Phoenix on them:

- **Correctness.** Correctness measures whether the decompiled output is equivalent to the original binary input. If a decompiler produces output that does not actually reflect the behavior of the input binary, it is of little utility in almost all settings. For program analysis, we want decompilers to be correct so that the decompiler does not introduce imprecision. In our experiments we utilize high-coverage tests to measure correctness.
- **Structuredness.** Recovering control flow structure helps program analysis and humans alike. Structured code is easier for programmers to understand [19], and helps scale program analysis in general [32]. Therefore, we propose that decompiler output with fewer unstructured control flow commands such as `goto` are better.

The benefit of our proposed metrics is that they can be evaluated quantitatively and thus can be automatically measured. These properties makes them suitable for an objective comparison of decompilers.

Existing Metrics Note that our metrics are vastly different than those appearing in previous decompiler work. Ci-fuentes proposed using the ratio of the size of the decompiler output to the initial assembly as a “compression ratio” metric, i.e., $1 - (\text{LOC decompiled}/\text{LOC assembly})$ [13]. The idea was the more compact the decompiled output is than the assembly code, the easier it would be for a human to understand the decompiled output. However, this metric side-steps whether the decompilation is correct or even compilable. A significant amount of previous work has proposed no metrics. Instead, they observed that the decompiler produced output, or had a manual qualitative evaluation on a few, small examples [11, 13, 21, 22, 39]. Previous work that does measure correctness [20, 28] only focuses on a small part of the decompilation process, e.g., type recovery or control flow structuring. However, it does not measure end-to-end correctness of the decompiler as a whole.

4.3 Coreutils Experiment Overview

We tested Phoenix on the GNU `coreutils` 8.17 suite of utilities. `coreutils` consists of 107² mature, standard programs used on almost every Linux system. `coreutils` also has a suite of high-coverage tests that can be used to measure correctness. Though prior work has studied individual decompiler components on a large scale (see §6), to the best of our knowledge, our evaluation on `coreutils` is an order of magnitude larger than any other systematic end-to-end decompiler evaluation in which specific metrics were defined and measured.

Tested Decompilers In addition to Phoenix, we tested the latest publicly available version of the academic decompiler Boomerang [39] and Hex-Rays [23], the *de facto* industry standard decompiler. We tested the latest Hex-Rays version, which is 1.7.0.120612 as of this writing.

We also considered other decompilers such as REC [35], DISC [26], and `dcc` [13]. However, these compilers either produced pseudo-code (e.g., REC), did not work on x86 (e.g., `dcc`), or did not have any documentation that suggested advancements beyond Boomerang (e.g., DISC).

²The number of utilities built depends on the machine that `coreutils` is compiled on. This is the number applicable to our testing system, which ran Ubuntu 12.04.1 x86-64. We compiled `coreutils` in 32-bit mode because the current Phoenix implementation only supports 32-bit binaries.

We encountered serious problems with both Boomerang and Hex-Rays in their default configurations. First, Boomerang failed to produce any output for all but a few `coreutils` programs. Boomerang would get stuck while decompiling one function, and would never move on to other functions. We looked at the code, but there appeared to be no easy or reasonable fix to enable some type of per-function timeout mechanism. Boomerang is also no longer actively maintained. Second, Hex-Rays did not output compliant C code. In particular, Hex-Rays uses non-standard C types and idioms that only Visual Studio recognizes, and causes almost every function to fail to compile with `gcc`. Specifically, the Hex-Rays website states: “[...] the produced code is not supposed to be compilable and many compilers will complain about it. This is a deliberate choice of not making the output 100% compilable because the goal is not to recompile the code but to analyze it.” Even if Hex-Rays output is intended to be analyzed rather than compiled, it should still be correct modulo compilation issues. After all, there is little point to pseudo-code if it is semantically incorrect.

Because Hex-Rays was the only decompiler we tested that actually produced output for real programs, we investigated the issue in more detail and noticed that the Hex-Rays output was only uncompliable because of the Visual Studio idioms and types it used. In order to offer a conservative comparison of Phoenix to existing work, we wrote a post-processor for Hex-Rays that translates the Hex-Rays output to compliant C. The translation is extremely straightforward. For example, one of the translations is that types such as `unsigned __intN` must be converted to `uintN_t`³. All experiments are reported with respect to the post-processed Hex-Rays output. We stress this is intended to make the comparison more fair: without the post-processing Hex-Rays output fails to compile using `gcc`.

4.4 Coreutils Experiment Details

4.4.1 Setup

Testing decompilers on real programs is difficult because they are not capable of decompiling all functions. This means that we cannot decompile every function in a binary, recompile the resulting source, and expect to have a working binary. However, we would like to be able to test the functions that *can* be decompiled. To this end, we propose the substitution method.

The goal of the substitution method is to produce a *recompiled* binary that consists of a combination of orig-

³Although it seems like this should be possible to implement using only a C header file containing some `typedefs`, a `typedef` has its qualifiers fixed. For instance, `typedef int t` is equivalent to `typedef signed int t`, and thus the type `unsigned t` is not allowed because `unsigned signed int` is contradictory.

inal source code and decompiled source code. We implemented the substitution method by using CIL [33] to produce a C file for each function in the original source code. We compiled each C file to a separate object file. We also produced object files for each function emitted by the decompiler in a similar manner. We then created an initial recompiled binary by linking all of the original object files (i.e., object files compiled from the original source code) together to produce a binary. We then iteratively substituted a decompiler object file (i.e., object files compiled from the decompiler’s output) for its corresponding original object file. If linking this new set of object files succeeded without an error, we continued using the decompiler object file in future iterations. Otherwise we reverted to using the original object file. For our experiments, we produced a recompiled binary for each decompiler and utility combination.

Of course, for fairness, we must ensure that the recompiled binaries for each decompiler have approximately the same number of decompiled functions, since non-decompiled functions use the original function definition from the `coreutils` source code, which presumably passes the test suite and is well-structured. The number of recompilable functions output by each decompiler is broken down by utility in Figure 6. Phoenix recompiled 10,756 functions in total, compared to 10,086 functions for Hex-Rays. The Phoenix recompiled binaries consist of 82.2% decompiled functions on average, whereas the Hex-Rays binaries contain 77.5%. This puts Phoenix at a slight disadvantage for the correctness tests, since it uses fewer original functions. Hex-Rays did not produce output after running for several hours on the `sha384sum` and `sha512sum` utilities. Phoenix did not completely fail on any utilities, and was able to decompile 91 out of 110 functions (82.7%) for both `sha384sum` and `sha512sum`. (These two utilities are similar). We discuss Phoenix’s limitations and failure modes in §5.

4.4.2 Correctness

We test the correctness of each recompiled utility by running the `coreutils` test suite with that utility and *original* versions of the other utilities. We do this because the `coreutils` test suite is self-hosting, that is, it uses its own utilities to set up the tests. For instance, a test for `mv` might use `mkdir` to setup the test; if the recompiled version of `mkdir` fails, we could accidentally blame `mv` for the failure, or worse, incorrectly report that `mv` passed the test when in reality the test was not properly set up.

Each tested utility U can either pass all tests, or fail. We do not count the number of failed tests, because many utilities have only one test that exercises them. We have observed decompiled utilities that crash on every execution and yet fail only a single test. Thus, it would be

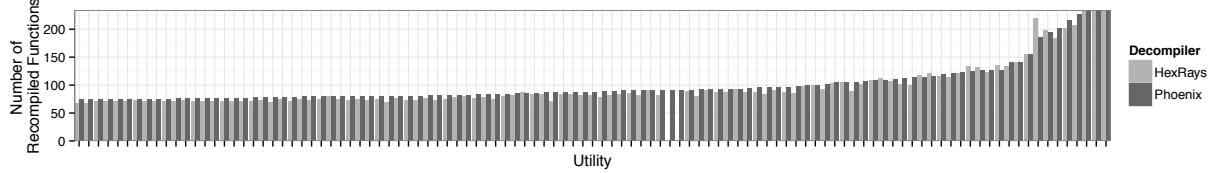


Figure 6: The number of functions that were decompiled and recompiled by each decompiler, broken down by utility. Hex-Rays failed on two utilities for unknown reasons.

	Phoenix	HR
Correct utilities recompiled	60	28
Correct utilities recompiled (semantics-preservation disabled)	46	n/a
Percentage recompiled functions (correct utilities only)	85.4%	73.8%

Table 5: Correctness measurements for the `coreutils` experiment. These results includes two utilities for which Hex-Rays recompiled zero functions (thus trivially passing correctness).

misleading to conclude that a recompiled program performed well by “only” failing one test.

The results of the correctness tests are in Table 5. To summarize, Hex-Rays recompiled 28 utilities that passed the `coreutils` test suite. Phoenix was able to recompile 60 passing utilities (114% more). However, we want to ensure that these utilities are not simply correct because they consist mostly of the original `coreutils` functions. This is not the case for Phoenix: the recompiled utilities that passed all tests consisted of 85.4% decompiled functions on average, which is actually higher than the overall Phoenix average of 82.2%. The correct Hex-Rays utilities consisted of 73.8% decompiled functions, which is less than the overall Hex-Rays average of 77.5%. As can be seen in Figure 6, this is because Hex-Rays completely failed on two utilities. The recompiled binaries for these utilities consisted completely of the original source code, which (unsurprisingly) passed all tests. Excluding those two utilities, Hex-Rays only compiled 26 utilities that passed the tests. These utilities consisted of 79.4% decompiled functions on average.

We also re-ran Phoenix with the standard structural analysis schemas, including those that are *not* semantics-preserving, in order to evaluate whether semantics-preservation has an observable effect on correctness. With these schemas, Phoenix produced only 46 correct utilities. This 30% reduction in correctness (from 60 down to 46) illustrates the importance of using semantics-preserving schemas.

	Phoenix	HR
Total gotos	40	51
Total gotos (without loop membership)	73	n/a
Total gotos (without refinement)	1,229	n/a

Table 6: Structuredness measurements for the `coreutils` experiment. The statistics only reflect the 8,676 recompilable functions output by both decompilers.

4.4.3 Structuredness

Finally, we measure the amount of structure recovered by each decompiler. Our approach here is straightforward: we count the number of goto statements emitted by each decompiler. To ensure a fair comparison, we only consider the intersection of recompilable functions emitted by both decompilers, which consists of 8,676 functions. Doing otherwise would penalize a decompiler for outputting a function with goto statements, even if the other decompiler could not decompile that function at all.

The overall structuredness results are depicted in Table 6, with the results broken down per utility in Figure 7. In summary, Phoenix recovered the structure of the 8,676 considered functions using only 40 gotos. Furthermore, Phoenix recovered significantly less structure when either refinement (1189 more gotos) or the new loop membership definition (33 more) was disabled. Our results suggest that structuring algorithms without iterative refinement [20, 32, 36] will recover less structure. The results also suggest that Hex-Rays employs a technique similar to iterative refinement.

5 Limitations and Future Work

5.1 BAP Failures

Phoenix uses BAP [10] to lift instructions to a simple language that is then analyzed. BAP does not have support for floating point and other exotic types of instructions. Phoenix will not attempt to decompile any function that contains instructions which are not handled by BAP. BAP can also fail for other reasons. It uses value set analy-

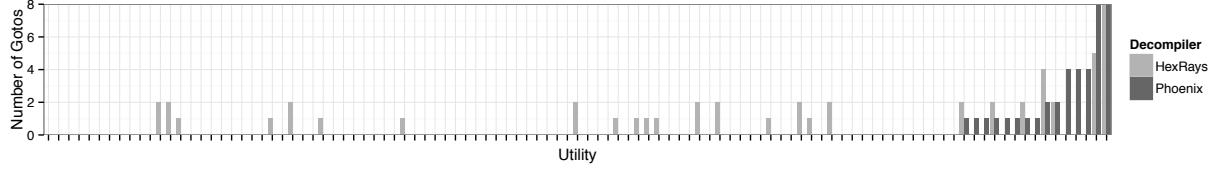


Figure 7: The number of gotos emitted by each decompiler, broken down by utility. Only functions that were decompiled and recompiled by both decompilers are counted.

sis (VSA) to perform CFG recovery, and to help with other parts of the decompilation process. If VSA or other mandatory analyses fail, then the decompiler cannot proceed. These problems can cascade to affect other functions. For instance, if CFG recovery for function g fails and function f calls g , function f will also fail, because it is necessary to know the type of g before calling it.

5.2 Correctness Failures

Because Phoenix is designed for program analysis, we want it to be correct. Our experiments show that, although Phoenix significantly improves over prior work with respect to correctness, Phoenix’s output is not always correct. The good news is that we can attribute most correctness errors in Phoenix to the underlying type recovery component we used, TIE [28]. Many of the problems, which we describe below, only became apparent when TIE was stress-tested by Phoenix.

Iterative Variable Recovery TIE does not always identify all local variables. For instance, if function f takes a pointer to an integer, and a function calls $f(x)$, then TIE infers that x is a subtype of a pointer to an integer. However, TIE does *not* automatically infer that $*x$, the locations that x can point to, are potentially integer variables. TIE does not recover such variables because it would need to iteratively discover variables, generate and solve type constraints to do so. Unfortunately, undiscovered variables can cause incorrect decompilation for Phoenix. For example, if the undiscovered variable is a struct on the stack, space for the struct is never allocated, which allows the called function to read and overwrite other variables on the stack of the callee. This is the leading cause of correctness errors in Phoenix. In the future, we plan to investigate running type recovery until the set of known variables reaches a fix point.

Calling Conventions TIE currently assumes that all functions use the `cdecl` calling convention, and does not support multi-register (64-bit) return values. Unfortunately, this can make Phoenix output incorrect or uncompileable code. In the future, we plan to use an interpro-

cedural liveness analysis to automatically detect calling conventions based on the behavior of a function and the functions that call it. Our goal is to detect and understand calling conventions automatically, even when they are non-standard. This is important for analyzing malware, some of which uses unusual calling conventions to try to confuse static analysis.

Recursive Types TIE has no support for recursive types, although these are used quite frequently for data structures like linked lists and binary trees. This means that the type

```
struct s { int a; struct s *next; }
```

will be inferred as

```
struct s { int a; void* next; }
```

which does not specify what type of element `next` points to. Since Phoenix is intended to be the front-end of an analysis platform, we would like to recover the most specific type possible. We plan to investigate more advanced type inference algorithms that can handle recursive types.

6 Related Work

At a high level, there are three lines of work relevant to Phoenix. First, work in end-to-end decompilers. Second, work in control structure recovery, such as loop identification and structural analysis. Third, literature pertaining to type recovery.

Decompilers The earliest work in decompilation dates back to the 1960’s. For an excellent and thorough review of the literature in decompilation and several related areas up to around 2007, see Van Emmerik’s thesis [39, ch. 2]. Another in-depth overview is available online [18].

Modern decompilers typically trace their roots in Cifuentes’ 1994 thesis on `dcc` [13], a decompiler for 80286 to C. The structuring algorithm used in `dcc` is based on interval analysis [2]. Cifuentes proposed the compression ratio metric (see §4.2), but did not measure correctness on the ten programs that `dcc` was tested on [14]. Since

compression is the target metric, *dcc* outputs assembly if it encounters code that it cannot handle. Cifuentes et al. have also created a SPARC asm to C decompiler, and measured compressibility and the number of recovered control structures on seven SPEC1995 programs [16]. Again, they did not test the correctness of the decompilation output. Cifuentes [13] pioneered the technique of recovering short-circuit evaluations in compound expressions (e.g., $x \&& (!y || z)$ in C).

Chang et al. [11] also use compressibility in their work on cooperating decompilers for the three programs they tested. Their main purpose was to show they can find bugs in the decompiled source that were known to exist in the binary. However, correctness of the decompilation itself was not verified.

Boomerang is a popular open-source decompiler started by Van Emmerik as part of his Ph.D. [39]. The main idea of Van Emmerik’s thesis is that decompilation analysis is easier on the Single Static Assignment (SSA) form of a program. In his thesis, Van Emmerik’s experiments are limited to a case study of Boomerang coupled with manual analysis to reverse engineer a single 670KB Windows program. We tested Boomerang as part of our evaluation, but it failed to produce any output on all but a few of our test cases after running for several hours.

The structuring algorithm used in Boomerang first appeared in Simon [37], who in collaboration with Cifuentes proposed a new algorithm known as “parenthesis theory”. Simon’s own experiments showed that parenthesis theory is faster and more space efficient than the interval analysis-based algorithm in *dcc*, but recovers less structure.

Hex-Rays is the *de facto* industry decompiler. The only information we have found on Hex-Rays is a 2008 write-up [23] by Hex-Rays’ author, Guilfanov, who revealed that Hex-Rays also performs structural analysis. However, Hex-Rays achieves much better structuredness than vanilla structural analysis, which suggests that Hex-Rays is using a heavily modified version. There are many other binary-to-C decompilers such as REC [35] and DISC [26]. However, our experience suggests that they are not as advanced as Hex-Rays.

Our focus is on decompiling C binaries. Other researchers have investigated decompiling binaries from managed languages such as Java [30]. The set of challenges they face are fundamentally different. On the one hand, these managed binaries contain extra information such as types; on the other hand, recovering the control flow itself in the presence of exceptions and synchronization primitives is a difficult problem.

Control Structure Recovery Control structure recovery is also studied in *compilation*. This is because by the time compilation is in the optimization stage, the input

program has already been parsed into a low-level intermediate representation (IR) in which the high-level control structure has been destroyed. Much work in program optimization therefore attempts to recover the control structures.

The most relevant line of work in this direction is the elimination methods in data flow analysis (DFA), pioneered by Allen [2] and Cooke [17] in the 1970’s and commonly known as “interval analysis”. Sharir [36] subsequently refined interval analysis into structural analysis. In Sharir’s own words, structural analysis can be seen as an “unparser” of the CFG. Besides the potential to speed up DFA even more when compared to interval analysis, structural analysis can also cope with irreducible CFGs.

Engel et al. [20] are the first to extend structural analysis to handle C-specific control statements. Specifically, their Single Entry Single Successor (SESS) analysis adds a new tail region type, which corresponds to the statements that appear before a *break* or *continue*. For example, suppose `if (b) { body; break; }` appears in a loop, then the statements represented by *body* would belong to a tail region. Engel et al. have extensively tested their implementation of SESS in a source-to-source compiler. However, their SESS analysis does not use iterative refinement, and can get stuck on unstructured code. We show in our evaluation that this leads to a large amount of structure being missed. Their exact algorithm for detecting tail regions is also left unspecified [20, Algorithm 2, Line 15].

Another line of related work lies in the area of program schematology, of which “Go To Statement Considered Harmful” by Dijkstra [19] is the most famous. Besides the theoretical study of the expressive power of *goto* vs. high-level control statements (see, e.g., [34]), this area is also concerned with the automatic structuring of (unstructured) programs, such as the algorithm by Baker [3].

Type Recovery Besides control structure recovery, a high-quality decompiler should also recover the types of variables. Much work has gone into this recently. Phoenix uses TIE [28], which recovers types statically. In contrast, REWARDS [29] and Howard [38] recover types from dynamic traces. Other work has focused on C++-specific issues, such as virtual table recovery [21, 22].

7 Conclusion

We presented Phoenix, a new binary-to-C decompiler designed to accurately and effectively recover abstractions. Phoenix can help leverage the wealth of existing source-based tools and techniques in security scenarios, where source code is often unavailable. Phoenix uses a novel control flow structuring algorithm that avoids a previously

unpublished correctness pitfall in decompilers, and uses iteratively refinement to recover more control flow structure than existing algorithms. We evaluated Phoenix and the *de facto* industry standard decompiler, Hex-Rays, on correctness and amount of control flow structure recovered. Phoenix decompiled twice as many utilities correctly as Hex-Rays, and recovered more structure.

Acknowledgments

This material is based upon work supported by DARPA under Contract No. HR00111220009. 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 DARPA.

References

- [1] Alfred Aho, Monica Lam, Ravi Sethi, and Jeffrey Ullman. *Compilers: Principles, Techniques, and Tools*. Addison Wesley, 2nd edition, 2006.
- [2] Frances E. Allen. Control Flow Analysis. In *Proceedings of ACM Symposium on Compiler Optimization*, pages 1–19, 1970.
- [3] Brenda S. Baker. An Algorithm for Structuring Flowgraphs. *Journal of the ACM*, 24(1):98–120, 1977.
- [4] Gogul Balakrishnan. *WYSINWYX: What You See Is Not What You eXecute*. PhD thesis, Computer Science Department, University of Wisconsin-Madison, August 2007.
- [5] Sébastien Bardin, Philippe Herrmann, and Franck Vedrine. Refinement-Based CFG Reconstruction from Unstructured Programs. In *Proceedings of the 12th International Conference on Verification, Model Checking, and Abstract Interpretation*, pages 54–69. Springer, 2011.
- [6] Al Bessey, Ken Block, Ben Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles Henri-Gros, Asya Kamsky, Scott McPeak, and Dawson Engler. A Few Billion Lines of Code Later. *Communications of the ACM*, 53(2):66–75, 2010.
- [7] The BitBlaze Binary Analysis Platform. <http://bitblaze.cs.berkeley.edu>, 2007.
- [8] Erik Bosman, Asia Slowinska, and Herbert Bos. Minemu: The World’s Fastest Taint Tracker. In *Proceedings of the 14th International Symposium on Recent Advances in Intrusion Detection*, pages 1–20. Springer, 2011.
- [9] David Brumley, Tzi-cker Chiueh, Robert Johnson, Huijia Lin, and Dawn Song. RICH: Automatically Protecting Against Integer-Based Vulnerabilities. In *Proceedings of the Network and Distributed System Security Symposium*. The Internet Society, 2007.
- [10] David Brumley, Ivan Jager, Thanassis Avgerinos, and Edward J. Schwartz. BAP: A Binary Analysis Platform. In *Proceedings of the 23rd International Conference on Computer Aided Verification*, pages 463–469. Springer, 2011.
- [11] Bor-yuh Evan Chang, Matthew Harren, and George C. Necula. Analysis of Low-Level Code Using Cooperating Decompilers. In *Proceedings of the 13th International Symposium on Static Analysis*, pages 318–335, 2006.
- [12] Walter Chang, Brandon Streiff, and Calvin Lin. Efficient and Extensible Security Enforcement Using Dynamic Data Flow Analysis. In *Proceedings of the 15th ACM Conference on Computer and Communications Security*, pages 39–50, 2008.
- [13] Cristina Cifuentes. *Reverse Compilation Techniques*. PhD thesis, Queensland University of Technology, 1994.
- [14] Cristina Cifuentes. Interprocedural Data Flow De-compilation. *Journal of Programming Languages*, 4(2):77–99, 1996.
- [15] Cristina Cifuentes and K. John Gough. Decomposition of Binary Programs. *Software: Practice and Experience*, 25(7):811–829, 1995.
- [16] Cristina Cifuentes, Doug Simon, and Antoine Fraboulet. Assembly to High-Level Language Translation. In *Proceedings of the International Conference on Software Maintenance*, pages 228–237. IEEE, 1998.
- [17] John Cocke. Global Common Subexpression Elimination. In *Proceedings of the ACM Symposium on Compiler Optimization*, pages 20–24, 1970.
- [18] The Decompilation Wiki. <http://www.program-transformation.org/Transform/DeCompilation>. Page checked 6/25/2013.
- [19] Edsger W. Dijkstra. Letters to the Editor: Go To Statement Considered Harmful. *Communications of the ACM*, 11(3):147–148, 1968.
- [20] Felix Engel, Rainer Leupers, Gerd Ascheid, Max Ferger, and Marcel Beemster. Enhanced Structural Analysis for C Code Reconstruction from IR Code. In *Proceedings of the 14th International Workshop*

- on Software and Compilers for Embedded Systems*, pages 21–27. ACM, 2011.
- [21] Alexander Fokin, Egor Derevenetc, Alexander Chernov, and Katerina Troshina. SmartDec: Approaching C++ Decompilation. In *Proceedings of the 18th Working Conference on Reverse Engineering*, pages 347–356. IEEE, 2011.
- [22] Alexander Fokin, Katerina Troshina, and Alexander Chernov. Reconstruction of Class Hierarchies for Decomposition of C++ Programs. In *Proceedings of the 14th European Conference on Software Maintenance and Reengineering*, pages 240–243. IEEE, 2010.
- [23] Ilfak Guifanov. Decompilers and Beyond. In *Black-Hat USA*, 2008.
- [24] Johannes Kinder and Helmut Veith. Jakstab: A Static Analysis Platform for Binaries. In *Proceedings of the 20th International Conference on Computer Aided Verification*, pages 423–427. Springer, 2008.
- [25] Christopher Kruegel, William Robertson, Fredrik Valeur, and Giovanni Vigna. Static Disassembly of Obfuscated Binaries. In *Proceedings of the 13th USENIX Security Symposium*, pages 255–270, 2004.
- [26] Satish Kumar. DISC: Decompiler for TurboC. <http://www.debugmode.com/dcompile/disc.htm>. Page checked 6/25/2013.
- [27] David Larochelle and David Evans. Statically Detecting Likely Buffer Overflow Vulnerabilities. In *Proceedings of the 10th USENIX Security Symposium*, pages 177–190, 2001.
- [28] JongHyup Lee, Thanassis Avgerinos, and David Brumley. TIE: Principled Reverse Engineering of Types in Binary Programs. In *Proceedings of the Network and Distributed System Security Symposium*. The Internet Society, 2011.
- [29] Zhiqiang Lin, Xiangyu Zhang, and Dongyan Xu. Automatic Reverse Engineering of Data Structures from Binary Execution. In *Proceedings of the Network and Distributed System Security Symposium*. The Internet Society, 2010.
- [30] Jerome Miecznikowski and Laurie Hendren. Decompiling Java Bytecode: Problems, Traps and Pitfalls. In *Proceedings of the 11th International Conference on Compiler Construction*, pages 111–127. Springer, 2002.
- [31] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised)*. The MIT Press, 1997.
- [32] Steven Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.
- [33] George C. Necula, Scott McPeak, Shree P. Rahul, and Westley Weimer. CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs. In *Proceedings of the 11th International Conference on Compiler Construction*, pages 213–228. Springer, 2002.
- [34] W. W. Peterson, T. Kasami, and N. Tokura. On the Capabilities of While, Repeat, and Exit Statements. *Communications of the ACM*, 16(8):503–512, 1973.
- [35] REC Studio 4—Reverse Engineering Compiler. <http://www.backerstreet.com/rec/rec.htm>. Page checked 6/25/2013.
- [36] Micha Sharir. Structural Analysis: A New Approach to Flow Analysis in Optimizing Compilers. *Computer Languages*, 5(3-4):141–153, 1980.
- [37] Doug Simon. *Structuring Assembly Programs*. Honours thesis, University of Queensland, 1997.
- [38] Asia Slowinska, Traian Stancescu, and Herbert Bos. Howard: A Dynamic Excavator for Reverse Engineering Data Structures. In *Proceedings of the Network and Distributed System Security Symposium*. The Internet Society, 2011.
- [39] Michael James Van Emmerik. *Static Single Assignment for Decompilation*. PhD thesis, University of Queensland, 2007.
- [40] Xi Wang, Haogang Chen, Zhihao Jia, Nickolai Zeldovich, and M. Frans Kaashoek. Improving Integer Security for Systems with KINT. In *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation*, pages 163–177, 2012.

Strato: A Retargetable Framework for Low-Level Inlined-Reference Monitors

Bin Zeng

*Department of Computer
Science and Engineering
Lehigh University*

Gang Tan

*Department of Computer
Science and Engineering
Lehigh University*

Úlfar Erlingsson

Google Inc.

Abstract

Low-level Inlined Reference Monitors (IRM) such as control-flow integrity and software-based fault isolation can foil numerous software attacks. Conventionally, those IRMs are implemented through binary rewriting or transformation on equivalent low-level programs that are tightly coupled with a specific Instruction Set Architecture (ISA). Resulting implementations have poor retargetability to different ISAs. This paper introduces an IRM-implementation framework at a compiler intermediate-representation (IR) level. The IR-level framework enables easy retargetability to different ISAs, but raises the challenge of how to preserve security at the low level, as the compiler backend might invalidate the assumptions at the IR level. We propose a constraint language to encode the assumptions and check whether they still hold after the backend transformations and optimizations. Furthermore, an independent verifier is implemented to validate the security of low-level code. We have implemented the framework inside LLVM to enforce the policy of control-flow integrity and data sandboxing for both reads and writes. Experimental results demonstrate that it incurs modest runtime overhead of 19.90% and 25.34% on SPECint2000 programs for x86-32 and x86-64, respectively.

1 Introduction

Software attacks are common, from code-injection attacks to more sophisticated techniques such as Return Oriented Programming (ROP [6, 29]). ROP chains the attacked program’s code snippets, referred to as *gadgets*, to achieve functionality desired by the attacker. It can bypass many defensive techniques such as StackGuard and Data Execution Prevention (DEP) [22].

Low-level inlined reference monitors (IRM [15–17]) are effective at preventing attacks against software systems. In this approach, checks are inlined into binary

code to ensure critical security properties. Take the example of software-based fault isolation (SFI). It is a code-sandboxing technique that isolates untrusted modules from trusted environments [32]. By having separate code and data regions and by making the data region un-executable, SFI prevents code-injection attacks in addition to containing faults in sandboxed modules.

Another effective IRM is control-flow integrity (CFI [1]). An essential step in many software attacks is to induce an illegal control flow transfer to maliciously injected code, or to some library function as in jump-to-libc attacks, or to some existing code snippet as in ROP attacks. CFI enforces a strong runtime guarantee that execution paths follow a predetermined control flow graph, which is constructed either by source-code analysis, binary analysis, or program profiling. CFI can greatly decrease where ROP gadgets can be discovered and further restrain the way gadgets can be chained, thus effectively mitigating ROP attacks.

Low-level IRMs such as SFI and CFI are usually implemented through low-level rewriting, either by performing binary instrumentation, assembly-code instrumentation, or by modifying a compilation tool chain’s backend to emit code with embedded checks. As an example, PittSFIeld was implemented by assembly-code instrumentation [21]. Google’s Native Client (NaCl [28, 37]) was built by modifying the backend of the GNU tool chain. One key benefit of rewriting at the low level is that a separate verifier can be built to check the result of rewriting. The separate verifier removes the rewriter outside of the TCB. Furthermore, in a distributed environment, only the verifier needs to be installed at the client’s side. The verifier checks the security of untrusted, remotely downloaded modules. The security architecture of NaCl follows the separation between the rewriter and the verifier.

On the other hand, low-level rewriting is tightly coupled with a certain ISA, resulting in poor reusability and retargetability, and hindering optimizations. It is non-

trivial to port a low-level IRM to another ISA and existing parts are hard to reuse. For instance, NaCl’s initial implementation was on x86-32 and its port to x86-64 and ARM involved significant effort in design and implementation [28]. One reason for the nontrivial effort is the differences among ISAs, including the instruction set, the available hardware features, the number and size of registers, and others. In addition, many components need to be built from scratch. A typical example is optimizations. Any decent IRM implementation requires optimizations to bring down the runtime cost. However, those optimizations are tied to an ISA and hard to reuse.

We explore the building of a retargetable framework for low-level IRMs on a high-level compiler intermediate representation; in particular, the LLVM IR [20]. The framework, called Strato¹, is general in the sense that various security policies can be conveniently enforced and much code can be reused among them and that inlined high-level checks for a specific policy can be lowered into distinct machine-code sequences. In Strato, we have enforced CFI and data sandboxing for both memory writes and reads.

IR-level rewriting comes with several benefits. First, it is retargetable. Security checks are inserted into the high-level representation. The check-insertion component is shared by all target ISAs the compiler supports. Optimizations that operate on the IR are also reused among different targets. To support a new target ISA, only the lowering from high-level checks to machine-instruction sequences needs to be changed. Even for the same ISA, it is easy to explore different machine-instruction sequences that implement the same high-level check since the lowering part can be easily changed. Our framework was originally built to support x86-32 and then extended to support x86-64; only a small amount of code was altered to retarget it for x86-64.

The second benefit of IR-level rewriting is that optimizations are easier to implement and more optimizations can be supported. An IR usually carries a wealth of structured information and attains many properties that are amenable to program analyses and optimizations. For instance, LLVM IR is in the Static Single Assignment (SSA) form [8, 9], making analysis easier to implement. In addition, LLVM IR preserves type information, loop information, and dominator-tree information, which facilitate analyses and optimizations. Finally, a high-level representation contains many fewer instructions than a typical target machine (e.g., in LLVM 2.9, LLVM IR has only 54 instructions while the x86-64 target has over 3,500). All these benefits make it easier to implement optimizers that remove or hoist security checks; we call these optimizers *security-check optimizers*.

However, the downside of pure IR-level rewriting is that it results in a larger TCB compared to low-level

rewriting. The compiler backend performs sophisticated transformations to generate low-level code, including instruction selection, ISA-specific optimizations, and register allocation. Those transformations can invalidate the hypotheses assumed by a security mechanism at the high level. First, a bug in a transformation can produce insecure low-level code. A more subtle issue is that those transformations may assume a machine model different from the attack model of a low-level IRM. As a simple example, a backend transformation might assume that a variable holds the last stored value after it is loaded back from the memory location into which the variable is spilled. However, many low-level IRMs such as CFI assume the memory may change arbitrarily between any two instructions because of memory-corruption attacks. Under this attack model, a spilled variable cannot be assumed to hold the same value. Consequently, the transformation may produce insecure code according to the attack model.

Therefore, the challenge is how to perform IR-level rewriting while still preserve low-level security. Strato adopts a twofold approach. First, it includes a novel constraint encoding and checking process to propagate assumptions required by security-check optimizers. The optimizers do not remove checks; instead, they mark them as removable and attach constraints to them. After the backend transformations, Strato checks whether the constraints have been invalidated by the backend. If they are not, the unnecessary security checks are removed or hoisted. Otherwise, the checks are left intact to preserve the security of the low-level code. In other words, the security-check optimizers mark optimizations at the IR level, but the effect is taken only at the low-level code after ensuring constraints are not violated. To further ensure the trustworthiness, we implement an independent verifier to validate the final low-level code, thus removing all the instrumentations, transformations, optimizations, and constraint checking out of the TCB. The verifier helped us uncover 35 critical bugs in early versions of Strato.

The key contributions of our work are as follows.

- A reusable and retargetable framework is proposed, built and evaluated to enforce low-level IRMs on a high-level IR. To the best of our knowledge, this is the first framework that brings the benefits of high-level representations to low-level IRMs and loses no trustworthiness. On top of that, we have implemented two low-level IRMs including CFI and data sandboxing for both x86-32 and x86-64. To demonstrate the benefits of the IR-level approach, we have implemented three conventional optimizations with ease and the runtime overhead is lower than previous work.

- Two techniques are proposed to ensure trustworthiness, including the constraint encoding/checking and the low-level verifier. A constraint language is used to encode assumptions that are carried across the code-generation barrier.
- We explore and evaluate a number of alternative security-check instruction sequences for both CFI and data sandboxing. Different instruction sequences have varying overhead on different ISAs and programs. We have discovered more efficient instruction sequences than previous work.

This paper is organized as follows. Section 2 describes related work. Section 3 introduces the overview of Strato. Section 4 presents how Strato performs check instrumentation and optimization; it also presents the constraint language. Section 5 discusses the phase of constraint checking and check lowering. Section 6 elaborates on the low-level verification process. Section 7 discusses the implementation and evaluation. The last section concludes and proposes future work.

2 Related Work

Strato is inspired by many previous low-level security techniques. Its special focus is to build a retargetable infrastructure to assist the exploration and optimizations of security techniques at a high-level representation.

2.1 Inlined Reference Monitors (IRMs)

IRMs embed checks into subject programs to enforce security policies. This approach can be carried at different language levels, from source code, to an intermediate representation, or to low-level code. A typical example of source-code IRM is CCured [25], which inserts checks into C code for memory safety. At the IR level, a number of systems insert checks for various kinds of policies [11, 15, 17, 24]. At the low level, checks can be inserted to enforce policies such as control-flow integrity. Clearly, this is a well-studied research area. Our system sets itself apart by performing IR-level rewriting and preserving low-level security. The IR-level rewriting is adopted for retargetability and for the ease of implementing optimizations. At the same time, we propose techniques to ensure that IR-level rewriting is valid with respect to security policies at the low-level.

Next we discuss closely related systems in the area of IRMs and compare them with Strato.

Software-based Fault Isolation. SFI isolates untrusted or faulty modules from a trusted environment [21, 28, 31, 32, 34, 37]. In SFI, checks are inserted before memory-access and control-flow instructions to ensure

memory access and control flow stay in a sandbox. A carefully designed interface is the only pathway through which sandboxed modules interact with the rest of the system. One subtle requirement of SFI and other IRMs is that the inserted security checks cannot be bypassed by computed jumps, making some form of control flow restriction necessary. Recent SFI implementations use instruction alignment for a crude form of control-flow integrity.

For efficiency, SFI is typically implemented through a combination of static verification and inlined checks. The safety of direct memory accesses and direct jumps can be statically checked. For computed memory visits and indirect jumps, checks are inlined to make sure that those operations stay in the sandbox. Traditional SFI implementations are performed through low-level rewriting. As a result, they are tightly tied to a specific target machine and difficult to port to other ISAs. One advantage of low-level rewriting is that it holds the promise of rewriting without source code being available. However, most previous SFI implementations still ask for the cooperation of the code producer by requiring assembly code or a special compiler to be used. A recent SFI system [34] makes substantial progress toward an implementation through pure binary rewriting; it remains to be seen whether the system can be generalized to IRMs other than SFI and how optimizations can be accommodated.

Control Flow Integrity. CFI ensures that runtime control flow follows a predetermined control flow graph even if the whole data memory is under the control of attackers [2, 3]. One way to enforce CFI is to insert IDs at the targets of a control transfer and a check before the control transfer [3]; the check ensures that the expected ID is at the actual control transfer destination. The system by Wang et al. enforces CFI through defunctionalization [33]. For computed control flow transfers, their system encodes all potential targets in a write-protected table and uses an index to retrieve the target. Before each computed control transfer, the index is checked to make sure that it falls into the table before it is used to fetch the target address from the table. Our system implements CFI in a way similar to the original implementation [3], but at the IR level. In addition, we explore and evaluate a number of instruction sequences for CFI enforcement and find efficient instruction sequences that reduce the runtime cost of CFI.

Combining CFI with other IRMs. On top of CFI, XFI employs a protected shadow stack to store return addresses [14]. XFI promotes control flow precision from Deterministic Finite Automata (DFA) to Pushdown Automata (PDA). However, XFI is platform specific and its runtime overhead is significant.

Our previous system [38] also implements both CFI

and data sandboxing and proposes optimizations to decrease the runtime cost. However, that implementation performs x86-32 assembly rewriting and cannot be re-targeted to other ISAs. By contrast, Strato can target any ISA that a compiler supports and the instrumentation and optimizations are shared among different targets. In addition, optimizations in the previous system use the same range-analysis technique adopted in its verifier, making its trustworthiness questionable. Finally, its verifier is path insensitive and is not as accurate as the one in the new system.

LLVM IR rewriting. A number of systems perform rewriting on the LLVM IR for security. SAFE-Code [11, 12] is an enhanced version of LLVM that can enforce object-level integrity (which is close to type safety). SoftBound [24] also takes the approach of IR-level rewriting. It instruments the LLVM IR for enforcing spatial memory safety. However, these systems enforce their policies only at the IR level, not at the low level. Our system has to solve the key challenge of how to preserve security at the low-level even with the IR-level rewriting.

Portable Native Client (PNaCl) is an ongoing effort at Google. A white paper describes its initial design [13]. PNaCl requires code be transmitted in the LLVM IR format, with portability as the goal. After mobile IR code is downloaded into the Chrome browser, PNaCl compiles the IR code into SFI-compliant native code and reuses NaCl to constrain native code. The important difference between PNaCl and our system is that their architecture does not accommodate security optimizers that remove or hoist checks. The constraint language in our system allows optimizers to perform optimizations and attach constraints that can be checked at the low level.

2.2 Program Shepherding and Virtual Machines

Program shepherding utilizes an efficient program interpreter to enforce security at runtime [19, 27]. The interpreter can enforce various policies during program execution. Similarly, virtual machines either JIT or interpret high-level representations, enforcing relevant security policies during the process. Although many policies can be enforced conveniently in interpreters and virtual machines, the sheer size and complexity of interpreters or JIT compilers make their trustworthiness questionable [10, 18]. Furthermore, the runtime performance of interpreters and virtual machines might be problematic compared with the IRM approach. Strato incurs lower overhead and has a much smaller TCB.

3 Overview of Strato

This section elaborates on the workflow of Strato. We will discuss where checks are inserted and optimized, and where constraint checking and verification happen. We have used Strato to implement CFI and data sandboxing, two specific IRMs. Therefore, we first discuss those IRMs’ attack model and security policies.

Attack Model. Strato adopts CFI’s attack model [1]. We assume there is a separate code and data region. The data region is under the control of an attacker, who is modeled as a concurrent thread that can overwrite any memory location in the data region. This rather pessimistic assumption is actually realistic given the abundance of memory corruption vulnerabilities. In addition, we assume that the code region and machine registers cannot be changed by attackers. The assumption on the code region can be discharged by hardware protection such as DEP [22] or the $W \oplus X$ protection in latest x86 processors. The assumption about registers is consistent with kernel-based multithreading. Note that even though the attacker cannot directly modify the code region or registers, he/she may indirectly induce such a change. For instance, if a program loads from memory to a register, the register’s new value is controlled by the attacker since the data region is controlled by the attacker. If the program further uses the register as the address of a memory-write operation, the operation may change the code region since the register’s value is controlled by the attacker. Therefore, a protection mechanism must prevent such indirect effects from damaging the system.

Security policy. In Strato, we have implemented two IRMs: CFI and data sandboxing. The CFI policy is with respect to a control-flow graph, whose edges connect control-transfer instructions to allowed destination basic blocks. We say a program obeys the CFI policy if all control transfers in the program during runtime follow the control-flow graph.

The data-sandboxing policy restricts memory reads and writes. Following previous work [21, 32, 38], we place a *guard zone* immediately before the data region and another guard zone immediately after the data region. We use *gz_size* to denote the size of both guard zones. We assume that access to guard zones is hardware trapped (through page protection). Guard zones facilitate optimizations of data sandboxing. With the guard zones, a memory read or write is safe with respect to the data-sandboxing policy if its address is either in the data region or in the guard zones.

Workflow. In order for an IRM-implementation framework to be retargetable, the majority of instrumentation and optimizations need to be decoupled from a specific ISA. A high-level representation provides such a vehicle. A remaining question is where to insert the

IRM-instrumentation phase inside a compiler. A typical optimizing compiler has many layers that transform an IR program to another IR program with simplified semantics or better performance. Therefore, the IRM-instrumentation phase can be scheduled at any stage between IR generation and the backend.

On one end of the spectrum, we can schedule the IRM-instrumentation pass right after the compiler frontend; that is, after the IR is generated by the frontend. The benefit is that it can reuse a large number of existing IR-level optimization passes, which can optimize away unnecessary security checks. However, it has two drawbacks. First, since the security of low-level code generated by the compiler is what we are interested in, we need a way to ensure those IR-level optimizations do not wrongly optimize away security checks. One way to accomplish this would be to modify the optimizations to carry enough information to the low level for certification, similar to proof-carrying code. However, modifying complex compiler optimizations is non-trivial. The second and more serious drawback of scheduling the IRM-instrumentation pass right after the frontend is that existing optimizations may not be safe according to the attack model of an IRM. As we discussed before, the CFI attack model assumes that data memory is untrusted. However, a typical IR assumes a much different machine model. As an example, LLVM IR adopts an Unlimited Register Machine (URM) model in the SSA form [20]. A real machine has a limited number of registers and so LLVM IR variables may be spilled into untrusted memory locations. Therefore, if an LLVM IR optimization depends on the URM model for correctness, then the optimized result may not be safe according to CFI's attack model.

On the other end of the spectrum, the IRM-instrumentation phase can be scheduled right before the compiler backend for code generation. This is the design adopted in Strato. The downside is that existing compiler optimizations are not reused and we have to develop our own optimizations to optimize IRM checks to improve efficiency. But optimizations for security checks can be implemented straightforwardly at the LLVM IR level, which has a small number of instructions and is in the SSA form. For the policy of data sandboxing, we implemented three optimizations with ease, shared by all targets supported by LLVM. With this design, there is no need to trust or modify a large number of existing IR-level compiler optimizations. Optimizing compilers have a large code base and bugs are unavoidable [35].

Fig. 1 presents the workflow of Strato, which is implemented as extra passes added to the LLVM compiler. We next explain the steps of how source code is translated to low-level code through Strato-augmented LLVM. We add stars to those steps that are added in Strato to distinguish them from those steps already in LLVM.

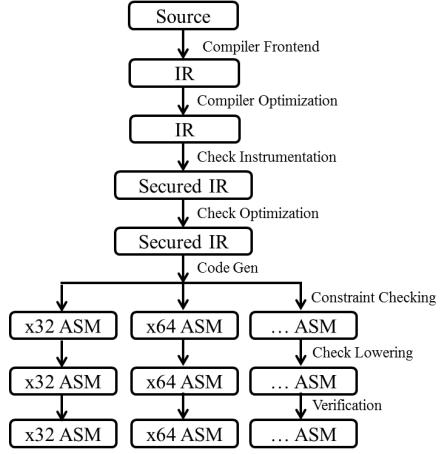


Figure 1: Workflow of Strato

- (1) Compiler frontend. LLVM's clang frontend generates the IR code.
- (2) Compiler optimizations. LLVM's transformations and optimizations change the IR code to simpler and optimized code.
- (3) *Check insertion. Security checks are inserted before dangerous instructions to generate secured IR code. The dangerous instructions and checks inserted depend on the security policy. Since the current policy is CFI and data sandboxing, security checks are inserted before memory loads and stores as well as computed jumps. Note this step inserts more checks than necessary. Later steps will remove unnecessary checks. Security checks are inserted as LLVM intrinsic functions, which will be lowered to machine-instruction sequences at a later step (if they are not optimized away).
- (4) *Check optimization. After check insertion, custom optimizations for removing security checks are performed on the IR code. We implement three effective optimizations to demonstrate the amenability of high-level IR to optimizations: redundant check elimination, sequential memory access optimization, and loop-based check optimization. Our optimizations differ from traditional ones in that no security checks are removed or moved around at this step. A check that is deemed unnecessary is marked as removable and constraints are attached to it. The check will be removed only after the constraints are checked to be valid at a later step. If those constraints are violated by later steps of the compilation, the check will not be removed.
- (5) Code generation. The compiler backend performs instruction selection, instruction scheduling, ISA-

- specific optimizations, and register allocation. Low-level assembly code is generated as the result.
- (6) *Constraint checking. If the constraints for a check are invalidated during compiler transformations and optimizations, the check is kept intact. Otherwise, it is removed.
 - (7) *Check lowering. Security checks are lowered to machine-instruction sequences. Usually a security check can be implemented by multiple machine-instruction sequences. This step therefore provides a convenient place to experiment with different sequences to evaluate which one produces the best performance.
 - (8) *Verification. An independent verifier is run to check the low-level code. If the verification fails, then the code is rejected.

The above design makes it straightforward to adapt to a different ISA. Steps including check insertion, check optimization, and constraint checking can be reused across ISAs. The check-lowering and the verifier components need to be tailored for a new ISA. As we will discuss, the amount of effort involved to retarget Strato’s implementation from x86-32 to x86-64 is small.

4 Check Instrumentation, Optimizations and the Constraint Language

The goal of security-check instrumentation is to guard dangerous operations with checks so that they cannot be abused by adversaries. For CFI, IDs are inserted before control-flow targets. Furthermore, checks are inserted before computed jumps, including indirect calls, indirect jumps and return instructions²; these checks ensure that the expected IDs are there at the targets of control-flow transfers [1].

For data sandboxing, Strato inserts a check before each load and store instruction; the check ensures that the instruction’s memory address is within the data region. In addition, after a definition of a pointer variable, a check is inserted to ensure that the pointer is within the data region. A check is also inserted at the entry of a function for a pointer parameter. In this step, Strato inserts more checks than necessary.

Since checks are inserted at the IR level, the same protection strategy is adopted for all machine targets, including x86-32 and x86-64. This provides uniformity and enables most of the code to be shared between targets. In contrast, NaCl adopts very different protection strategies for x86-32, x86-64, and ARM (segmentation on x86-32, large addresses and guard regions on x86-64, and address masking on ARM).

After check instrumentation, optimizations are run on the secured IR to mark unnecessary checks. To demonstrate the ease of implementing optimizations at the IR level, we have implemented three optimizations for removing unnecessary data-sandboxing checks: redundant check elimination, sequential memory access optimization, and loop-based optimization. In these optimizations, checks are not removed. Rather, checks that are deemed removable are marked and constraints are attached to them. Checks whose constraints are still valid after the backend processing are removed in the later constraint-checking step.

Redundant Check Elimination. Since the check-instrumentation step inserts a check after the definition of a pointer variable and also before the use of the variable via a load or store, the checks before the uses are redundant at the IR level. Fig. 2 presents an example. Column (a) presents the original C code and column (b) presents the LLVM IR code before check instrumentation. During check instrumentation, three checks are inserted. First, a check is placed at the beginning of the function for the pointer parameter `ptr`. Second, since there are a load in block labeled `else` and a store in block labeled `then`, checks need to be placed before them. `check2` and `check3` are unnecessary at the IR level. However, they cannot be removed in the IR code because `ptr.safe` might be spilled into the untrusted stack during register allocation. Instead, `check2` is marked as removable and a constraint is attached, specifying that the check can be removed if and only if `ptr.safe` is not spilled between `check1` and `check2`. In Fig. 2(c), constraints are at lines starting with the `#` symbol. `check3` in block `then` is attached with a similar constraint. After register allocation, the constraint checker checks whether `ptr.safe` has been spilled. If not, the two checks are removed. Otherwise, they are kept intact.

LLVM IR is in the SSA form, making it easy to implement the above optimization. First, the def-use chain is explicit in the SSA form. Furthermore, the SSA form ensures that `ptr.safe` is not modified between `check1` and `check2`. By contrast, if carried out on machine code, the optimizer would have to perform dataflow analysis to determine whether a pointer has been guarded and modified.

Sequential Memory Access Optimization. Most programming languages support aggregate types such as structs in C and classes in C++. A common pattern exists in member accesses: a base pointer plus a constant offset is used to visit a specific member. With the guard zones before and after the data region, a memory access with a base pointer and a constant offset is safe as long as the base pointer is within the data region and the offset is smaller than the guard-zone size. This observation can be exploited to remove checks if members of an object

(a) Original C code	(b) Unsecured IR code	(c) Secured and optimized IR code
<pre>int foo (int v, int *ptr) { int tmp = 0; if (v > 47) *ptr = v; else tmp = *ptr; return tmp; }</pre>	<pre>entry: tmp = 0 if(v > 47) goto then else: tmp = load *ptr goto end then: store v, *ptr end: ret tmp</pre>	<pre>entry: ptr.safe = call guard(ptr) // check1 tmp = 0 if(v > 47) goto then else: ptr.safe1 = call guard(ptr.safe) // check2 # noSpill(ptr.safe, check1, check2) tmp = load *ptr.safe1 goto end then: ptr.safe2 = call guard(ptr.safe) // check3 # noSpill(ptr.safe, check1, check3) store v, *ptr.safe2 end: ret tmp</pre>

Figure 2: An example for illustrating redundant check elimination. `guard` is the security check to ensure that the input pointer is in the data region, which is implemented as an LLVM intrinsic function.

(a) Original C code	(b) Unsecured IR code for sum	(c) Secured and optimized IR code
<pre>struct s { long x; long y; }; int sum (struct s *p) { return p->x + p->y; }</pre>	<pre>x = gep p, 0, 0 tmp1 = load *x y = gep p, 0, 1 tmp2 = load *y sum = add tmp1, tmp2 ret sum</pre>	<pre>p.safe = call guard(p) // check1 x = gep p.safe, 0, 0 x.safe = call guard(x) // check2 # noSpill(p.safe, check1, check2) # sizeof(struct s)*0 + sizeof(long)*0 < gz_size tmp1 = load *x.safe y = gep p.safe, 0, 1 y.safe = call guard(y) // check3 # noSpill(p.safe, check1, check3) # sizeof(struct s)*0 + sizeof(long)*1 < gz_size tmp2 = load *y.safe sum = add tmp1, tmp2 ret sum</pre>

Figure 3: An example for illustrating sequential memory access optimization.

are visited sequentially and the base pointer is shared by multiple visits.

In LLVM IR, the `getelementptr` instruction takes a base pointer and multiple indices as operands and is used to compute the address of a sub-element of an aggregate data structure. If the base pointer has been guarded, the offset is a constant, and the offset is determined to be smaller than the guard zone size, then the pointer computed by `getelementptr` does not need to be guarded again. However, the size of each member cannot be determined at the IR level because it may be target dependent. For example, type `long` takes 4 bytes in x86-32 and 8 bytes in x86-64. As a result, the check after the definition of a pointer variable through `getelementptr` can be marked as removable and attached with constraints specifying that the base pointer cannot be spilled and the final offset from the base pointer should be less than the guard-zone size.

Fig. 3 presents an example. In column (a), `struct s` contains two `long` members and the function `sum` computes their sum. In column (c), Strato inserts `check1` at the entry to function `sum` and `check2` and `check3` after each `getelementptr` instruction (abbreviated as `gep` in the figure). The constraints for `check2` specify that it can be removed if there is no spill between `check1` and `check2` for pointer `p.safe` and the offset `sizeof(struct s)*0 + sizeof(long)*0` is smaller than the guard-zone size (this condition can be determined to be true even at the IR level because it is always zero; however, values of other expressions may be target dependent). Similar constraints are attached for `check3`.

Loop-based check optimization. Loop optimization is important because programs tend to spend the majority of runtime in loops. Performance is improved if a security check inside a loop can be hoisted outside the loop. For example, if a pointer is not modified inside

(a) Original C code	(b) Unsecured IR code	(c) Secured and optimized IR code
<pre>long sum (long *ar, long len) { long rst = 0, i; for (i=0; i<len; ++i) rst += ar[i]; return rst; }</pre>	<pre>rst = 0 i = 0 if (len <= 0) goto end for.body: ptr = gep ar, i tmp = load *ptr rst += tmp i += 1 if (i >= len) goto end goto for.body: end: ret rst</pre>	<pre>rst = 0 i = 0 ar.safe = call guard(ar) // check1 if (len <= 0) goto end for.body: ptr = gep ar.safe, i ptr.safe = call guard(ptr) // check2 # noSpill (ar.safe, check1, check2) # noSpill (i, check1, check2) # sizeof(long) * 1 < gz_size tmp = load *ptr.safe rst += tmp i += 1 if (i >= len) goto end goto for.body: end: ret rst</pre>

Figure 4: An example for illustrating loop optimization.

the loop, then a check for the pointer can be hoisted. As another example, if a pointer is incremented or decremented for a small stride (less than the guard-zone size) inside the loop, and there is a memory access through the pointer in the loop, then the check can be hoisted. The reason this is safe is because access to the guard zones is trapped; if the initial value of the pointer is checked to be inside the data region, then the change to the pointer in one loop iteration will make the pointer to be either in the data region or in guard zones and the access through the pointer serves as a check. This optimization follows the loop optimization described by Zeng et al. [38]; please refer to that paper for detailed analysis of the soundness of the optimization. As another optimization example, if there is a pointer that is calculated from the induction variable of the loop, the increment to the induction variable is a small stride (less than the guard zone size), and there is a memory access through the pointer in the loop, then the check can be hoisted. LLVM IR encodes loop information explicitly, making it easy to detect induction variables and strides.

In our optimizations, a hoistable check is not moved. Instead, a new check is inserted into the loop preheader and the old check is marked as removable and attached with constraints. An important constraint is that the relevant pointer cannot be spilled. Fig. 4 presents a concrete example. The program in column (a) adds elements in array `ar`. The program visits memory once per iteration. The check can be hoisted if and only if the induction variable is the only variable used to calculate the memory location and the stride is smaller than the guard-zone size and there is a memory visit using the memory location in every path inside the loop.

```
constraint ::= noSpill [var, src, dst]
            | term comparator term
term ::= term + term | term * term |
       var | constant | gz.size
comparator ::= < | > | == | >= | <=
```

Figure 5: Syntax of the constraint language.

Summary about optimizations. The three optimizations demonstrate that a high-level IR can simplify the implementations of optimizations, which are reused among target ISAs. Additional optimizations enabled by a high-level IR can further decrease the runtime cost of Strato.

The constraint language. For completeness, we present the syntax of the constraint language in Fig. 5. A check may be attached with one or more constraints. All constraints need to be satisfied in order for the check to be removed; that is, there is an implicit conjunction when interpreting a list of constraints.

The constraint `noSpill [var, src, dst]` denotes that `var` cannot be spilled between `src` and `dst`, where `src` and `dst` are program locations. Note the semantics of this constraint is that the variable cannot be spilled along *every* control-flow path from `src` to `dst`. Therefore, the `noSpill` constraints in the example of Fig. 5 effectively require that `ar.safe` and `i` cannot be spilled in the entire loop. The *comparison constraint* “`term1 comparator term2`” represents a relation between `term1` and `term2`. It can be used to encode the constraint that a constant offset should be less than the guard-zone size, as in Fig. 3.

The design of the constraint language depends on what optimizations Strato supports and also LLVM’s back-

end. For instance, the `noSpill` constraint is there because LLVM’s backend may break this assumption by spilling variables to memory. An IR-level optimization may check more conditions. But if there is no possibility for the backend to break a condition, that condition does not need to be encoded and propagated. For instance, in loop optimizations, the condition that there must be a memory access through the pointer in question can be checked at the IR-level alone. Another point is that it is possible new optimizations may require adding new predicates to the constraint language. We believe the constraint language can be extended straightforwardly.

5 Constraint Checking and Check Lowering

After LLVM’s backend processing, Strato performs constraint checking and check lowering. Constraint checking examines the constraints attached to each check and checks whether they are valid. If the constraints are valid, the check is removed. If they are invalid, the check is lowered to a sequence of machine instructions.

Our constraint language is designed so that constraints can be checked straightforwardly at the low level, with the help of information preserved by LLVM. For example, a comparison constraint becomes constant expressions at the low level because after fixing the ISA sizes of types become constants. To check a `noSpill` constraint, the constraint checker first identifies the correspondence between IR variables and registers (LLVM preserves enough information for this purpose) and uses data-flow analysis to check whether the register that corresponds to the IR variable is moved to memory between the source and destination locations.

Remaining checks are lowered to machine-instruction sequences. A high-level check can be implemented by many machine-instruction sequences. The overhead of different sequences varies. Strato makes it easy to try different machine-instruction sequences—only the check-lowering step needs to be modified. We have evaluated a large number of machine-instruction sequences for checks in CFI and data sandboxing. We discuss examples of possible sequences next, but leave the discussion about the performance overhead of various sequences to the evaluation section.

ID encodings. Strato’s CFI instrumentation requires the encoding of IDs at control-flow targets. ID-encoding instructions have to satisfy two conditions. First, the instruction must take a long immediate value as an operand, which is used to encode the ID. Second, the instruction cannot introduce side-effects that change the semantics of the program. The original CFI uses `prefetch` instructions for encoding IDs. We have evaluated a large

number of alternative instructions that satisfy the two conditions. They are put into three groups. Instructions in the first group take an immediate value and assign it to a machine register. For example, “`movl $ID, %eax`” assigns the immediate value `ID` to register `eax`. It can be used to encode the `ID` as long as `eax` is dead at the point where the instruction is inserted.³. Instructions in the second group perform arithmetic operations on a register with the `ID` and assign the result to a register. For example, “`add $ID, %eax`” can be used to encode the `ID` as long as `eax` and the flags register are dead. Instructions in the last group take a register and the `ID` value and defines only the flags register. For example, “`cmp $ID, %eax`” can be used as long as the flags register is dead at the point of insertion.

6 Verification

Compiler optimizations and transformations are untrustworthy because compilers have a large code base and are buggy [26, 36]. Bugs in optimizations that remove security checks are even harder to catch because they do not crash programs but introduce vulnerabilities silently. To remove those optimizations out of the TCB, Strato includes a verifier at the end of the compilation pipeline to validate the assembly output of the compiler.

The verifier checks if the assembly code satisfies the CFI and data sandboxing policy. CFI verification is straightforward. The LLVM assembly preserves enough information to reconstruct a control-flow graph, which is used by the verifier to check if necessary ID-encoding and ID-checking instructions are there in the assembly. Data-sandboxing verification is more challenging because Strato’s optimizers may remove or hoist checks. Strato’s verifier follows the design of a previous verifier [38] to implement range analysis for data-sandboxing verification (with improvements; see below). The basic idea is to compute the ranges of registers at all program points and check if the ranges of memory addresses fall into the data region plus guard zones. The calculation of ranges uses a standard iterative algorithm until a fixed point is reached.

Strato’s verifier improves the previous range-analysis verifier by adding path sensitivity. Ranges of registers may be different along different paths after a sequence of comparison and jump instructions. As an example, the assembly snippet in AT&T syntax in Fig. 6 is extracted from 175.vpr in SPECint2000. The range of register `eax` shrinks down to the data region after the `andl` masking, where `$DATA_MASK` is the constant mask for the data region. The `movl` instruction expands the range of `eax` to `[bottom, top]` because it loads from untrusted memory.⁴ Without path sensitivity, the range of `eax` would remain `[bottom, top]` at the entry to the two

```

        andl $DATA_MASK, %eax
        movl (%eax), %eax
        cmpl $3, %eax
        ja .LBB5_8
        movl *._LJTI(%eax,4), %eax
.LBB2:
        ...
.LBB8:
        ...

```

Figure 6: An example for illustrating path sensitivity in range analysis.

```

popl %ecx
cmp1 $ID, 1(%ecx)
jne error
jmp1 *%ecx

```

Figure 7: A wrong CFI sequence for return instructions.

successor blocks labeled with .LBB2 and .LBB8; consequently, the verifier would report an out-of-range error on the `movl` instruction because its memory address is “`.LJTI + eax*4`”, where `.LJTI` is a constant in the data region. With path sensitivity, the verifier computes the range of `eax` to be `[0, 3]` before `movl` and successfully validates that the address is within the data region plus guard zones. The verifier in the previous system [38] used instruction pattern matching to verify this pattern. By adopting a path-sensitive analysis, Strato’s verifier is more general and can verify all security-check optimizations we have presented.

With the help of the verifier, we discovered 35 subtle bugs in early versions of Strato. Those bugs would be hard to discover otherwise. We classify the bugs into three groups as follows.

- Bugs in CFI instrumentation code. CFI implementation inserts IDs at branch targets and check instructions before computed jumps. The check instructions need to load the IDs at target locations. Therefore, they visit memory and need to be sandboxed as well according to data sandboxing. As an example, the snippet in Fig. 7 contains an early version of a CFI check sequence for return instructions. The return address is popped into register `ecx`, which is then used to load the ID for comparison. The `cmp1` instruction visits the code region using address `ecx+1`, which is unsafe because it is from the untrusted stack. Our verifier successfully caught this bug and we fixed the sequence by inserting a data-masking instruction on `ecx` before `cmp1`.
- Bugs in the source program. Our verifier even

found a bug in the source code of `253.perlrbmk`, a program in SPECint2000. The bug is a possible null-pointer dereference. The `perlrbmk` program has its own `malloc` function, which can return a null pointer when a memory allocation fails and the `malloc` is inlined by the compiler. A null pointer is represented as value 0 and is outside the range of data region plus guard zones. It was caught by the verifier. We modified the source code of `perlrbmk` to fix this bug.

- Bugs in LLVM intrinsic functions such as `llvm.memset` and `llvm.memcpy`. LLVM synthesizes programs into intrinsic function calls as an optimization. These intrinsic function calls can be lowered into a sequence of machine instructions or direct calls to the library functions depending on the tradeoff between code size and the function-call overhead. At the IR level, there is no way to predict how an intrinsic function will be lowered. If they are lowered into machine-instruction sequences, then their pointer arguments need to be sandboxed as they may visit memory. Our verifier caught these bugs and we fixed those machine-instruction sequences to insert data-masking instructions for pointers.

The combined verifier for x86-32 and x86-64 contains approximately 7k LOC including white space lines, comments, and debug statements. The majority of the code is a giant switch table for all the machine instructions that the verifier has to support (In LLVM 2.9, x86-64 target contains 3,747 machine instructions). The size of the verifier is a concern and we leave its verification for future work [23].

Finally, we note that our verifier performs verification at the assembly level. We use it mostly to catch bugs in the Strato compiler so that the compiler is out of the TCB. A more desirable design is to implement a verifier for binaries directly. This is actually a matter of engineering: we just need to modify the assembler to encode the control-flow graph as extra information in a binary so that the binary-level verifier can disassemble the binary reliably; the kinds of verification tasks involved are the same after disassembly.

7 Implementation and Evaluation

We have implemented Strato on top of LLVM 2.9. The check-instrumentation step is implemented as a pass and scheduled after LLVM’s IR optimization passes. The security-check optimizations are performed right after check instrumentation. The constraint checking and check lowering are implemented in one pass in the

compiler backend after register allocation. The range-analysis based verifier is scheduled at the very end in the backend. Constraints are encoded as LLVM *metadata*. In total, the instrumentation and optimizations consist of approximately 3,750 lines of C++ code, shared between x86-32 and x86-64. The constraint checking and check-lowering component has 1,420 lines of C++ code with an additional 180 lines added for x86-64. The verifier includes 6,960 lines of C++ code, with 1,240 lines added for x86-64.

The object code after data-sandboxing and CFI instrumentation cannot run directly and needs specialized linker scripts and a specialized loader. We have developed linker scripts for both C and C++ programs targeting x86-32 and x86-64. The linker scripts link object code generated by LLVM to three sections (including code, data, and read-only data) at specified addresses. We have also developed a loader that loads various sections in a binary at specified addresses in the address space and sets up appropriate protection for those sections using the `mprotect` system call. We reused PittS-Field's library wrappers and libraries for x86-32 [21]; we also adapted them for x86-64.

For evaluation, we have built and run the benchmark suites `bakeoff` and `SPECint2000` in Strato. The `bakeoff` benchmark suite contains three programs: `hotlist`, `lld`, and `md5`. It has been used by previous code-sandboxing frameworks for evaluation [14, 16, 31]. `SPECint2000` contains twelve computation-intensive programs and is widely used for compiler evaluation. All benchmark programs in `bakeoff` and `SPECint2000` can be successfully compiled in Strato. All programs were compiled with the `-O3` full optimization level except for `254.gap` in `SPECint2000`, which ran correctly only with `-O0` enabled due to bugs in LLVM 2.9's optimizations. All experiments were conducted on a Ubuntu 11.10 box with an Intel Core 2 Duo CPU at 3.16GHz and 8GB of RAM. Experiments were averaged over three runs and the standard deviation was less than two percent of the arithmetic mean.

Security benefits. Security benefits of Strato depend on what IRMs have been incorporated. The current implementation supports CFI and data sandboxing. The low-level output of all programs in `bakeoff` and `SPECint2000` can be successfully verified by Strato's verifier. Assuming the verifier is correct, the compiled code of those benchmark programs satisfies the CFI and data sandboxing policy.

CFI and data sandboxing come with well-documented security benefits. The original CFI work discusses that CFI can block a test suite of 18 attack vectors, as well as some heap-overflow attacks [3]. Data sandboxing as in SFI is effective in isolating faults in untrusted modules, as reported in NaCl [37] and Robusta [30].

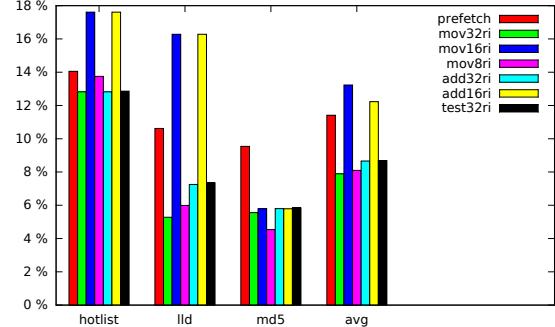


Figure 8: Performance overhead for CFI with various ID-encoding instructions on `bakeoff` programs.

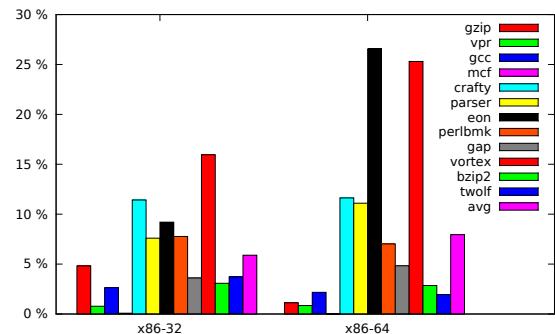


Figure 9: Performance overhead for CFI with `movl` as the ID-encoding instruction on `SPECint2000`.

CFI and data sandboxing cannot prevent all attacks. For example, non-control data attacks [7] cannot be prevented if critical data are stored inside the sandbox. These attacks can corrupt the data without violating a control-flow graph. However, another IRM called Data-Flow Integrity (DFI) [4] can prevent such attacks.

Performance evaluation. IRMs insert runtime checks into programs and slow down program execution. We present the performance overhead as the percentage of execution-time increase of instrumented programs compared with uninstrumented programs.

We first evaluated the performance implication of alternative machine-instruction sequences that implement the same high-level checks. We have tested a large number of alternative ID-encoding instructions, classified into three groups discussed in Sec 5. Fig. 8 presents the runtime overhead of various ID-encoding instructions on `bakeoff` programs for x86-32 when enforcing the CFI policy. In the figure, color bars are used for different ID encodings. The figure presents only a subset of what we have tried due to space limit. In addition, we evaluated different ID lengths such as 32-bit IDs, 16-bit IDs and 8-bit IDs. Shorter IDs do not necessarily have better performance and they shrink the space for IDs. In the figure, we use `mov32ri` to represent the case of using a 32-bit `mov` instruction that moves a 32-bit immediate value to a general register. As can be seen from the figure, most ID-encoding instructions are more efficient than `prefetch`,

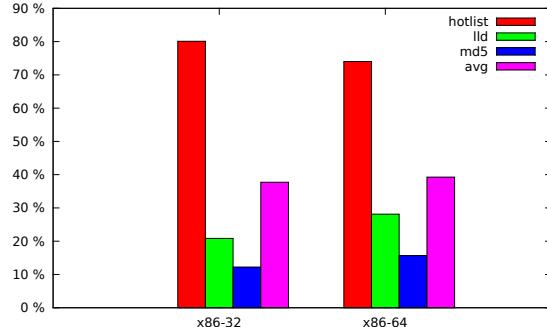


Figure 10: Performance overhead for CFI combined with data sandboxing for both reads and writes on bakeoff.

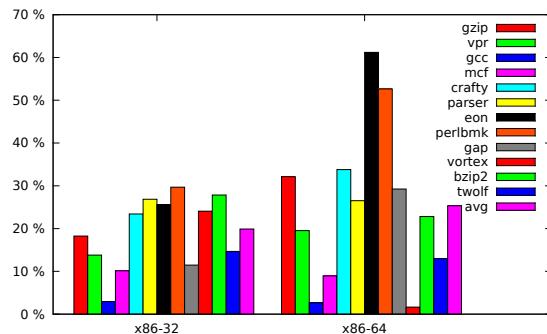


Figure 11: Performance overhead for CFI combined with data sandboxing for both reads and writes on SPECint2000.

the one used in the original CFI implementation.⁵ The case of 32-bit `movl` instruction has the lowest runtime overhead.

Fig. 9 presents the performance overhead of CFI alone on SPECint2000 with `movl` as the ID-encoding instruction. CFI incurs an average of 5.89% and 7.95% slowdown on x86-32 and x86-64, respectively. Our CFI implementation is competitive with previous CFI systems. The original CFI work [3] has 15% overhead and our own previous work [38] has 7.7% overhead on x86-32.

Fig. 10 and Fig. 11 present the overhead of enforcing CFI and data sandboxing for bakeoff and SPECint2000 programs, respectively. Both cases of x86-32 and x86-64 are presented. The numbers are with respect to the case of using the `mov32ri` instruction as the ID encoding in CFI, and using the `and` instruction for sandboxing memory addresses. On average, Strato incurs 37.7% on x86-32 and 39.3% on x86-64 for bakeoff programs, and 19.9% on x86-32 and 25.3% on x86-64 for SPECint2000 programs. The high overhead on `hotlist` is due to the two checks in the inner loop of two nested loops and they cannot be optimized away or hoisted.

Strato’s performance is competitive with previous SFI/CFI systems. We note most previous systems sandbox only memory writes for protecting integrity, but not memory reads for protecting confidentiality. There are many more memory reads than writes in programs.

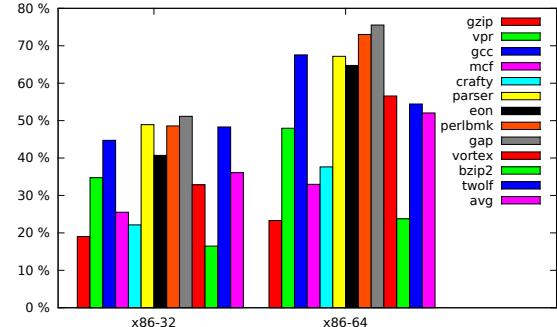


Figure 12: Code size increase on SPECint2000.

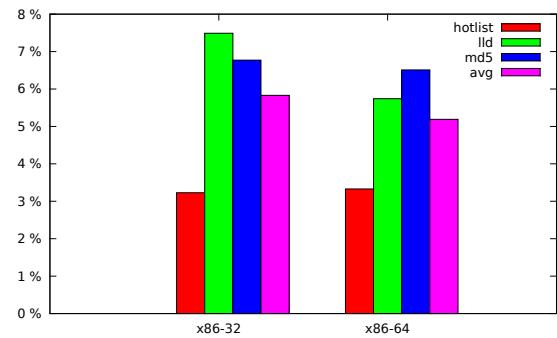


Figure 13: Code size increase on bakeoff.

Strato’s data sandboxing protects both memory reads and writes. There are two other SFI/CFI systems that sandbox both reads and writes. XFI’s average performance overhead for bakeoff programs are 53.7% on x86-32; in comparison, Strato’s overhead is 37.7%. XFI does not report performance results for SPECint2000. Our previous system [38] also sandboxes both reads and writes at the assembly level. It reports an average overhead of 27.2% for x86-32; in comparison, Strato’s overhead is 19.9%. We believe that the performance difference is because Strato’s optimizations can take advantage of structured information available at the IR level. For instance, the previous system uses the dominator-tree analysis to recover loops and induction variables at the assembly level, while LLVM IR tells explicitly where loops and induction variables are. In summary, Strato provides competitive performance and provides retargetability, lacked by previous systems.

Code Size. We measured the code-size increase on SPECint2000 and bakeoff programs. Strato does not alter the data sections of programs. It increases the size of text sections by inserting extra security checks. Fig. 12 and Fig. 13 present the text-section size increase on SPEC CPU2000 and bakeoff programs, respectively. On average, the text section grows 36.10% on x86-32 and 52.05% on x86-64 for SPECint2000 programs, and 5.83% on x86-32 and 5.19% on x86-64 for bakeoff programs. Text-section size inflates more on SPECint2000 because it contains larger programs with many more

functions; the compiler aligns functions on boundaries. Although disk space is not a major problem in a typical computing environment, it may matter in embedded systems. Benchmark programs were compiled with `-O3`, which is optimized for runtime performance, not for binary size. If the binary size is a major concern, programs can be optimized with `-Os`, which uses shorter instruction sequences.

Memory Usage. We also evaluated the memory usage of Strato. The memory-footprint increase for benchmark programs is negligible since Strato does not change the data memory. It increases memory footprint only through the code-size increase, but the code section takes a small fraction of total runtime memory.

8 Conclusions and Future Work

Conclusions. We have introduced an IRM framework to enforce low-level security policies by working on a high-level intermediate representation. For retargetability, Strato performs its instrumentation and optimizations on a high-level IR, which brings the benefits of structured information and a small instruction set. In addition, we have designed techniques that deal with problems that might arise due to backend transformations and optimizations. A constraint language is proposed to propagate invariants across the backend for validation. Furthermore, a path-sensitive verifier is implemented to verify the final output of the whole framework. Our experimental results show that the framework’s performance is competitive with previous systems. Our framework explores an alternative design point of how low-level IRMs can be implemented. This design point provides retargetability, performance, trustworthiness, and ease of implementation.

Future work. There are many other low-level IRMs that we can incorporate into Strato, including data-flow integrity [4] and fine-grained access control of memory [5]. The workflow of Strato is general enough to accommodate those IRMs, but individual components such as check insertion and lowering need to be updated for a particular IRM. We are interested in designing a check-optimization engine that can be shared by many IRMs; for example, optimizations such as redundant check elimination can be shared. Ultimately, we are interested in generalizing Strato so that it is parametrized by a policy specification, which guides the phases of check insertion, optimization, and lowering.

9 Acknowledgments

We thank Cliff Biffle for providing us with his initial development, useful documents, and suggestions.

We also thank Mengtao Sun for his help to the project and anonymous reviewers for their insightful comments. This research is supported by US NSF grants CCF-0915157, CCF-1149211, CCF-1217710, and a research award from Google.

References

- [1] ABADI, M., BUDIU, M., ERLINGSSON, U., AND LIGATTI, J. Control-flow integrity. In *Proceedings of the 12th ACM conference on Computer and communications security* (New York, NY, USA, 2005), CCS ’05, ACM, pp. 340–353.
- [2] ABADI, M., BUDIU, M., ERLINGSSON, Ú., AND LIGATTI, J. A theory of secure control flow. In *ICFEM* (2005), K.-K. Lau and R. Banach, Eds., vol. 3785 of *Lecture Notes in Computer Science*, Springer, pp. 111–124.
- [3] ABADI, M., BUDIU, M., ERLINGSSON, U., AND LIGATTI, J. Control-flow integrity: principles, implementations, and applications. *ACM Trans. Inf. Syst. Secur.* 13, 1 (Nov. 2009), 4:1–4:40.
- [4] CASTRO, M., COSTA, M., AND HARRIS, T. Securing software by enforcing data-flow integrity. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (2006), pp. 147–160.
- [5] CASTRO, M., COSTA, M., MARTIN, J.-P., PEINADO, M., AKRITIDIS, P., DONNELLY, A., BARHAM, P., AND BLACK, R. Fast byte-granularity software fault isolation. In *ACM SIGOPS Symposium on Operating Systems Principles (SOSP)* (2009), pp. 45–58.
- [6] CHECKOWAY, S., DAVI, L., DMITRIENKO, A., SADEGHI, A.-R., SHACHAM, H., AND WINANDY, M. Return-oriented programming without returns. In *Proceedings of the 17th ACM conference on Computer and communications security* (New York, NY, USA, 2010), CCS ’10, ACM, pp. 559–572.
- [7] CHEN, S., XU, J., SEZER, E. C., GAURIAR, P., AND IYER, R. K. Non-control-data attacks are realistic threats. In *In USENIX Security Symposium* (2005), pp. 177–192.
- [8] CYTRON, R., FERRANTE, J., ROSEN, B. K., WEGMAN, M. N., AND ZADECK, F. K. An efficient method of computing static single assignment form. In *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (New York, NY, USA, 1989), POPL ’89, ACM, pp. 25–35.
- [9] CYTRON, R., FERRANTE, J., ROSEN, B. K., WEGMAN, M. N., AND ZADECK, F. K. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.* 13, 4 (Oct. 1991), 451–490.
- [10] DEAN, D., FELTEN, AND WALLACH. Java security: From hot-jar to netscape and beyond. In *Proceedings of the 1996 IEEE Symposium on Security and Privacy* (Washington, DC, USA, 1996), SP ’96, IEEE Computer Society, pp. 190–.
- [11] DHURJATI, D., AND ADVE, V. S. Backwards-compatible array bounds checking for C with very low overhead. In *ICSE* (2006), pp. 162–171.
- [12] DHURJATI, D., KOWSHIK, S., AND ADVE, V. S. SAFECode: enforcing alias analysis for weakly typed languages. In *PLDI* (2006).
- [13] DONOVAN, A., MUTH, R., CHEN, B., AND SEHR, D. PNaCl: Portable Native Client Executables (white paper). http://src.chromium.org/viewvc/native_client/data/site/pnacl.pdf, 2010.
- [14] ERLINGSSON, U., ABADI, M., VRABLE, M., BUDIU, M., AND NECULA, G. C. XFI: software guards for system address spaces.

- In *Proceedings of the 7th symposium on Operating systems design and implementation* (Berkeley, CA, USA, 2006), OSDI '06, USENIX Association, pp. 75–88.
- [15] ERLINGSSON, U., AND SCHNEIDER, F. B. IRM enforcement of java stack inspection. In *Proceedings of the 2000 IEEE Symposium on Security and Privacy* (Washington, DC, USA, 2000), SP '00, IEEE Computer Society, pp. 246–.
- [16] ERLINGSSON, U., AND SCHNEIDER, F. B. SASI enforcement of security policies: a retrospective. In *Proceedings of the 1999 workshop on New security paradigms* (New York, NY, USA, 2000), NSPW '99, ACM, pp. 87–95.
- [17] EVANS, D., AND TWYMAN, A. Flexible policy-directed code safety. In *IEEE Symposium on Security and Privacy (S&P)* (1999), pp. 32–45.
- [18] GOVINDAVAJHALA, S., AND APPEL, A. W. Using memory errors to attack a virtual machine. In *Proceedings of the 2003 IEEE Symposium on Security and Privacy* (Washington, DC, USA, 2003), SP '03, IEEE Computer Society, pp. 154–.
- [19] KIRIANSKY, V., BRUENING, D., AND AMARASINGHE, S. P. Secure execution via program shepherding. In *Proceedings of the 11th USENIX Security Symposium* (Berkeley, CA, USA, 2002), USENIX Association, pp. 191–206.
- [20] LATTNER, C., AND ADVE, V. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)* (Palo Alto, California, Mar 2004).
- [21] MCCAMANT, S., AND MORRISETT, G. Evaluating SFI for a CISC architecture. In *Proceedings of the 15th conference on USENIX Security Symposium - Volume 15* (Berkeley, CA, USA, 2006), USENIX-SS'06, USENIX Association.
- [22] MICROSOFT. A detailed description of the data execution prevention (dep) feature in windows xp service pack 2, windows xp tablet pc edition 2005, and windows server 2003, September 2006.
- [23] MORRISETT, G., TAN, G., TASSAROTTI, J., TRISTAN, J.-B., AND GAN, E. Rocksalt: Better, faster, stronger SFI for the x86. In *ACM Conference on Programming Language Design and Implementation (PLDI)* (2012), pp. 395–404.
- [24] NAGARAKATTE, S., ZHAO, J., MARTIN, M. M. K., AND ZDANCEWIC, S. Softbound: highly compatible and complete spatial memory safety for C. In *PLDI* (2009), pp. 245–258.
- [25] NECULA, G., MCPEAK, S., AND WEIMER, W. CCured: type-safe retrofitting of legacy code. In *29th ACM Symposium on Principles of Programming Languages (POPL)* (2002), pp. 128–139.
- [26] REGEHR, J. The future of compiler correctness, August 2010.
- [27] SCOTT, K., AND DAVIDSON, J. Safe virtual execution using software dynamic translation. In *Annual Computer Security Applications Conference* (2002), pp. 209–218.
- [28] SEHR, D., MUTH, R., BIFFLE, C., KHIMENKO, V., PASKO, E., SCHIMPFF, K., YEE, B., AND CHEN, B. Adapting software fault isolation to contemporary cpu architectures. In *Proceedings of the 19th USENIX conference on Security* (Berkeley, CA, USA, 2010), USENIX Security'10, USENIX Association, pp. 1–1.
- [29] SHACHAM, H. The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86). In *Proceedings of the 14th ACM conference on Computer and communications security* (New York, NY, USA, 2007), CCS '07, ACM, pp. 552–561.
- [30] SIEFERS, J., TAN, G., AND MORRISETT, G. Robusta: Taming the native beast of the JVM. In *17th ACM Conference on Computer and Communications Security (CCS)* (2010), pp. 201–211.
- [31] SMALL, C. A tool for constructing safe extensible c++ systems. In *Proceedings of the 3rd conference on USENIX Conference on Object-Oriented Technologies (COOTS) - Volume 3* (Berkeley, CA, USA, 1997), COOTS'97, USENIX Association, pp. 13–13.
- [32] WAHBE, R., LUCCO, S., ANDERSON, T. E., AND GRAHAM, S. L. Efficient software-based fault isolation. In *Proceedings of the fourteenth ACM symposium on Operating systems principles* (New York, NY, USA, 1993), SOSP '93, ACM, pp. 203–216.
- [33] WANG, Z., AND JIANG, X. Hypersafe: A lightweight approach to provide lifetime hypervisor control-flow integrity. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy* (Washington, DC, USA, 2010), SP '10, IEEE Computer Society, pp. 380–395.
- [34] WARELL, R., MOHAN, V., HAMLEN, K. W., AND LIN, Z. Securing untrusted code via compiler-agnostic binary rewriting. In *Proceedings of the 28th Annual Computer Security Applications Conference* (2012), pp. 299–308.
- [35] YANG, X., CHEN, Y., EIDE, E., AND REGEHR, J. Finding and understanding bugs in c compilers. *SIGPLAN Not.* 46, 6 (June 2011), 283–294.
- [36] YANG, X., CHEN, Y., EIDE, E., AND REGEHR, J. Finding and understanding bugs in C compilers. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation* (New York, NY, USA, 2011), PLDI '11, ACM, pp. 283–294.
- [37] YEE, B., SEHR, D., DARDYK, G., CHEN, J. B., MUTH, R., ORM, T., OKASAKA, S., NARULA, N., FULLAGAR, N., AND INC, G. Native client: A sandbox for portable, untrusted x86 native code. In *In Proceedings of the 2007 IEEE Symposium on Security and Privacy* (2009).
- [38] ZENG, B., TAN, G., AND MORRISETT, G. Combining control-flow integrity and static analysis for efficient and validated data sandboxing. In *18th ACM Conference on Computer and Communications Security* (Oct. 2011), ACM.

Notes

¹The name Strato comes from stratosphere, which is an intermediate layer of Earth's atmosphere that contains ozone absorbing ultraviolet light from the Sun.

²Return instructions are changed to a sequence of pop, check, and indirect jump instructions to prevent a concurrent attacker from modifying the stack after the check.

³eax is a caller-saved register and is dead at the entry to a function. Furthermore, dead registers can be identified through liveness analysis

⁴In our implementation, bottom is 0 and top is the maximum unsigned integer, which is $2^{32} - 1$ for x86-32 and $2^{64} - 1$ for x86-64.

⁵Since the original CFI work, some versions of Intel and AMD hardware changed the behavior of prefetch; it becomes more expensive, since it pulls in TLB entries. As a result, choice of IDs used in prefetch greatly affects its runtime cost.

On the Security of Picture Gesture Authentication

Ziming Zhao^{†‡} Gail-Joon Ahn^{†‡} Jeong-Jin Seo[†] Hongxin Hu[§]

[†]*Arizona State University* [‡]*GFS Technology, Inc.* [§]*Delaware State University*
{zzhao30,gahn,jseo15}@asu.edu *hhu@desu.edu*

Abstract

Computing devices with touch-screens have experienced unprecedented growth in recent years. Such an evolutionary advance has been facilitated by various applications that are heavily relying on multi-touch gestures. In addition, picture gesture authentication has been recently introduced as an alternative login experience to text-based password on such devices. In particular, the new Microsoft Windows 8™ operating system adopts such an alternative authentication to complement traditional text-based authentication. In this paper, we present an empirical analysis of picture gesture authentication on more than 10,000 picture passwords collected from over 800 subjects through online user studies. Based on the findings of our user studies, we also propose a novel attack framework that is capable of cracking passwords on previously unseen pictures in a picture gesture authentication system. Our approach is based on the concept of selection function that models users' password selection processes. Our evaluation results show the proposed approach could crack a considerable portion of collected picture passwords under different settings.

1 Introduction

Using text-based passwords that include alphanumerics and symbols on touch-screen devices is unwieldy and time-consuming due to small-sized screens and the absence of physical keyboards. Consequently, mobile operating systems, such as iOS and Android, integrate a numeric PIN and a draw pattern as alternative authentication schemes to provide user-friendly login services. However, the password spaces of these schemes are significantly smaller than text-based passwords, rendering them less secure and easy to break with some knowledge of device owners [8].

All correspondences should be addressed to Dr. Gail-Joon Ahn at gahn@asu.edu.

To bring a fast and fluid login experience on touch-screen devices, the Windows 8™ operating system comes with a picture password authentication system, namely picture gesture authentication (PGA) [25], which is also an instance of background draw-a-secret (BDAS) schemes [18]. This new authentication mechanism hit the market with miscellaneous computing devices including personal computers and tablets. At the time of writing, over 60 million Windows 8™ licenses have been sold [21] and it is estimated that 400 million computers and tablets will run Windows 8™ with this newly introduced authentication scheme in one year [28]. Consequently, it is imperative to examine and explore potential attacks on picture gesture authentication in such a prevalent operating system for further understanding user experiences and enhancing this commercially popular picture password system.

Many graphical password schemes—including DAS [24], Face [9], Story [15], PassPoints [41] and BDAS [18]—have been proposed in the past decade (for more, please refer to [6, 7, 13, 14, 16, 23, 34, 37]). Amongst these schemes, click-based schemes, such as PassPoints, have attracted considerable attention and some research has analyzed the patterns and predictable characteristics shown in their passwords [12, 39]. Furthermore, harvesting characteristics from passwords of a target picture and exploiting hot-spots and geometric patterns on the target picture have been proven effective for attacking click-based schemes [17, 32, 38]. However, PGA allows complex gestures other than a simple click. Moreover, a new feature in PGA, autonomous picture selection by users, makes it unrealistic to harvest passwords from the target pictures for learning. In other words, the target picture is previously *unseen* to any attack models. All existing attack approaches lack a generic knowledge representation of user choice in password selection that should be abstracted from specific pictures. The absence of this abstraction makes existing attack approaches impossible or abysmal (if

possible) to work on previously unseen target pictures.

In this paper, we provide an empirical analysis of user choice in PGA based on real-world usage data, showing interesting findings on user choice in selecting background picture, gesture location, gesture order, and gesture type. In addition, we propose a new attack framework that represents and learns users' password selection patterns from training datasets and generates ranked password dictionaries for previously unseen target pictures. To achieve this, it is imperative to build generic knowledge of user choice from the abstraction of hot-spots in pictures. The core of our framework is the concept of a selection function that simulates users' selection processes in choosing their picture passwords. Our approach is not coupled with any specific pictures. Hence, the generation of a ranked password list is then transformed into the generation of a ranked selection function list which is then executed on the target pictures. We present two algorithms for generating the selection function list: one algorithm is to appropriately develop an optimal guessing strategy for a large-scale training dataset and the other deals with the construction of high-quality dictionaries even when the size of the training dataset is small. We also discuss the implementation of our attack framework over PGA, and evaluate the efficacy of our proposed approach with the collected datasets.

The contributions of this paper are summarized as follows:

- We compile two datasets of PGA usage from user studies² and perform an empirical analysis on collected data to understand user choice in background picture, gesture location, gesture order, and gesture type;
- We introduce the concept of a selection function that abstracts and models users' selection processes when selecting their picture passwords. We demonstrate how selection functions can be automatically identified from training datasets; and
- We propose and implement a novel attack framework which could be potentially redesigned as a picture-password-strength meter for PGA. Our evaluation results show that our approach cracked 48.8% passwords for previously unseen pictures in one of our datasets and 24.0% in the other within fewer than 2^{19} guesses (the entire password space is $2^{30.1}$).

The rest of this paper is organized as follows. Section 2 gives an overview of picture gesture authentication. Section 3 discusses our empirical analysis on picture gesture authentication. In Section 4, we illustrate

²These datasets with the detailed information will be available at <http://sefcom.asu.edu/pga/>.

our attack framework. Section 5 presents the implementation details and evaluation results of the proposed attack framework. We discuss several research issues in Section 6 followed by the related work in Section 7. Section 8 concludes the paper.

2 Picture Gesture Authentication: An Overview

Like other login systems, Windows 8™ PGA has two independent phases, namely registration and authentication. In the registration stage, a user chooses a picture from his or her local storage as the background. PGA does not force users to choose pictures from a predefined repository. Even though users may choose pictures from common folders, such as the Picture Library folder in Windows 8™, the probability for different users to choose an identical picture as the background for their passwords is low. This phenomenon requires potential attack approaches to have the ability to perform attacks on previously unseen pictures. PGA then asks the user to draw exactly three gestures on the picture with his or her finger, mouse, stylus, or other input devices depending on the equipment he or she is using. A gesture could be viewed as the cursor movements between a pair of 'finger-down' and 'finger-up' events. PGA does not allow free-style gestures, but only accepts tap (indicating a location), line (connecting areas or highlighting paths), and circle (enclosing areas) [29]. If the user draws a free-style gesture, PGA will convert it to one of the three recognized gestures. For instance, a curve would be converted to a line and a triangle or oval will be stored as a circle. To record these gestures, PGA divides the longest dimension of the background image into 100 segments and the short dimension on the same scale to create a grid, then stores the coordinates of the gestures. The line and circle gestures are also associated with additional information such as directions of the finger movements.

Once a picture password is successfully registered, the user may login the system by drawing corresponding gestures instead of typing his or her text-based password. In other words, PGA first brings the background image on the screen that the user chose in the registration stage. Then, the user should reproduce the drawings he or she set up as his or her password. PGA compares the input gestures with the previously stored ones from the registration stage. The comparison is not strictly rigid but shows tolerance to some extent. If any of gesture type, ordering, or directionality is wrong, the authentication fails. When they are all correct, an operation is further taken to measure the distance between the input password and the stored one. For tapping, the gesture passes authentication if the predicate $12 - d^2 \geq 0$ satisfies, where d denotes the distance between the tap coordi-

nates and the stored coordinates. The starting and ending points of line gestures and the center of circle gestures are measured with the same predicate [29].

The differences between PGA and the first BDAS scheme proposed in [18] include: i) in PGA, a user uploads his or her picture as the background instead of choosing one from a predefined picture repository; ii) a user is only allowed to draw three specific types of gestures in PGA, while BDAS takes any form of strokes. The first difference makes PGA more secure than the previous scheme, because a password dictionary could only be generated after the background picture is acquired. However, the second characteristic reduces the theoretical password space from its counterpart. Pace et al. [29] quantified the size of the theoretical password space of PGA which is $2^{30.1}$ with current length-three configuration in Windows 8TM. For more details, please refer to [29].

3 An Empirical Analysis of Picture Gesture Authentication

In this section, we present an empirical analysis on user choice in PGA by analyzing data collected from our user studies. Our empirical study is based on human cognitive capabilities. Since human cognition of pictures is limited in a similar way to their cognition of texts, the picture passwords selected by users are probably constrained by human cognitive limits which would be similar to the ones in text-based passwords [42].

3.1 Experiment Design

For the empirical study, we developed a web-based PGA system for conducting user studies. The developed system resembles Windows 8TM PGA in terms of its workflow and appearance. The differences between our implementation and Windows 8TM PGA include: i) our system works with major browsers in desktop PCs and tablets whereas Windows 8TM PGA is a stand-alone program; ii) some information, such as the criterion for circle radius comparison, is not disclosed. In other words, our implementation and Windows 8TM PGA differ in some criteria (we regard radii the same if their difference is smaller than 6 segments in grid). In addition, our developed system has a tutorial page that includes a video clip educating how to use the system and a test page on which users can practice gesture drawings.

Our study protocol, including the type of data we plan to collect and the questionnaire we plan to use, was reviewed by our institution's IRB. The questionnaire consisted of four sections: i) general information of the subject (gender, age, level of education received, and race); ii) general feeling toward PGA (is it easier to remember, faster to input, harder to guess, and easier to observe

than text-based password); iii) selection of background picture (preferred picture type); and iv) selection of password (preferred gesture location and type).

We started user studies after receiving the IRB approval letter in August 2012 and compiled two datasets from August 2012 to January 2013 using this system. *Dataset-1* was acquired from a testbed of picture password used by an undergraduate computer science class. *Dataset-2* was produced by advertising our studies in schools of engineering and business in two universities and Amazon's Mechanical Turk crowdsourcing service that has been used in security-related research work [26]. Turkers who had finished more than 50 tasks and had an approval rate greater than 60% were qualified for our user study.

For registration, subjects in *Dataset-1* were asked to provide their student IDs for a simple verification after which they were guided to upload a picture, register a password and then use the password to access class materials including slides, homework, assignments, and projects. Subjects used this system for the Fall 2012 semester which lasted three and a half months at our university. If subjects forgot their passwords during the semester, they would inform the teaching assistant who reset their passwords. Subjects were allowed to change their passwords by clicking a change password link after login. There were 56 subjects involved in *Dataset-1* resulting in 58 unique pictures, 86 registered passwords, and 2,536 login attempts.

Instead of asking subjects to upload pictures for *Dataset-2*, we chose 15 pictures (please refer to Appendix B for the pictures) in advance from the PASCAL Visual Object Classes Challenge 2007 dataset [19]. We chose these pictures because they represent a diverse range of pictures in terms of category (portrait, wedding, party, bicycle, train, airplane and car) and complexity (pictures with few and plentiful stand-out regions). Subjects were asked to choose one password for each picture by pretending that it was protecting their bank information. The 15 pictures were presented to subjects in a random order to reduce the dependency of password selection upon the picture presentation order. 762 subjects participated in the *Dataset-2* collection resulting in 10,039 passwords. The number of passwords for each picture in the *Dataset-2* varies slightly, with an average of 669, because some subjects quit the study without setting up passwords for all pictures.

For both datasets, subjects were asked to finish the aforementioned questionnaire to help us understand their experiences. We collected 685 (33 for *Dataset-1*, 652 for *Dataset-2*) copies of survey answers in total. According to the demographic-related inquiries in the exit survey, 81.8% subjects in *Dataset-1* are self-reported male and 63.6% are between 18 and 24 years old. While partic-

Table 1: Survey Question: Which of the following best describes what you are considering when you choose locations to perform gestures?

Multi-choice Answers	Dataset		
	1	2	Overall
I try to find locations where special objects are.	24 (72.7%)	389 (59.6%)	413 (60.3%)
I try to find locations where some special shapes are.	8 (24.2%)	143 (21.9%)	151 (22.1%)
I try to find locations where colors are different from their surroundings.	0 (0%)	57 (8.7%)	57 (8.3%)
I randomly choose a location to draw without thinking about the background picture.	1 (3.0%)	66 (10.1%)	67 (9.8%)

ipants in *Dataset-2* are more diverse with 64.4% male, 37.2% among 18 to 24 years old, 45.4% among 25 - 34, and 15.0% among 35 - 50. Even though the subjects in our studies do not represent all possible demographics, the data collected from them represents the most comprehensive PGA usage so far. Their tendencies could provide us with significant insights into the user choice in PGA.

3.2 Results

This section summarizes our empirical analysis on the above-mentioned datasets by presenting five findings.

3.2.1 Finding 1: Relationship Between Background Picture and User’s Identity, Personality, or Interests

We analyzed all unique pictures³ in *Dataset-1*, and the background pictures chosen by subjects range from celebrity to system screenshot. We categorize them into six classes: i) people (27/58), ii) civilization (7/58), iii) landscape (3/58), iv) computer-generated picture (14/58), v) animals (6/58), and vi) others (1/58).

For the category of ‘people’, 6 pictures were categorized as ‘me’; 12 pictures were subjects’ families; 4 were pictures of subjects’ friends; and 5 were celebrities. The analysis of answers to the survey question “*Could you explain why you choose such types of pictures?*” revealed two opposite attitudes towards using picture of people. The advocates for such pictures considered: i) it is more friendly. e.g. “*The image was special to me so I enjoy seeing it when I log in*”; ii) it is easier for remembering passwords. e.g. “*Marking points on a person is easier to remember*”; and iii) it makes password more secure. e.g. “*The picture is personal so it should be much harder for someone to guess the password*”. However, other participants believed it may leak his or her identity or privacy. e.g. “*revealing myself or my family to anyone who picks up the device*”. They preferred other types of pictures

³Due to the confidentiality agreement with the subjects, we are not able to share pictures that are marked having personally identifiable information.

Table 2: Attributes of Most Frequently Used PoIs

Attributes	# Gesture	# Password	# Subject
Eye	36	20	19
Nose	21	13	10
Hand/Finger	6	5	4
Jaw	5	3	3
Face (Head)	4	2	2

because “*less personal if someone gets my picture*” and “*landscape usually doesn’t have any information about who you are*”.

14 pictures in *Dataset-1* could be categorized as computer-generated pictures including computer game posters, cartoons, and some geometrical graphs. 24.1% (14/58) of such pictures were observed in *Dataset-1* but the survey results indicated 6.4% (42/652) of participants were in such a usage pattern in *Dataset-2* based on the following survey question: “*Please indicate the type of pictures you prefer to use as the background*”. We concluded the population characteristics (male, age 18-24, college students) in *Dataset-1* were the major reason behind this phenomenon. The answers to “*Could you explain why you choose such types of pictures?*” in *Dataset-1* supported this conjecture: “*computer game is something I am interested [in] it*” and “*computer games picture is personalized to my interests and enjoyable to look at*”.

It is obvious that pictures with personally identifiable information may leak personal information. However, it is less obvious that even pictures with no personally identifiable information may provide some clues which may reveal the identity or persona of a device owner. Traditional text-based password does not have this concern as long as the password is kept secure. Previous graphical password schemes, such as Face and PassPoints, do not have this concern either because pictures are selected from a predefined repository.

3.2.2 Finding 2: Gestures on Points of Interest

The security of background draw-a-secret schemes mostly relies on the location distribution of users’ gestures. It is the most secure if the locations of users’ gestures follow a uniform distribution on any picture. However, such passwords would be difficult to remember and may not be preferable by users. By analyzing the collected passwords, we notice that subjects frequently chose standout regions (points of interest, PoIs) on which to draw. As shown in Table 1, only 9.8% subjects claimed to choose locations randomly without caring about the background picture. The observation is supported by survey answers to “*Could you explain the way you choose locations to perform gestures?*”: “*If I have to remember it; it [would] better stand out.*” and “*Something that would make it easier to remember*”.

Even though the theoretical password space of PGA is

Table 3: Numbers of Gesture Type Combinations and Average Time Spent on Creating Them

		$3 \times t$	$3 \times l$	$3 \times c$	$2 \times t+l$	$2 \times t+c$	$2 \times l+t$	$2 \times l+c$	$2 \times c+t$	$2 \times c+l$	$t+l+c$
<i>Dataset-1</i>	#	60	3	0	9	1	7	1	0	0	5
	Average Time (Seconds)	5.74	12.39	N/A	10.12	21.56	11.17	17.51	N/A	N/A	11.22
<i>Dataset-2</i>	#	3438	1447	253	1211	380	1000	622	192	442	1054
	Average Time (Seconds)	4.33	7.11	9.96	6.02	6.14	7.72	9.98	8.78	10.19	9.37

Table 4: Numbers of Gesture-order Patterns

	H+	H-	V+	V-	DIAG	Others
<i>Dataset-1</i>	43	5	16	4	22	18
	50.0%	5.8%	18.6%	4.6%	25.5%	20.9%
<i>Dataset-2</i>	3144	1303	1479	887	2621	3326
	31.3%	12.9%	14.7%	8.8%	26.1%	33.1%

larger than text-based passwords with the same length, a background picture affects user choice in gesture location, reducing the feasible password space tremendously. We summarize three popular ways that subjects used to identify standout regions: i) finding regions with objects. e.g. “*I chose eyes and other notable features*” and “*I chose locations such as nose, mouth or whole face*”; ii) finding regions with remarkable shapes. e.g. “*if there is a circle there I would draw a circle around that*”; and iii) finding regions with outstanding colors. The detailed distribution of these selection processes is shown in Table 1. 60.3% of subjects prefer to find locations where special objects catch their eyes while 22.1% of subjects would rather draw on some special shapes.

3.2.3 Finding 3: Similarities Across Points of Interest

We analyzed the attributes of PoIs that users preferred to draw on. We paid more attention to the pictures of people because it was the most popular category. In the 31 registered passwords for the 27 pictures of people uploaded by 22 subjects in *Dataset-1*, we analyzed the patterns of PoI choice. As shown in Table 2, 36 gestures were drawn on eyes and 21 gestures were drawn on noses. Other locations that attracted subjects to draw included hand/finger, jaw, face (head), and ear. Interestingly, 19 subjects out of 22 (86.3%) drew on eyes at least once, while 10 subjects (45.4%) performed gestures on noses. The tendencies to choose similar PoIs by different subjects are common in other picture categories as well. Figure 1 shows another example where two subjects uploaded two versions of *Starry Night* in *Dataset-1*. The passwords they chose show strikingly similar patterns with three taps on stars, even if there is no single gesture location overlap.

3.2.4 Finding 4: Directional Patterns in PGA Password

Salehi-Abhari et al. [32] suggest many passwords in click-based systems follow some directional patterns. We are interested in whether PGA passwords show similar characteristics. For simplicity, we consider the coordinates of tap and circle gestures as their locations and the middle



Figure 1: Two Versions of *Starry Night* and Corresponding Passwords

point of the starting and ending points of line as its location. If the x or y coordinate of a gesture sequence follows a consistent direction regardless of the other coordinate, we say the sequence follows a LINE pattern. We divide LINE patterns into four categories: i) H+, denoting left-to-right ($x_i \leq x_{i+1}$); ii) H-, denoting right-to-left ($x_i \geq x_{i+1}$); iii) V+, denoting top-to-bottom ($y_i \leq y_{i+1}$); and iv) V-, denoting bottom-to-top ($y_i \geq y_{i+1}$). If a sequence of gestures follows a horizontal pattern and a vertical pattern at the same time, we say it follows a DIAG pattern.

We examined the occurrence of each LINE and DIAG pattern in the collected data. As shown in Table 4, more than half passwords in both datasets exhibited some LINE patterns, and a quarter of them exhibited some DIAG patterns. Among four LINE patterns, H+ (drawing from left to right) was the most popular one with 50.0% and 31.3% occurrences in *Dataset-1* and *Dataset-2*, respectively. And, V+ (drawing from top to bottom) was the second most popular with 18.6% and 14.7% occurrences in two datasets, respectively. This finding shows it is reasonable to use gesture-order patterns as one heuristic factor to prioritize generated passwords.

3.2.5 Finding 5: Time Disparity among Different Combinations of Gesture Types

We analyzed all registered passwords to understand the gesture patterns and the relationship between gesture type and input time. For 86 registered passwords (258 gestures) in *Dataset-1*, 212 (82.1%) gesture types were taps, 39 (15.1%) were lines, and only 7 (2.7%) were circles. However, the corresponding occurrences for 10,039 registered passwords (30,117 gestures) in *Dataset-2* were 15,742 (52.2%), 10,292 (34.2%), and 4,083 (13.5%), respectively. Obviously, subjects in *Dataset-2* chose more diverse gesture types than subjects in *Dataset-1*. As shown in Table 3, there was a strong connection between the time subjects spent on reproducing passwords and

the gesture types they chose. Three taps, the most common gesture combination, appeared in both datasets with the lowest average time (5.74 seconds and 4.33 seconds in corresponding dataset). On the other hand, the passwords with two circles and one line took the longest average input time (10.19 seconds in *Dataset-2*). In the user studies, subjects in *Dataset-2* were asked to set up the passwords by pretending they were protecting their bank information. However, subjects in *Dataset-1* actually used these passwords to access the class materials which they accessed more than four times a week on average. This may be a reason why subjects in *Dataset-1* prefer passwords with simpler gesture type combinations that are easier to reproduce in a timely manner.

4 Attack Framework

In this section, we present an attack framework on Windows 8™ picture gesture authentication, leveraging the findings addressed in Section 3. Our attack framework takes the target picture’s PoIs, a set of learning pictures’ PoIs and corresponding password pairs as input, and produces a list of possible passwords, which is ranked in the descending order of the password probabilities.

Next, we first discuss the attack models followed by the representations of picture password and PoI. We then illustrate the idea of a selection function and its automatic identification. We also present two algorithms for generating a selection function sequence list and describe how it can generate picture password dictionaries for previously unseen target pictures.

4.1 Attack Models

Depending on the resources an attacker possesses, we articulate three different attack models: i) *Pure Brute-force Attack*: an attacker blindly guesses the picture password without knowing any information of the background picture and the users’ tendencies. The password space in this model is $2^{30.1}$ in PGA [29]. ii) *PoI-assisted Brute-force Attack*: an attacker assumes the user only performs drawings on PoIs of the background picture and this model randomly guesses passwords on identified PoIs. The password space for a picture with 20 PoIs in this model is $2^{27.7}$ [29]. Salehi-Abari et al. [32] designed an approach to automatically identify hot-spots in a picture and generate passwords on them. iii) *Knowledge-based PoI-assisted Attack*: in addition to the assumption for PoI-assisted brute-force attack, an attacker ought to have some knowledge about the password patterns learned from collected picture and password pairs (not necessarily from the target user or picture). The guessing space in this model is the same as the one in PoI-assisted brute-force attack. However, the generated dictionaries in this model are ranked with the higher possibility passwords

on the top of the list.

Attack schemes could also be divided into two categories based on whether or not an attacker has the ability to attack previously unseen pictures. The method presented in [32] is able to attack previously unseen pictures for click-based graphical password. It uses click-order heuristics to generate partially ranked dictionaries. However, this approach cannot be applied directly to background draw-a-secret schemes because the gestures allowed in such schemes are much more complex and the order-based heuristics could not capture users’ selection processes accurately. In contrast, our attack framework could abstract generic knowledge of user choice in picture password schemes. In addition, as a working *knowledge-based PoI-assisted* model, it is able to generate ranked dictionaries for previously unseen pictures.

4.2 Password and PoI Representations

We first formalize the representation of a password in PGA with the definition of a location-dependent gesture which represents a single gesture on some locations in a picture.

Definition 1 A location-dependent gesture (LdG) denoted as π is a 7-tuple $\langle g, x_1, y_1, x_2, y_2, r, d \rangle$ that consists of gesture’s type, location, and other attributes.

In this definition, g denotes the type of LdG that must be one of tap, line, and circle. A tap LdG is further represented by the coordinates of a gesture $\langle x_1, y_1 \rangle$. A line LdG is denoted by the coordinates of the starting and ending points of a gesture $\langle x_1, y_1 \rangle$ and $\langle x_2, y_2 \rangle$. A circle LdG is denoted by the coordinates of its center $\langle x_1, y_1 \rangle$, radius r , and direction $d \in \{+, -\}$ (clockwise or not). We define the password space of location-dependent gesture as $\Pi = \Pi_{\text{tap}} \cup \Pi_{\text{line}} \cup \Pi_{\text{circle}}$. A valid PGA password is a length-three sequence of LdGs denoted as $\vec{\pi}$, and the PGA password space could be denoted as $\vec{\Pi}$.

A point of interest is a standout region in a picture. PoIs could be regions with semantic-rich meanings, such as face (head), eye, car, clock, etc. Also, they could stand out in terms of their shapes (line, rectangle, circle, etc.) or colors (red, green, blue, etc.). We denote a PoI by the coordinates of its circumscribed rectangle and some describing attributes. A PoI is a 5-tuple $\langle x_1, y_1, x_2, y_2, D \rangle$, where $\langle x_1, y_1 \rangle$ and $\langle x_2, y_2 \rangle$ are the coordinates of the top-left and bottom-right points of the circumscribed rectangle, and $D \subseteq 2^{\mathcal{D}}$ is a set of attributes that describe this PoI. \mathcal{D} has three sub-categories \mathcal{D}_o , \mathcal{D}_s and \mathcal{D}_c and four wildcards $*_o, *_s, *_c$, and $*$, where $\mathcal{D}_o = \{\text{head, eye, nose, ...}\}$, $\mathcal{D}_s = \{\text{line, rectangle, circle, ...}\}$, and $\mathcal{D}_c = \{\text{red, blue, yellow, ...}\}$. Wildcards are used when no specific information is available. For example, if a PoI is identified with objectness measure [3] that gives

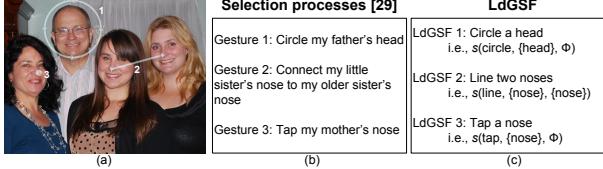


Figure 2: (a) Background picture and password (b) User's selection processes that were taken from [30] (c) Corresponding LdGSFs that simulate user's selection processes

no semantics about the identified region, we mark the PoI's describing attribute as *.

4.3 Location-dependent Gesture Selection Functions

A key concept in our framework is the location-dependent gesture selection function (LdGSF) which models and simulates the ways of thinking that users go through when they select a gesture on a picture. The motivation behind this abstraction is that the set of PoIs and their locations differ from picture to picture, but the ways that users think to choose locations for drawing a gesture exhibit certain patterns. This conjecture is supported by our observations from collected data and surveys discussed in Section 3. With the help of LdGSF, the PoIs and corresponding passwords in training pictures are used to generalize picture-independent knowledge that describes how users choose passwords.

Definition 2 A *location-dependent gesture selection function (LdGSF)* is a mapping $s : G \times 2^{\mathcal{D}} \times 2^{\mathcal{D}} \times \Theta \rightarrow 2^{\Pi}$ which takes a gesture, two sets of PoI attributes, and a set of PoIs in the learning picture as input to produce a set of location-dependent gestures.

The universal set of LdGSF is defined as S . A length-three sequence of LdGSF is denoted as \vec{s} , and a set of length-three LdGSF sequences is denoted as \vec{S} . $s(\text{tap}, \{\text{red}, \text{apple}\}, \emptyset, \theta_k)$ is interpreted as ‘tap a red apple in the picture p_k ’ and $s(\text{circle}, \{\text{head}\}, \emptyset, \theta_k)$ as ‘circle a head in p_k ’. Note that, no specific information of the locations of ‘red apple’ and ‘head’ is provided here which makes the representations independent from actual locations of objects in the picture.

One challenge we face is some PoIs may be big enough to take several unique gestures. Let us consider a picture with a big car image in it. Simply saying ‘tap a car’ could result in lots of distinct tap gestures in the circumscribed rectangle of the car. One solution to this problem is to divide the circumscribed rectangle into a grid with the scale of toleration threshold. However, this solution would result in too many password entries in the generated dictionary. For simplicity, we introduce five inner points for one PoI, namely center, top, bottom, left, and right that denote the center of the PoI and

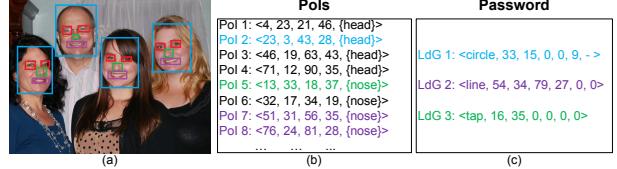


Figure 3: (a) Background picture and identified PoIs (b) Identified PoIs (c) Password representations (Colors are used to indicate the connections between the PoIs in (b) and LdGs in (c))

four points of the center of two consecutive corners. Any gesture that falls into the proximities of these five points of a PoI would be considered as an action on this PoI. For some PoIs that are big enough to take an inner line gesture, we put \emptyset as the input of the second set of PoI attributes. $s(\text{line}, \{\text{mouth}\}, \emptyset, \theta_k)$ denotes ‘line from the left(right) to the right(left) on the same mouth’. While, $s(\text{line}, \{\text{mouth}\}, \{\text{mouth}\}, \theta_k)$ means ‘connect two different mouths’.

Figure 2 shows an example demonstrating how LdGSF simulates a user's selection processes that were taken from [30]. In reality, a user's selection process on a PoI and gesture selection may be determined by some subjective knowledge and cognition. For example, ‘circle my father's head’ and ‘tap my mother's nose’ may involve some undecidable computing problems. One solution to handle this issue is to approximate subjective selection processes in objective ways by including some modifiers. ‘circle my father's head’ may be transformed into ‘circle the *uppermost* head’ or ‘circle the *biggest* head’. However, it is extremely difficult, if not impossible, to accurately approximate subjective selection processes in this way, and it may bring serious over-fitting problems in the learning stage. Instead, we choose to ignore subjective information by abstracting ‘circle my father's head’ to ‘circle a head’. A drawback of this abstraction is that an LdGSF may return more than one LdG and we have no knowledge to rank them directly, as they come from the same LdGSF. Using Figure 2(a) as an example, ‘circle a head’ outputs four different LdGs on each head in the picture. The LdGSF sequence shown in Figure 2(c) generates $4 \times (4 \times 3) \times 4 = 192$ passwords. To cope with this issue, we use gesture-order to rank the passwords generated by the same LdGSF sequence that will be detailed in Section 4.5. Next, we present an automated approach to extract users' selection processes from the collected data and represent them with LdGSFs.

Figure 3 shows an example demonstrating that how to extract users' selection processes from PoIs automatically. First, PoIs in the background picture are identified using mature computer vision techniques such as object detection, feature detection and objectness measure. Then, each LdG in a password is compared with

PoIs based on their coordinates and sizes. If a match between PoIs and LdGs is found, a new LdGSF is created as the combination of the LdG’s gesture type and PoI’s attributes. For instance, the location and size of LdG 1 in Figure 3(c) matches PoI 2 in Figure 3(b) (the locations of the circle gesture and PoI center are compared first; then, the radius of the circle is compared with 1/2 of PoI’s height and width). Then, an LdGSF $s(\text{circle}, \{\text{head}\}, \emptyset)$ is created which is equivalent to the LdG shown in Figure 2(c).

To choose a password in PGA, the user *selects* a length-three LdGSF sequence. With the definition of LdGSF, the generation of ranked password list is simplified into the generation of the ranked LdGSF sequence list. Let $\text{order}: \vec{S} \rightarrow \{1..|\vec{S}|\}$ be a bijection which indicates the order LdGSF sequences should be performed. The objective of generating ranked LdGSF sequence list is to find such a bijection.

4.4 LdGSF Sequence List Generation and Ordering

Now we present our approach to find the aforementioned bijection that indicates the order that the LdGSF sequences should be performed on a target picture for generating the password dictionary. Our framework is not dependent on certain rules, but is adaptive to the tendencies shown by users who participate in the training set. The characteristic of adaptiveness helps our framework generate dedicated guessing paths for different training data. Next, we present two algorithms for obtaining such a feature.

4.4.1 BestCover LdGSF Sequence List Generation

We first propose an LdGSF sequence list generation algorithm named BestCover that is derived from $\mathcal{B}_{\text{emts}}$ [44]. The objective of BestCover LdGSF sequence list generation is to optimize the guessing order for the sequences in the list by minimizing the expected number of sequences that need to be tested on a random choice of picture in the training dataset.

The problem is formalized as follows: **Instance:** The collection of LdGSF sequences $\vec{s}_1, \dots, \vec{s}_n$ and corresponding picture password $\vec{\pi}_1, \dots, \vec{\pi}_n$, for which $\vec{s}_i(\theta_i) \ni \vec{\pi}_i, i \in \{1..n\}$ and $\theta_1, \dots, \theta_n$ are the sets of PoIs in pictures p_1, \dots, p_n . **Question:** Expected Min Selection Search (emss): The objective is to find order so as to minimize $\mathbb{E}(\min\{i : \vec{s}_i(\theta_r) \ni \vec{\pi}_r\})$, where $\vec{s}_i = \text{order}^{-1}(i)$ and the expectation is taken with respect to a random choice of $r \leftarrow \{1..n\}$.

The hardness of this problem is that different LdGSFs and LdGSF sequences may generate the same list of LdGs and passwords. For instance, ‘tap a red object’ and ‘tap an apple’ turn out the same result on a picture

in which there is a red apple. An overlap in different LdGSF results is similar to the coverage characteristics in the set cover problem. We can prove the NP-hardness of emss by reducing from emts [44]. Due to space limitations, we omit the corresponding proof. We give an approximation algorithm for emss in Algorithm 1 that is a modification from $\mathcal{B}_{\text{mssc}}$ [20]. The time complexity of BestCover is $O(n^2 + |\vec{S}'| \log(|\vec{S}'|))$.

Algorithm 1: BestCover($(\vec{s}_1, \dots, \vec{s}_n), (\vec{\pi}_1, \dots, \vec{\pi}_n)$)

```

for  $i = 1..n$  do
     $T_{\vec{s}_i} \leftarrow \{k : \vec{s}_i(\theta_k) \ni \vec{\pi}_k\};$ 
end
 $\vec{S}' \leftarrow \{\vec{s} : |T_{\vec{s}}| > 0\};$ 
for  $i = 1..|\vec{S}'|$  do
     $\text{order}^{-1}(i) \leftarrow \vec{s}_k$ , that  $T_{\vec{s}_k}$  has most elements that are not
    included in  $\bigcup_{i' < i} \text{order}^{-1}(i')$ ;
end
return  $\text{order}$ 

```

BestCover is good for a training dataset that consists of comprehensive and large scale password samples, because it assumes the target passwords exhibit same or at least very similar distributions to the training data. However, if the training dataset is small and biased, the results from BestCover may over-fit the training data and fail in testing data.

4.4.2 Unbiased LdGSF Sequence List Generation

The over-fitting problem in BestCover is brought about by the biased PoI attribute distributions in training data. For example, we have a training set with 9 pictures of apples and 1 picture of a car, and 5 corresponding passwords have circles on apples and 1 has a circle on car. In the generated LdGSF sequence list, BestCover will put sequences with ‘circle an apple’ prior to the ones with ‘circle a car’, because the former ones have an LdGSF that was used in more passwords. However, we can see the probability for users to circle car (1/1) is higher than apples (5/9) if we consider the occurrences of apple and car in pictures.

Unbiased LdGSF sequence list generation copes with this issue by considering the PoI attribute distributions. It removes the biases from the training dataset by normalizing the occurrences of LdGSFs with the occurrences of their corresponding PoIs. Let $D_{\vec{s}_k} \subseteq \theta$ denote the event that θ contains enough PoIs that have attributes specified in \vec{s}_k . If a PoI with a specific type of attributes does not exist in a picture, the probability that a user select the PoI with such an attribute on this picture to draw a password is 0, denoted as $\Pr(\vec{s}_k | D_{\vec{s}_k} \subseteq \theta) = 0$, e.g. a user would not think and perform ‘tap a red apple’ on a picture without the existence of the red apple. We assume each LdGSF in a sequence is independent of each other and approximately compute $\Pr(\vec{s}_k | D_{\vec{s}_k} \subseteq \theta)$ with Equation 1.

$$\begin{aligned}
& Pr(\vec{s}_k | D_{\vec{s}_k} \subseteq \theta) \\
&= Pr(s_1 s_2 s_3 | D_{s_1} \subseteq \theta \wedge D_{s_2} \subseteq \theta \wedge D_{s_3} \subseteq \theta) \\
&= Pr(s_1 | D_{s_1} \subseteq \theta) \times Pr(s_2 | D_{s_2} \subseteq \theta) \times Pr(s_3 | D_{s_3} \subseteq \theta)
\end{aligned} \tag{1}$$

For each $s_i \in S$, we compute $Pr(s_i | D_{s_i} \subseteq \theta)$ with Equation 2:

$$Pr(s_i | D_{s_i} \subseteq \theta) = \frac{\sum_{j=1}^n count(D_{s_i}, \vec{\pi}_j)}{\sum_{j=1}^n count(D_{s_i}, \theta_j)} \tag{2}$$

where $\sum_{j=1}^n count(D_{s_i}, \vec{\pi}_j)$ denotes the number of LdGs in passwords of the training set that share the same attributes with s_i , and $\sum_{j=1}^n count(D_{s_i}, \theta_j)$ denotes the number of PoIs in the training set that share the same attributes with s_i . $Pr(s_i | D_{s_i} \subseteq \theta)$ describes the probability of using a certain LdGSF when there are enough PoIs with the required attributes.

The Unbiased algorithm generates an LdGSF sequence list by ranking $Pr(\vec{s}_k | D_{\vec{s}_k} \subseteq \theta)$ instead of $Pr(\vec{s}_k)$ in descending order as shown in Algorithm 2. The time complexity of Unbiased is $O(n|S| + |S| \log(|S|))$. The Unbiased algorithm would be better for the scenarios where fewer samples are available or samples are highly biased.

Algorithm 2: Unbiased(S)

```

for  $s \in S$  do
    | Compute  $Pr(s | D_s \subseteq \theta)$  with Equation 2;
end
for  $\vec{s} \in \vec{S}$  do
    | Compute  $Pr(\vec{s} | D_{\vec{s}} \subseteq \theta)$  with Equation 1;
end
for  $i = 1..|\vec{S}|$  do
    |  $order^{-1}(i) \leftarrow \vec{s}_k$ , that  $Pr(\vec{s}_k | D_{\vec{s}_k} \subseteq \theta)$  holds the  $i$ -th position
    | in the descending ordered  $Pr(\vec{s} | D_{\vec{s}} \subseteq \theta)$  list;
end
return  $order$ 

```

4.5 Password Dictionary Generation

The last step in our attack framework is to generate the password dictionary for a previously unseen target picture. First, the PoIs in the previously unseen picture are identified. Then, a dictionary is acquired by applying the LdGSF sequences on the PoIs, following the order created by the BestCover or Unbiased algorithm. Obviously, the passwords generated by an LdGSF sequence that holds a higher position in the LdGSF sequence list will also be in higher positions in the dictionary. However, as addressed earlier, BestCover and Unbiased algorithms do not provide extra information to rank the passwords generated by the same LdGSF sequence. Inspired by using the click-order patterns as the heuristics for dictionary generation [32], we propose to rank

such passwords generated by the same LdGSF sequence with gesture-orders. In the training stage, we record the gesture-order occurrence of each LINE and DIAG pattern and rank the patterns in descending order. In the attack stage, for the passwords generated by the same LdGSF sequence, we reorder them with their gesture-orders in the order of LINE and DIAG patterns. Passwords that do not belong to any LINE or DIAG pattern hold lower positions.

5 Implementation and Evaluation

5.1 PoI Identification

We chose OpenCV [1] as the computer vision framework for our implementation and collected several feature detection tools for automatically identifying PoIs in background pictures. The computer vision techniques we adopted include: i) object detection: the goal of object detection is to find the locations and sizes of semantic objects of a certain class in a digital image. Viola-Jones object detection framework [40] is the first computationally affordable online object detection framework that utilizes Haar-like features instead of image intensities. Each learned classifier is represented and stored as a haar cascade. We collected 30 proven haar cascades from [31] for 8 different object classes including face (head), eye, nose, mouth, ear, head, body, and clock. ii) low-level feature detection: due to the high positive and high negative rates of object detection, we also resorted to some low-level feature detection algorithms that identify standout regions without extracting semantics. To identify regions whose colors are different from their surroundings, we first converted the color pictures to black and white, then found the contours using algorithms in [35]. For the circle detection, we used Canny edge detector [10] and Hough transform algorithms [5]. iii) objectness measure: objectness measure [3] deals with class-generic object detection. Different from detecting objects in a specific class, the objectness measure finds the locations and sizes of class-generic objects whose colors and textures are opposed to the background images. Objectness measure could be considered as a technique combining several low-level feature detectors together. We used an objectness measure library from [2] that is able to locate objects and give numerical confidence values with its results.

Figure 4 displays the PoI detection results on four example pictures in *Dataset-2*. As we can see in Figure 4(b), circle detection could identify both bicycle wheels and car badge, but its false positive rate is a little high. Contour detection is the most robust algorithm with a low false positive rate which could locate regions whose colors are different as shown in Figure 4(c). Objectness measure shown in Figure 4(d) could also iden-

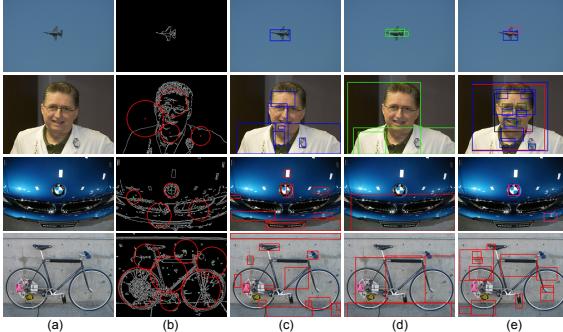


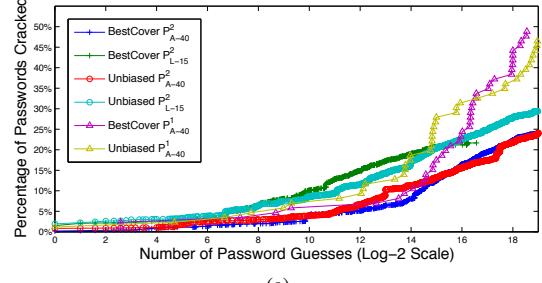
Figure 4: PoI Identification on Example Pictures in *Dataset-2*: (a) Original pictures (b) Circle detection with Hough transform (c) Contour detection (d) Objectness measure (e) Object detection

tify regions whose colors and textures are different from their surroundings. Since most haar cascades we used are designed for facial landmarks, they work smoothly on portraits as does the second picture in Figure 4(e). However, the results show relatively high false positive rates on pictures from other categories. In order to identify more PoIs as accurate as possible, our approach in PoI identification leveraged two steps. In the first step, all possible PoIs were identified using different kinds of tools. In the second step, we examined all identified PoIs and removed duplicates by comparing their locations, sizes and attributes. Then, our approach generated a PoI set called P_{A-40}^1 and P_{A-40}^2 for each picture in *Dataset-1* and *Dataset-2*, respectively. Those PoI sets consisted of at most 40 PoIs with the highest confidences.

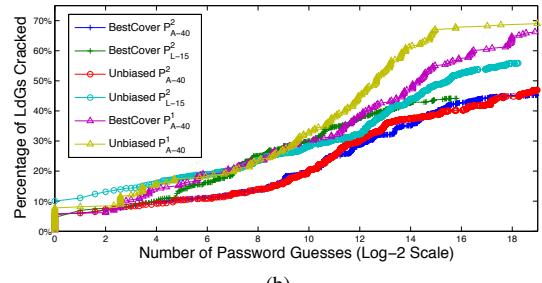
Since our attack algorithms are independent from the PoI identification algorithms, we are also interested in examining how our attack framework performs with ideal PoI annotations for pictures. Besides using the automated PoI identification techniques, we manually annotated pictures in *Dataset-2* for some outstanding PoIs as well. To annotate the pictures, we simply recorded the locations and attributes of at most fifteen most appealing regions in the pictures without referring to any password in the collected dataset. We call this annotated PoI set P_{L-15}^2 .

5.2 Attack Evaluation

Offline Attacks. Due to the introduction of a tolerance threshold, picture passwords may be more difficult to store securely compared with text-based passwords that are normally saved after salted hashing. Even though the approach that Windows 8TM is adopting to store picture passwords remains undisclosed, we could consider two attack scenarios where picture passwords are prone to offline attacks. In the first scenario, all passwords which fall into the vicinity (defined by the threshold) of chosen passwords could be stored in a file with salted hashes for comparison. An attacker who has access to this file



(a)



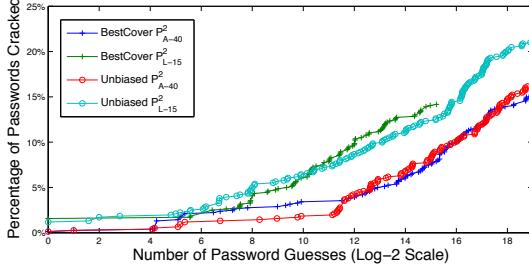
(b)

Figure 5: (a) Percentage of passwords cracked vs. number of password guesses, per condition. (b) Percentage of LdGs cracked vs. number of password guesses, per condition. For *Dataset-1*, there are 86 passwords that include 258 LdGs. For *Dataset-2*, there are 10,039 passwords that have 30,117 LdGs.

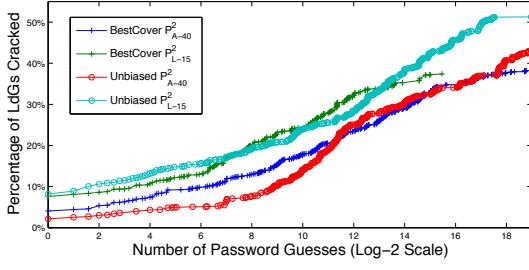
could perform offline dictionary attacks like cracking text-based password systems. In the second scenario, picture passwords could be used for other purposes besides logging into Windows 8TM, where no constraint on the number of attempts is enforced. For example, a registered picture password could be transformed and used as a key to encrypt a file. An attacker who acquires the encrypted file would like to perform an offline attack.

In order to attack passwords from a previously unseen picture, the training dataset excluded passwords from the target picture. More specifically, to evaluate *Dataset-1* (58 unique pictures), we used passwords from 57 pictures as the training data and attacked the passwords for the last picture. To evaluate *Dataset-2* (15 unique pictures), we used passwords for 14 pictures as training data, learned the patterns exhibited in the training data, and generated a password dictionary for the last picture. The same process was carried out 58 and 15 times for *Dataset-1* and *Dataset-2*, respectively, in which the target picture was different in each round. The size of the dictionary was set as 2^{19} which is 11-bit smaller than the theoretical password space. We compared all collected passwords for the target picture with the generated dictionary for the picture, and recorded the number of password guesses.

The offline attack results within 2^{19} guesses in different settings are shown in Figure 5. There are 86 passwords in *Dataset-1*, which have a total of 258 LdGs.



(a)

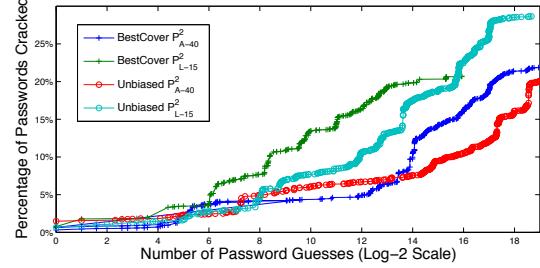


(b)

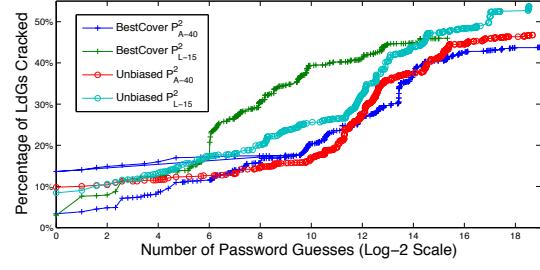
Figure 6: (a) Percentage of passwords cracked vs. number of password guesses, per condition. (b) Percentage of LdGs cracked vs. number of password guesses, per condition. Only the first chosen password by each subject in *Dataset-2* was considered. There are 762 passwords that have 2,286 LdGs.

And 10,039 passwords were collected in *Dataset-2*, containing a total of 30,117 LdGs. For *Dataset-1*, BestCover cracks 42 (48.8%) passwords out of 86 while Unbiased cracks 40 (46.5%) passwords for the same dataset with P_{A-40}^1 . For *Dataset-1*, 178 LdGs (68.9%) out of 258 are cracked with Unbiased and 171 (66.2%) are broken with BestCover. On the other hand, Unbiased with P_{L-15}^2 breaks 2,953 passwords (29.4%) out of 10,039 for *Dataset-2*. This implies Unbiased with P_{A-40}^2 cracking 2,418 passwords (24.0%) is the best result for all purely automated attacks on *Dataset-2*. As Figure 5 suggests, BestCover outperforms Unbiased slightly when ample training data is available. The better performance of both algorithms on *Dataset-1* is because the password gesture combinations in *Dataset-1* are relatively simpler than the ones in *Dataset-2* as we discussed in Section 3.2.5.

In *Dataset-2*, subjects may not choose all 15 passwords with the same care as they were eager to finish the process. To reduce this effect, we ran another analysis in which only the first chosen password by each subject was considered. There are 762 passwords that have 2,286 LdGs. Like previous analysis, the training dataset excluded passwords from the target picture. As shown in Figure 6, results of this analysis are not as good as previous ones. Unbiased with P_{L-15}^2 breaks 160 passwords (21.0%) out of 762. Unbiased with P_{A-40}^2 cracking 123 passwords (16.1%). BestCover cracks 108 (14.2%) and 116 (15.2%) with P_{L-15}^2 and P_{A-40}^2 , respectively.



(a)



(b)

Figure 7: (a) Percentage of passwords cracked vs. number of password guesses, per condition. (b) Percentage of LdGs cracked vs. number of password guesses, per condition. Only passwords for pictures 243, 1116, 2057, 4054, 6467, and 9899 were considered. There are 4,003 passwords that have 12,009 LdGs.

Since some pictures in *Dataset-2* are similar, we ran an additional analysis in which only passwords for pictures 243 (airplane), 1116 (portrait), 2057 (car), 4054 (wedding), 6467 (bicycle), and 9899 (dog) were considered. There are 4,003 passwords that have 12,009 LdGs. Unbiased with P_{L-15}^2 breaks 1,147 passwords (28.6%) while 803 passwords (20.1%) are cracked by Unbiased with P_{A-40}^2 . BestCover cracks 829 (20.7%) and 875 (21.8%) with P_{L-15}^2 and P_{A-40}^2 respectively. Results of this analysis are not as good as results with passwords from all pictures.

Online Attacks. The current Windows 8TM allows five failure attempts before it forces users to enter their text-based passwords. Therefore, breaking a password under five guesses implies the feasibility for launching an online attack. Figure 8 shows a refined view of the number of passwords and LdGs cracked with the first five guesses per condition. Purely automated attack Unbiased with P_{A-40}^2 breaks 83 passwords (0.8%) with the first guess and cracks 94 passwords (0.9%) within the first five guesses, while BestCover with P_{A-40}^2 cracked 20 passwords (0.2%) for the first guess and 38 passwords (0.4%) within five guesses. Additionally, Unbiased with P_{A-40}^2 breaks 1,723 LdGs (5.7%) with the first guess. With the help of manually labeled PoI set P_{L-15}^2 , the results are even better. For example, Unbiased breaks 195 passwords (1.9%) for the first guess and 266 (2.6%) within the first five guesses. In the meantime, Unbi-

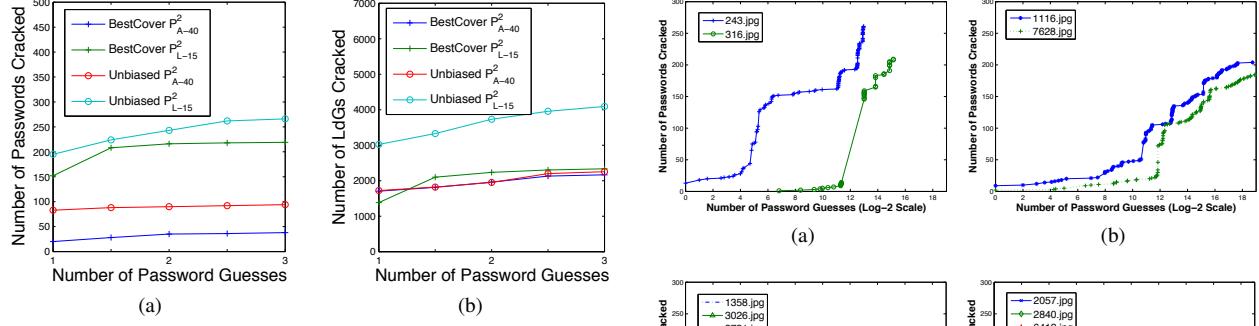


Figure 8: (a) Number of passwords cracked within five guesses, per condition. (b) Number of LdGs cracked within five guesses, per condition.

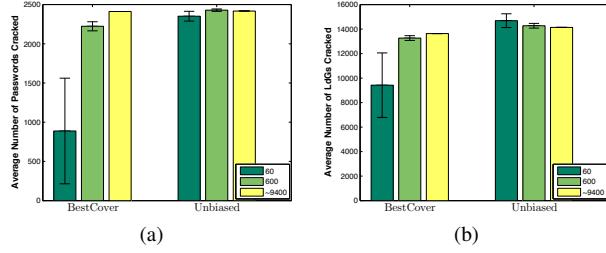


Figure 9: (a) Average number of passwords cracked vs. different training data sizes. (b) Average number of LdGs cracked vs. different training data sizes. P_{A-40}^2 is used for this analysis. Average over 3 analyses, with one standard deviation shown.

ased with P_{L-15}^2 breaks 3,022 LdGs (10.0%) with the first guess and 4,090 LdGs (13.5%) with five guesses.

Effects of Training Data Size. In Figure 9, we show the password and LdG cracking results with different sizes of training datasets. For each algorithm, we used P_{A-40}^2 as the PoI set and performed three analyses with 60, 600, and all available passwords (about 9,400) as training data, respectively. The sizes of 60 and 600 represent two cases: i) a training set (60) is ten times smaller than the target set (about 669); and ii) a training set (600) is almost the same size as the target set (about 669). For training datasets with the sizes of 60 and 600, we randomly selected these training passwords and performed each analysis three times to get the averages and standard deviations.

As Figure 9 shows, BestCover with 60 training samples could only break an average of 888 passwords (8.8%) out of 10,039. And the standard deviation is as strong as 673. While Unbiased with 60 training samples can crack 2,352 passwords (23.4%) that is almost the same as the results generated from all available training samples. Also, the standard deviation for three trials is as low as 62. The results from BestCover with 600 training samples are much better than the counterparts with 60 training samples. All these observations are expected as Unbiased could eliminate the biases considered

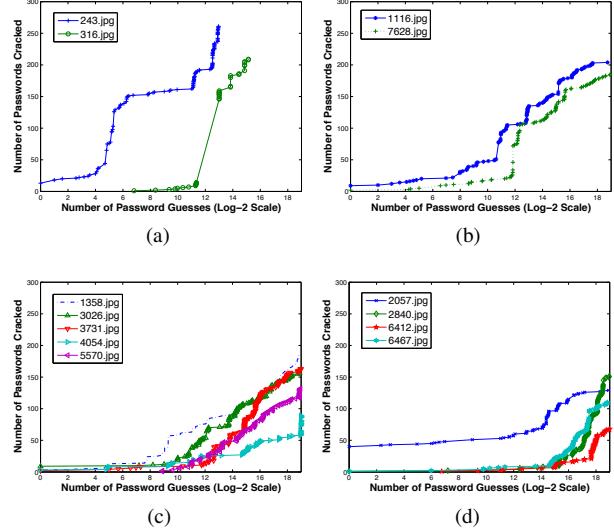


Figure 10: (a) pictures with fewer PoIs (b) portraits (c) pictures with people in them (d) pictures with lots of PoIs. Unbiased algorithm on P_{A-40}^2 is used for this analysis. (Please refer to Appendix B for the pictures).

in BestCover. The results clearly demonstrate the benefit of using the Unbiased algorithm when a training dataset is small.

Effects on Different Picture Categories. We measured the attack results on different picture categories as shown in Figure 10 where each subfigure depicts the number of passwords cracked versus the number of password guesses. Each curve in a subfigure corresponds to a picture as shown in the legend. Our approach cracks more passwords for a picture, if the curve is skewed upward. And the cracking is faster (with fewer guesses), if the curve is leaned toward the left.

Figure 10(a) provides a view of the attack results on target pictures 243 and 316, each of which has only one airplane flying in the sky. Fewer PoIs in these two pictures make subjects choose more similar passwords. Unbiased with P_{A-40}^2 breaks 261 passwords (39.0%) for the picture 243 and 209 (31.2%) for the picture 316. The cracking success rates are much higher than the average success rate in *Dataset-2* under the same condition. Note that the size of generated dictionaries for these two pictures are smaller than 2^{19} due to the number of available PoIs.

In Figure 10(b), we show the results on two *portrait* pictures where Unbiased with P_{A-40}^2 cracks 389 passwords (29.0%) for both in total. The attack success rate is much higher than the average success rate in *Dataset-2*. This is due to the fact that state-of-the-art computer vision algorithms work well on facial landmarks and subjects' tendencies of drawing on these features are high. The results show that passwords on simple pictures with fewer PoIs or portraits, for which state-of-the-art com-

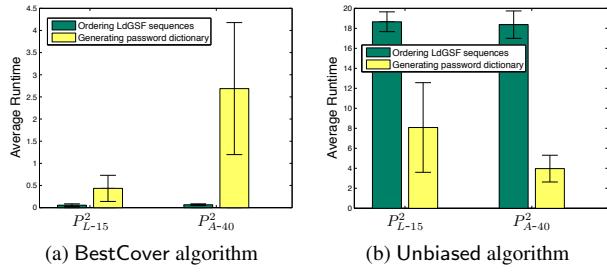


Figure 11: Average runtime in seconds to order LdGSF sequences using BestCover and Unbiased. Average over 15 pictures in *Dataset-2* with one standard deviation shown.

puter vision techniques could detect PoIs with high accuracy, are easier for attackers to break.

Figure 10(c) shows the attack results on 5 pictures of people. Some of these pictures only have very small figures of people and others have larger figures but not big enough to be considered as a portrait. Unbiased with P_{A-40}^2 cracks 726 passwords (21.7%) for these 5 pictures in total, which is lower than the average success rate in *Dataset-2*.

Figure 10(d) shows the attack results on 4 miscellaneous pictures, two of which are bicycle pictures and the other two are car pictures. The picture, 6412.jpg, has a bicycle leaning against the wall. Different colors on the bicycle and wall in this picture make it cluttered and have lots of PoIs. Unbiased with P_{A-40}^2 only cracks 68 passwords (10.1%) for this picture. However, Unbiased with P_{A-40}^2 cracked 458 (17.1%) for all 4 pictures.

Performance. We also evaluated the performance of our attack approach. Our analyses were carried out on a computer with dual-core processor and 4GB of RAM. In Figure 11, we show the average runtime for our algorithms to order the LdGSF sequences and generate dictionary for a picture in *Dataset-2*. Each bar represents the average time in seconds over 15 pictures with the standard deviation using different algorithms and PoI sets. The results show that BestCover is much faster than Unbiased under the same condition. The average runtime for BestCover on P_{A-40}^2 to order LdGSF sequences is only 0.06 seconds and to generate a dictionary is 2.68 seconds, while Unbiased spends 18.36 and 3.96 seconds, respectively. As we analyzed in Section 4.4, such a difference is caused by the complexity of each algorithm. With such a prompt response, BestCover could be used for online queries.

6 Discussion

6.1 Picture-Password-Strength Meter

Our framework could enhance the security of PGA so it would eventually protect users and their devices by pro-

viding a picture-password-strength meter. One way to help users choose secure passwords is to enforce some composition policies, such as ‘three taps are not allowed’. However, a recent effort [26] on text-based password found that rule-based password compositions are ineffective because they can allow weak passwords and reject strong ones. The cornerstone of accurate strength measurement is to quantify the strength of a password. With a ranked password dictionary, our framework, as the first potential picture-password-strength meter, is capable of quantifying the strength of selected picture passwords. More intuitively, a user could be informed of the potential number of guesses for breaking a selected password through executing our attack framework.

6.2 Other Attacks on PGA

Besides keyloggers that record users’ finger movements, there are some other attack methods that may affect the security of PGA and other background draw-a-secret schemes. Shoulder surfing, an attack where attackers simply observe the user’s finger movements, is one of them. In our survey, 54.3% participants believe the picture password scheme is easier for attackers to observe when they are providing their credentials than text-based password. Several new shoulder surfing resistant schemes [22, 43] were proposed recently. However, the usability is always a major concern for these approaches. The smudge attack [4] which recovers passwords from the oily residues on a touch-screen has also been proven feasible to the background draw-a-secret schemes and could pose threats to PGA.

6.3 Limitations of Our Study

While we took great efforts to maintain our studies’ validity, some design aspects of our studies and developed system may have caused subjects to behave differently from what they do on Windows 8™ PGA. Subjects in *Dataset-2* pretended to access their bank information but did not have anything at risk. Schechter et al. [33] suggest that role playing like this affects subjects’ security behavior, so passwords in *Dataset-2* may not be representative of real passwords chosen by real users. Besides, we did not record whether a subject used a tablet with touch-screen or a desktop with mouse. The different ways of input may affect the composition of passwords. Moreover, *Dataset-2* includes multiple passwords per user and this may have impacted the results. In our analyses, training password datasets include passwords from the targeted subject. Even though this may have affected the results, we believe it is less influential. Because, for each analysis, there were around 9,400 training passwords for which only 14 came from the targeted user.

Since all training passwords were treated equally, the influence brought by the 0.14% training data is low. As discussed in Section 5.2, even though our online attack results showed the feasibility of our approach, it still requires more realistic and significant attack cases. As part of future work, we plan to integrate smudge attacks [4] into our framework to improve the efficacy of our online attacks.

7 Related Work

The security and vulnerability of text-based password have attracted considerable attention because of several infamous password leakage incidents in recent years. Zhang et al. [44] studied the password choices over time and proposed an approach to attack new passwords from old ones. Castelluccia et al. [11] proposed an adaptive Markov-based password strength meter by estimating the probability of password using training data. Kelley et al. [26] developed a distributed method to calculate how effectively password-guessing algorithms could guess passwords. Even though the attack framework we presented is dedicated to cracking background draw-a-secret passwords, the idea of abstracting users' selection processes of password construction introduced in this paper could also be applicable to cracking and measuring text-based passwords.

The basic idea of attacking graphical password schemes is to generate dictionaries that consist of potential passwords [36]. However, the lack of sophisticated mechanisms for dictionary construction affects the attack capabilities of existing approaches. Thorpe et al. [38] proposed a method to harvest the locations of training subjects' clicks on pictures in click-based passwords to attack other users' passwords on the same pictures. In the same paper [38], they presented another approach which creates dictionaries by predicting hot-spots using image processing methods. Oorschot et al. [27] cracked DAS using some password complexity factors, such as reflective symmetry and stroke-count. Salehi-Abari et al. [32] proposed an automated attack on the PassPoints scheme by ranking passwords with click-order patterns. However, the click-order patterns introduced in their approach could not capture users' selection processes accurately, especially when a background image significantly affects user choice.

8 Conclusion

We have presented a novel attack framework against background draw-a-secret schemes with special attention on picture gesture authentication. We have described an empirical analysis of Windows 8™ picture gesture authentication based on our user studies. Using the pro-

posed attack framework, we have demonstrated that our approach was able to crack a considerable portion of picture passwords in various situations. We believe the findings and attack results discussed in this paper could advance the understanding of background draw-a-secret and its potential attacks.

Acknowledgements

The authors are grateful to Lujo Bauer of Carnegie Mellon University and Sonia Chiasson of Carleton University for useful comments while this work was in progress. The authors also thank the anonymous reviewers whose comments and suggestions have significantly improved the paper.

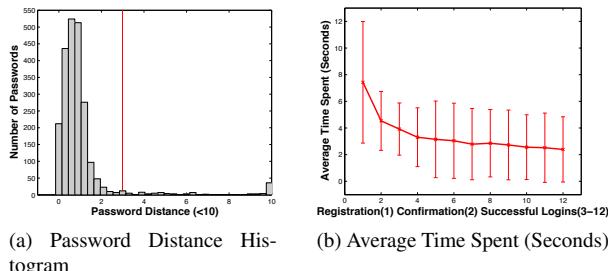
References

- [1] OpenCV. <http://opencv.willowgarage.com>.
- [2] ALEXE, B., DESELAERS, T., AND FERRARI, V. Objectness measure v1.5. <http://groups.inf.ed.ac.uk/calvin/objectness/objectness-release-v1.5.tar.gz>.
- [3] ALEXE, B., DESELAERS, T., AND FERRARI, V. Measuring the objectness of image windows. *IEEE Transactions Pattern Analysis and Machine Intelligence* (2012).
- [4] AVIV, A., GIBSON, K., MOSSOP, E., BLAZE, M., AND SMITH, J. Smudge attacks on smartphone touch screens. In *Proceedings of the 4th USENIX conference on Offensive technologies* (2010), USENIX Association, pp. 1–7.
- [5] BALLARD, D. Generalizing the hough transform to detect arbitrary shapes. *Pattern recognition* 13, 2 (1981), 111–122.
- [6] BICAKCI, K., ATALAY, N., YUCEEL, M., GURBASLAR, H., AND ERDENIZ, B. Towards usable solutions to graphical password hotspot problem. In *Proceedings of the 33rd Annual IEEE International on Computer Software and Applications Conference* (2009), vol. 2, IEEE, pp. 318–323.
- [7] BIDDLE, R., CHIASSON, S., AND VAN OORSCHOT, P. Graphical passwords: Learning from the first twelve years. *ACM Computing Surveys* 44, 4 (2011), 2012.
- [8] BONNEAU, J., PREIBUSCH, S., AND ANDERSON, R. A birthday present every eleven wallets? the security of customer-chosen banking pins. *Financial Cryptography and Data Security* (2012), 25–40.
- [9] BROSTOFF, S., AND SASSE, M. Are passfaces more usable than passwords? a field trial investigation. *People And Computers* (2000), 405–424.
- [10] CANNY, J. A computational approach to edge detection. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 6 (1986), 679–698.
- [11] CASTELLUCCIA, C., DÜRMUTH, M., AND PERITO, D. Adaptive password-strength meters from markov models. In *Proceedings of the 19th Network and Distributed System Security Symposium* (2012), vol. 2012.
- [12] CHIASSON, S., FORGET, A., BIDDLE, R., AND VAN OORSCHOT, P. User interface design affects security: Patterns in click-based graphical passwords. *International Journal of Information Security* 8, 6 (2009), 387–398.

- [13] CHIASSON, S., STOBERT, E., FORGET, A., BIDDLE, R., AND VAN OORSCHOT, P. Persuasive cued click-points: Design, implementation, and evaluation of a knowledge-based authentication mechanism. *IEEE Transactions on Dependable and Secure Computing* 9, 2 (2012), 222–235.
- [14] CHIASSON, S., VAN OORSCHOT, P., AND BIDDLE, R. Graphical password authentication using cued click points. Springer, pp. 359–374.
- [15] DAVIS, D., MONROSE, F., AND REITER, M. On user choice in graphical password schemes. In *Proceedings of the 13th conference on USENIX Security Symposium* (2004), USENIX Association, pp. 11–11.
- [16] DHAMIJA, R., AND PERRIG, A. Déjà vu: A user study using images for authentication. In *Proceedings of the 9th conference on USENIX Security Symposium* (2000), USENIX Association.
- [17] DIRIK, A. E., MEMON, N., AND BIRGET, J.-C. Modeling user choice in the passpoints graphical password scheme. In *Proceedings of the 3rd symposium on Usable privacy and security* (2007), ACM, pp. 20–28.
- [18] DUNPHY, P., AND YAN, J. Do background images improve draw a secret graphical passwords? In *Proceedings of the 14th ACM conference on Computer and communications security* (2007), ACM, pp. 36–47.
- [19] EVERINGHAM, M., VAN GOOL, L., WILLIAMS, C. K. I., WINN, J., AND ZISSERMAN, A. The PASCAL Visual Object Classes Challenge 2007 (VOC2007) Results. <http://www.pascal-network.org/challenges/VOC/voc2007/workshop/index.html>.
- [20] FEIGE, U., LOVÁSZ, L., AND TETALI, P. Approximating min sum set cover. *Algorithmica* 40, 4 (2004), 219–234.
- [21] FOLEY, M. J. Microsoft: 60 million windows 8 licenses sold to date. <http://www.zdnet.com/microsoft-60-million-windows-8-licenses-sold-to-date-7000009549/>, 2013.
- [22] FORGET, A., CHIASSON, S., AND BIDDLE, R. Shoulder-surfing resistance with eye-gaze entry in cued-recall graphical passwords. In *Proceedings of the 28th international conference on Human factors in computing systems* (2010), ACM, pp. 1107–1110.
- [23] GAO, H., GUO, X., CHEN, X., WANG, L., AND LIU, X. Yagg: Yet another graphical password strategy. In *Proceedings of the 24th Annual Computer Security Applications Conference* (2008), IEEE, pp. 121–129.
- [24] JERMYN, I., MAYER, A., MONROSE, F., REITER, M., AND RUBIN, A. The design and analysis of graphical passwords. In *Proceedings of the 8th USENIX Security Symposium* (1999), Washington DC, pp. 1–14.
- [25] JOHNSON, J. Picture gesture authentication, US Patent 163201, 2012.
- [26] KELLEY, P., KOMANDURI, S., MAZUREK, M., SHAY, R., VIDAS, T., BAUER, L., CHRISTIN, N., CRANOR, L., AND LOPEZ, J. Guess again (and again and again): Measuring password strength by simulating password-cracking algorithms. In *Proceedings of the IEEE Symposium on Security and Privacy* (2012), IEEE, pp. 523–537.
- [27] OORSCHOT, P., AND THORPE, J. On predictive models and user-drawn graphical passwords. *ACM Transactions on Information and system Security (TISSEC)* 10, 4 (2008), 5.
- [28] OVIDE, S. Microsoft's windows 8 test: Courting consumers. <http://online.wsj.com/article/SB1000142405297020453050457-8078743616727514.html>.
- [29] PACE, Z. Signing in with a picture password. <http://blogs.msdn.com/b/b8/archive/2011/12/16/signing-in-with-a-picture-password.aspx>.
- [30] PACE, Z. Signing into windows 8 with a picture password. <http://www.youtube.com/watch?v=Ek9N2tQzHOA>.
- [31] REIMONDO, A. Haar cascades. <http://alereimondo.no-ip.org/OpenCV/34>.
- [32] SALEHI-ABARI, A., THORPE, J., AND VAN OORSCHOT, P. On purely automated attacks and click-based graphical passwords. In *Proceedings of the 24th Annual Computer Security Applications Conference* (2008), IEEE, pp. 111–120.
- [33] SCHECHTER, S. E., DHAMIJA, R., OZMENT, A., AND FISCHER, I. The emperor's new security indicators. In *Proceedings of the 2007 IEEE Symposium on Security and Privacy* (2007), IEEE, pp. 51–65.
- [34] SUO, X., ZHU, Y., AND OWEN, G. Graphical passwords: A survey. In *Proceedings of the 21st Annual Computer Security Applications Conference* (2005), IEEE, pp. 10–19.
- [35] SUZUKI, S., ET AL. Topological structural analysis of digitized binary images by border following. *Computer Vision, Graphics, and Image Processing* 30, 1 (1985), 32–46.
- [36] THORPE, J., AND VAN OORSCHOT, P. Graphical dictionaries and the memorable space of graphical passwords. In *Proceedings of the 13th conference on USENIX Security Symposium* (2004), USENIX Association, pp. 10–10.
- [37] THORPE, J., AND VAN OORSCHOT, P. Towards secure design choices for implementing graphical passwords. In *Proceedings of the 20th Annual Computer Security Applications Conference* (2004), IEEE, pp. 50–60.
- [38] THORPE, J., AND VAN OORSCHOT, P. Human-seeded attacks and exploiting hot-spots in graphical passwords. In *Proceedings of 16th USENIX Security Symposium* (2007), USENIX Association, p. 8.
- [39] VAN OORSCHOT, P., AND THORPE, J. Exploiting predictability in click-based graphical passwords. *Journal of Computer Security* 19, 4 (2011), 669–702.
- [40] VIOLA, P., AND JONES, M. Robust real-time face detection. *International journal of computer vision* 57, 2 (2004), 137–154.
- [41] WIEDENBECK, S., WATERS, J., BIRGET, J., BRODSKIY, A., AND MEMON, N. Authentication using graphical passwords: effects of tolerance and image choice. In *Proceedings of the Symposium on Usable privacy and security* (2005), ACM, pp. 1–12.
- [42] YUILLE, J. C. *Imagery, memory, and cognition*. Lawrence Erlbaum Assoc Inc, 1983.
- [43] ZAKARIA, N., GRIFFITHS, D., BROSTOFF, S., AND YAN, J. Shoulder surfing defence for recall-based graphical passwords. In *Proceedings of the 7th Symposium on Usable Privacy and Security* (2011), ACM, p. 6.
- [44] ZHANG, Y., MONROSE, F., AND REITER, M. The security of modern password expiration: An algorithmic framework and empirical analysis. In *Proceedings of the 17th ACM conference on Computer and communications security* (2010), ACM, pp. 176–186.

A Memorability and Usability Analysis

The tolerance introduced in PGA is a trade-off between security and usability. In order to quantify this tradeoff, we calculate the distance between input PGA passwords with the registered ones. When the types or directions of gestures do not match, we regard input passwords incomparable with the registered ones. Otherwise, the distance



(a) Password Distance Histogram

(b) Average Time Spent (Seconds)

Figure 12: Memorability and Usability

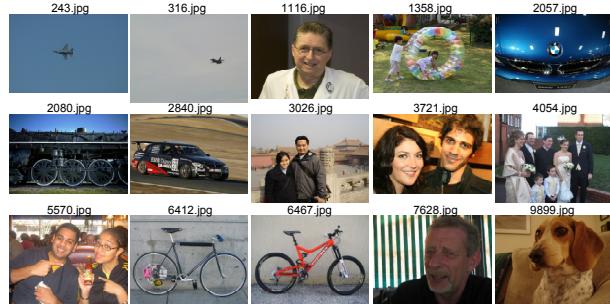
is defined as the average distance of all gestures. We denote the password presented for the i -th attempt $\vec{\pi}^{(i)}$ and $\vec{\pi}^{(0)}$ as the password registered for the same picture.

In the 2,536 login attempts collected in *Dataset-1*, 422 are unsuccessful in which 146 are type or direction errors and 276 are distance errors. Figure 12(a) shows the distance distribution for the password whose distance is less than 10 and the red line denotes the threshold for being classified as successful. The result shows the current setup in our system is quite reasonable to capture most closely presented passwords.

Figure 12(b) shows the average time in seconds that subjects spent on registering, confirming, and reproducing passwords. $x = 1$ denotes the registration, $x = 2$ denotes the confirmation, and all others denote the later login attempts. As we can notice, the average time for the registration is 7.43 seconds while 4.53 seconds are taken for the confirmation. With subjects getting used to the picture password system, the average time spent for successful logins is reduced to as low as 2.51 seconds. On the other hand, the average time spent on all unsuccessful login attempts is 5.86 seconds.

B Dataset-2 Pictures

Figure 13 shows 15 images that are used in *Dataset-2* as the background pictures for password selection.

Figure 13: Background Pictures Used in *Dataset-2*

C LdGSF Identification

We discuss the identified LdGSFs by linking PoIs and passwords in *Dataset-2* with the help of two PoI sets

P_{L-15}^2 and P_{A-40}^2 using our LdGSF identification algorithm discussed in Section 4.3. The results from P_L are closer to users' actual selection processes, while the results from P_A are the best approximations to users' selection processes we could get in a purely automated way with state-of-the-art computer vision techniques.

Table 5: Top 10 Identified LdGSFs using P_{L-15}^2

Rank	$Pr(s_k)$	$Pr(s_k D_{s_k} \subseteq \theta)$
1	(tap, {head}, \emptyset)	(tap, {nose}, \emptyset)
2	(tap, {* _c }, \emptyset)	(tap, {mouth}, \emptyset)
3	(tap, {circle}, \emptyset)	(tap, {circle}, \emptyset)
4	(tap, {eye}, \emptyset)	(tap, {eye}, \emptyset)
5	(circle, {head}, \emptyset)	(tap, {* _c }, \emptyset)
6	(tap, {nose}, \emptyset)	(tap, {head}, \emptyset)
7	(circle, {circle}, \emptyset)	(circle, {circle}, \emptyset)
8	(circle, {eye}, \emptyset)	(tap, {ear}, \emptyset)
9	(line, {* _c }, {* _c })	(line, {mouth}, {mouth})
10	(line, {eye}, {eye})	(tap, {forehead}, \emptyset)

The top ten identified LdGSFs using P_{L-15}^2 are shown in Table 5 ordered by their $Pr(s_k)$ and $Pr(s_k | D_{s_k} \subseteq \theta)$. It also suggests that ‘tap a head’ is found the most times in the passwords, while ‘tap a nose’ is the most popular one when there is a nose in the picture. The result seems unreasonable at the first glance since there is always a nose in a head. Actually, it is because if the head in the picture is really small, we simply annotate the circumscribed rectangle as head instead of marking the inner rectangles with more specific attributes. Table 5 indicates that gestures on human organs are the most popular selection functions adopted by subjects.

Table 6: Top 10 Identified LdGSFs using P_{A-40}^2

Rank	$Pr(s_k)$	$Pr(s_k D_{s_k} \subseteq \theta)$
1	(tap, {circle}, \emptyset)	(tap, {clock}, \emptyset)
2	(tap, {mouth}, \emptyset)	(circle, {clock}, \emptyset)
3	(tap, {eye}, \emptyset)	(tap, {shoulder}, \emptyset)
4	(tap, {head}, \emptyset)	(tap, {eye}, \emptyset)
5	(tap, {* _c }, \emptyset)	(tap, {head}, \emptyset)
6	(tap, {* _c }, \emptyset)	(tap, {body}, \emptyset)
7	(circle, {eye}, \emptyset)	(tap, {mouth}, \emptyset)
8	(tap, {body}, \emptyset)	(tap, {circle}, \emptyset)
9	(circle, {circle}, \emptyset)	(tap, {* _c }, \emptyset)
10	(circle, {head}, \emptyset)	(tap, {* _c }, \emptyset)

The top ten identified LdGSFs using P_{A-40}^2 are shown in Table 6. By comparing Table 5 and Table 6, we could notice differences caused by using annotated PoI set and automated detected PoI set. The fact that $s(\text{tap}, {*}, \emptyset)$ is among the top ten LdGSFs is an indicator that the automatic PoI identification could not classify many PoIs and simply mark them as *. It is surprising to find out there are two LdGs on clock in top ten ordered by $Pr(s_k | D_{s_k} \subseteq \theta)$ at first, because there is no clock in any picture in *Dataset-2*. The closest guess is OpenCV falsely identified some circle shape objects as clocks, but the number is not very big since there is no LdG on a clock in the top ten ordered by $Pr(s_k)$.

Explicating SDKs: Uncovering Assumptions Underlying Secure Authentication and Authorization

Rui Wang^{† *}, Yuchen Zhou^{‡ *}, Shuo Chen[†], Shaz Qadeer[†], David Evans[‡], Yuri Gurevich[†]

[†] Microsoft Research
Redmond, WA, USA

{ruiwan, shuochen, qadeer, gurevich}@microsoft.com

[‡] University of Virginia
Charlottesville, VA, USA
{yz8ra, evans}@virginia.edu

Abstract

Most modern applications are empowered by online services, so application developers frequently implement authentication and authorization. Major online providers, such as Facebook and Microsoft, provide SDKs for incorporating authentication services. This paper considers whether those SDKs enable typical developers to build secure apps. Our work focuses on systematically explicating implicit assumptions that are necessary for secure use of an SDK. Understanding these assumptions depends critically on not just the SDK itself, but on the underlying runtime systems. We present a systematic process for identifying critical implicit assumptions by building semantic models that capture both the logic of the SDK and the essential aspects of underlying systems. These semantic models provide the explicit basis for reasoning about the security of an SDK. We use a formal analysis tool, along with the semantic models, to reason about all applications that can be built using the SDK. In particular, we formally check whether the SDK, along with the explicitly captured assumptions, is sufficient to imply the desired security properties. We applied our approach to three widely used authentication/authorization SDKs. Our approach led to the discovery of several implicit assumptions in each SDK, including issues deemed serious enough to receive Facebook bug bounties and change the OAuth 2.0 specification. We verified that many apps constructed with these SDKs (indeed, the majority of apps in our study) are vulnerable to serious exploits because of these implicit assumptions, and we built a prototype testing tool that can detect several of the vulnerability patterns we identified.

1 Introduction

Modern applications commonly consist of a client program and an online service that provides functionality such as cloud storage, social networking, and geographic data. Accessing the service requires authentication of users and authorization of resource requests. Traditionally, the authentication and authorization mechanisms were provided by operating systems and carefully implemented in a few core apps such as SSH, remote desktop, etc; with modern apps, however, many developers end up needing to implement such mechanisms. To aid this, major identity providers have developed SDKs that developers can use to integrate authentication and authorization into their apps such as the three SDKs we study in this work: the Facebook PHP SDK, the Microsoft Live Connect SDK, and the Windows 8 Authentication Broker SDK. According to our sampling of popular apps in Windows App Store, 52% of them use these SDKs (see Appendix A).

Authentication/authorization SDKs are becoming a critical foundation for apps. However, no previous study has rigorously examined the security these SDKs provide to real-world apps. Typically, SDK providers simply release SDK code, publish documentation and ex-

amples, and leave the rest to app developers. An important question remains: *if developers use the SDKs in reasonable ways, will the resulting applications be secure?* We show in this paper that the answer today is “No”. The majority of apps built using the SDKs we studied have serious security flaws. This is not due to direct vulnerabilities in the SDK, but rather because achieving desired security properties by using an SDK depends on many implicit assumptions that are not readily apparent to app developers. These assumptions are not documented anywhere in the SDK or its developer documentation. In several cases, even the SDK providers are unaware of the assumptions (see Section 5.2).

The goal of our work is to systematically identify the assumptions needed to use an SDK to produce secure applications. We emphasize that it is not meaningful to verify an SDK by itself. Instead, our goal is to explicate the assumptions upon which secure use of the SDK depends. We do this by devising precise definitions of desired security properties, constructing an explicit model of the SDK and the complex services with which it interacts, and systematically exploring the space of applications that can be built using the SDK. Our approach involves a combination of manual effort and automated formal verification. Any counterexample found by the

* The two lead authors are ordered alphabetically.

verification tool indicates either (1) that our system models are not accurate, in which case we revisit the real systems to correct the model; or (2) that our models are correct, but additional assumptions need to be captured in the model and followed by application developers. The explication process is an iteration of the above steps so that we document, examine and refine our understanding of the underlying systems for an SDK. At the end, we get a set of formally captured assumptions and a semantic model that allow us to make meaningful assurances about the SDK: *an application constructed using the SDK following the documented assumptions satisfies desired security properties.*

We argue that explication should be part of the engineering process of developing an SDK. Identified SDK assumptions can either be removed by modifying the SDK, or be documented precisely. In addition, in some cases it is feasible to develop automatic tests that detect common ways applications violate the assumptions (we provide an example in Section 6.2).

Results. The work presented in this paper reflects a 12 person-month effort (six months of two lead authors) in systematically explicating the three target SDKs. The resulting models (<https://github.com/sdk-security>) are publicly released so that the community can review and enhance them. As a result of the explication process, we uncovered many SDK assumptions (summarized in Section 5). Some assumptions were especially serious because they can be violated when an app developer has a reasonable alternative interpretation of the developer’s guide (*dev guide*) or when an app runs on certain realistic platforms. These reports were treated very seriously by the SDK providers: five cases that we reported to Facebook have been fixed (three of which were rewarded by Facebook bounties [14]). An issue uncovered in the Live Connect SDK resulted in Microsoft improving its dev guide. Our report to the OAuth Working Group convinced the group to add a subsection to the OAuth 2.0 draft.

With all the SDK assumptions specified, we were able to successfully verify all the models with the uncovered assumptions (Section 4). Uncovering these SDK assumptions also enables effective app testing since a violation of an assumption often leads to a successful exploit. Our study shows that many released apps are indeed vulnerable due to violations of these assumptions. We tested three sets of apps, including client apps in Windows 8 App Store and service apps using Facebook sign-on, and found that 78%, 86% and 67% of these apps suffer from vulnerabilities related to the implicit assumptions we uncovered (Section 6.2).

2 Illustrative Example

To motivate our work, we describe a simple example in the context of the Live Connect SDK. It illustrates what can go wrong when SDKs are provided without thoroughly specifying their underlying security assumptions.

2.1 Intended Use

Suppose we want to develop an app using Live ID as the Identity Provider (IdP). We start with the dev guide for Live Connect [25]. The hyperlinks in the start page lead us to a page of detailed instructions about “signing users in” [26] which provides code snippets in Javascript, C#, Objective-C and Java showing how to use Live Connect SDK to sign users into a client app. Ignoring the specifics in these different languages, all the code snippets essentially cover the authentication logic shown in Figure 1.

In the figure, WL stands for “Windows Live”. A developer first needs to call WL.login. The call takes an argument value, “wl.basic”, indicating that the app will need to read the user’s basic information after WL.login returns an access token in step (2). The access token is a concept in the OAuth protocol [22]. It is an opaque string dynamically created by the Live ID server for each call to WL.login. Once the app gets the access token, it calls the REST API me to get the user’s basic info using this HTTP request:

https://apis.live.net/v5.0/me?access_token=ACCESS_TOKEN

The Live ID service responds with the user’s basic information in message (4), such as her full name and user ID. This completes the process, authenticating the user with the provided information.

2.2 Hazardous Use

The developer guide as depicted in Figure 1 is valid for a client-only app, but it does not make it clear that the same logic *must not* be used with an app that also incorporates an online service. Without stating this explicitly, developers may be inclined to use the SDK insecurely as shown in Figure 2. The interactions with the Live ID

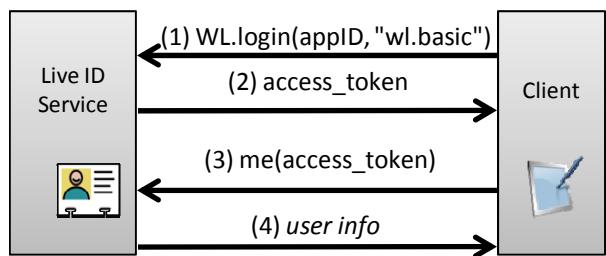


Figure 1. Authentication Logic for “Signing Users In”.

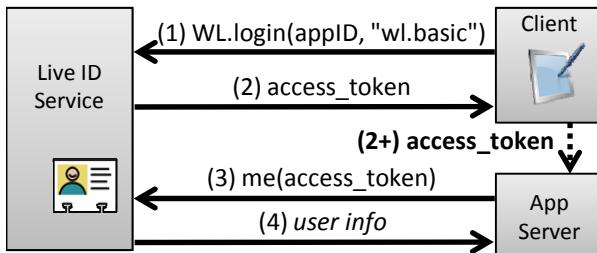


Figure 2. Hazardous Use.

service are identical in the two figures. The only difference is that in the second scenario, the access token is sent to the service app (i.e., the server side of the app) in message (2+) and it is the service app that calls me to authenticate the user.

This can lead to a serious vulnerability that allows any app on the device to sign into the service app as the user. The rogue app sends a request to the Live ID service for an access token to view public information for the victim, such as a profile record on Facebook. Live ID responds with an access token. The problem is this token, intended for authorizing access to the public resource, is mistakenly used by the service app to authenticate its owner as the victim. This allows the rogue app to get into the victim’s account on the service app. This mistake is fairly common in real-world apps. Although we first observed it analyzing the Live Connect SDK, we later found that many apps using the Facebook SDK have the same issue. As described in Section 6.2, we tested 27 apps in the Windows 8 App Store and found that 21 of them are vulnerable due to this mistake.

2.3 Resolution

From one perspective, this is simply a matter of developers writing buggy apps, and the blame for the security vulnerability rests with the app developers. We argue, though, that the purpose of the SDK is to enable typical developers to produce apps that use authentication and authorization in a way that provides desired security properties, and the prevalence of buggy apps created using this SDK indicates a failure of the larger engineering process. The developer exercised reasonable prudence by using the access token to query the ID service for user information and followed exactly the process described in the SDK’s documentation (depicted in Figure 1). The problem is lack of a deeper understanding of the relationship between authentication and authorization, and the role of the access token (i.e., why is it safe to use the access token as shown in Figure 1 but not as used in Figure 2). Correct use depends on subtle understanding of what kind of evidence each message represents and whether or not the whole message sequence establishes an effective proof for a security decision. It is unrealistic to expect most

developers to understand these subtleties, especially without clear guidance from the SDK.

We contacted the developers of some of the vulnerable apps. A few apps have been fixed in response to our reports. We also notified the OAuth Working Group (WG) in June 2012 about these vulnerable apps.¹ Dick Hardt, editor of OAuth 2.0 specification (RFC 6749) [22], emailed us requesting language to be included in the specification dealing with this issue. We proposed the initial text and discussed with WG members. This resulted in Section 10.16 “*Misuse of Access Token to Impersonate Resource Owner in Implicit Flow*” being added to the OAuth specification.

The key point this example illustrates is that security of apps constructed with an SDK depends on an understanding of the external services the app depends on, as well as subtleties in the use of tokens and assumptions about evidence used in authentication and authorization decisions. We believe the prevalence of vulnerable apps constructed using current SDKs is compelling evidence that a better engineering process is needed, rather than just passing the blame to overburdened developers. In particular, we advocate for a process that explicates SDKs by systematically identifying the underlying assumptions upon which secure usage depends.

3 Explicating SDKs

In order to explicate the SDKs, we need to clearly define the desired security properties. This section introduces our target scenario and threat model, and then defines the desired security properties and overviews our process for uncovering implicit SDK assumptions.

3.1 Scenario

A typical question about security is whether some property holds for a system, even in the presence of an adversary interacting with the system in an unconstrained manner. We can view this as a software testing problem: the system is a concrete program, while the adversary is an abstract one (i.e., a *test harness* in the terminology of software testing) that explores all interaction sequences with the concrete system. In our scenario, however, the target system is not concrete. We wish to reason about *all* applications that can be built with the SDK follow-

¹ Subsequently, we learned that John Bradley, a WG member, had posted a blog post in January 2012 about a similar issue [10]. The post considers the problem a vulnerability of the protocol, while we view it as a consequence of an unclear assumption about SDK usage because there are correct ways to use OAuth for client+service authentication.

ing documented guidelines. Hence, we need to consider both the client app and service as abstract modules.

Figure 3 illustrates the modules in our setup. There are three main components: a client device, the application server *foo.com*, and the identity provider (IdP). The bottom layer of the client device is the client runtime, such as the HTML engine or the HTTP layer. The middle layer is the client SDK. The client app, *FooApp_C*, is created by the developer to interact with the application server. We assume *FooApp_C* always uses the client SDK for authentication and authorization. Like the client, the application server has three layers: the service runtime represents the server platform, such as PHP or ASP.NET; the server side of the SDK we study; and the application server code. We assume that *FooApps* does not directly interact with the service runtime, but only uses it via the service SDK. Note that both *FooApp_C* and *FooApps* identify themselves to IdP as “*FooApp*” with an app ID pre-assigned by IdP. The IdP cannot tell if the caller is a client or the application server.

The modules with brick pattern backgrounds are *concrete modules* with concrete implementations. They can be divided into two layers. *The SDK layer* consists of the Client SDK and the Service SDK. The *underlying system layer* consists of the client runtime, the service runtime, and the IdP. These are complex modules that one typically does not understand in detail in the beginning of the study. Developing a semantic model for these components involves substantial systems investigation effort (as described in Section 4.3) because the seemingly clear SDK logic actually depends on a much more mysterious (and often incompletely documented) underlying layer. We consider the formal semantic models resulting from this study as one of the main contributions of this work.

The client and server application modules are *abstract modules*. They do not have concrete implementations: our goal is to reason about all possible apps built using the SDK. Nevertheless, the app modules do have constraints on their behaviors: *FooApp_C* and *FooApps* are only allowed to use the target SDKs for authentication

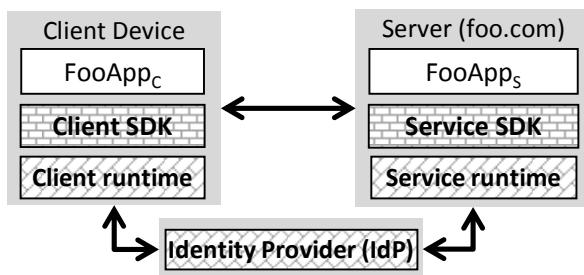


Figure 3. Modules in Client+Service App.

and authorization, and must not violate rules documented in the SDK developer guides.

3.2 Threat Model

We want to reason about security properties of *all* apps that could reasonably be constructed with the SDK. We assume a malicious application, *MalApp_C*, may be installed on the user’s device. *MalApp_C*’s behavior is not constrained by the client SDK, but it is limited to functionality provided by the client runtime (e.g., it cannot access cookies of other domains or handcraft HTTP requests). The attacker also controls an unconstrained external machine, which we call “Mallory”. As shown in Figure 4, we can think of Mallory as a combination of a client and server that can freely communicate with the client, application server, and IdP. We model *MalApp_C* and Mallory as abstract modules.

3.3 Security Properties

Our analysis depends on a formal definition of the security properties the SDK is intended to provide.

Granularity: session. Informally, people often say things like “a client is authenticated as Alice”, or “a server is authorized on Alice’s behalf”. However, it is important to point out explicitly that it is not the client or the server, but the *session* between them, that is authenticated or authorized. More specifically, the end result of an authentication/authorization protocol between a client and a server is to know whom *the session* represents and what *the session* is allowed to do. It should not affect the identity or permission of any other session. Therefore, we always keep the session (identified by its session ID) explicit in our modeling.

Basis of security: secrets and signed data. All mechanisms we study share a commonality: they use secrets or data signed by the identity provider as unforgeable evidence to differentiate some entities from others. These secrets and signed data are either preconfigured or generated at runtime at the underlying system layer.

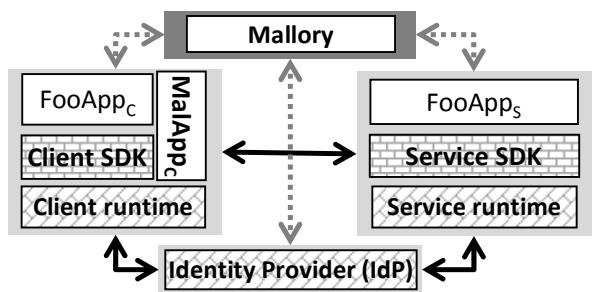


Figure 4. Threat Model.

We distinguish five types of secrets in the studied SDKs: access tokens, Codes², refresh tokens, app secrets and session IDs. The first four are protocol data in OAuth, which we will explain in later examples. The only identity-provider-signed data we have seen are *signed requests*, defined by Facebook, and *authentication tokens*, defined by Live ID. They are signed data structures containing some or all of the following data: access token, Code, app ID and user ID.

The desired security properties, therefore, need to consider what data the adversary may have obtained. This is made explicit by adding a *knowledge pool* to the model. All secrets and signed data received by the attacker are recorded in the knowledge pool, and can be used by the attacker in all subsequent actions.

Desired security properties. We define the security goal of the authentication/authorization SDKs based on the protections they provide to apps. Apps written using the SDK following explicit programming guidelines should be protected from the following violations:

(1) *Authentication violation.* If some knowledge, k , is about to be added to the pool, and k is sufficient to convince the authentication logic of FooApps that the knowledge holder is Alice, it implies that Mallory (and MalApp_C, since they share the knowledge pool) can authenticate as Alice, which is an authentication violation.

(2) *Authorization violation.* Depending on the type of k , there are two kinds of authorization violations. If k is Alice's access token, Alice's Code, or the session ID for the session between FooApp_C and FooApps, it implies that Mallory has obtained the permission to do whatever the session can do. Another authorization violation is when k is the app secret of FooApp. This would allow Mallory to do whatever FooApps can do on the identity provider.

(3) *Association violation.* The ultimate goal of authentication/authorization is not only to know who the user is or what she can do, but to correctly bind three pieces of data: the user's identity (i.e., the authentication result), the user's permissions (i.e., the authorization result), and the session's identity (usually known as session ID). This association is actually the end result of authentication/authorization and is what the application logic depends on after the process is accomplished. Mistakes in the association (such as binding Mallory's identity to Alice's permission, or binding Alice's identity to Mallory's session) are security violations.

² To avoid confusion with other meanings of “code”, such as “source code”, we always capitalize the first letter to refer to the “OAuth Code” in this paper.

3.4 The Process of Explicating SDKs

Figure 5 rearranges the modules (from Figure 4) and combines the concrete modules one each layer into one. The dashed line between abstract and concrete modules represents the interface between the test harness and the target system. The essential question is: *what assumptions are necessary for FooApp to achieve the desired security properties?*

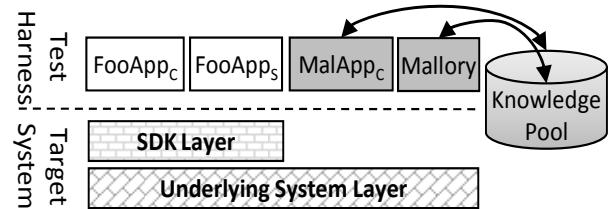


Figure 5. Modules Rearranged for Explicating.

Explicating SDKs is a systematic investigation effort to explicitly document our knowledge about these modules and examine the knowledge against defined security goals. As shown in Figure 6, it is an iterative process, in which we repeatedly refine our model and formally check if it is sufficient to establish the security properties or additional assumptions are needed. A failed check (i.e., a counterexample in the model) indicates either that our understanding of the actual systems needs to be revisited or that additional assumptions are needed to ensure the desired security properties.

The outcome of the process is the assumptions we explicitly added to the model. In Section 5.2, we show that many of the uncovered assumptions can indeed be violated in realistic situations.

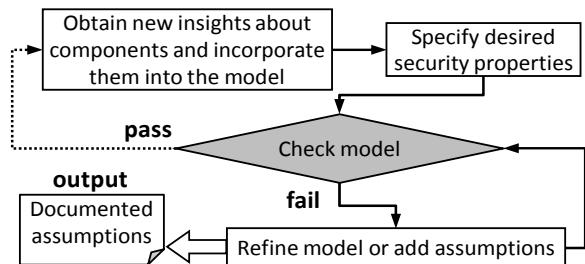


Figure 6. Engineering Process for Explicating SDKs.

4 Semantic Modeling

This section gives an overview of the semantic modeling effort for the three SDKs. The resulting models are available at <https://github.com/sdk-security/>. They reflect six months of effort by our two lead authors (i.e., 12 person-months) in creating and refining the system models.

4.1 Modeling language

To specify the semantics of the modules, we want a language that has a suitable formal analysis technology for verification. In the first period of our investigation, we used Corral [24], a property checking tool that can perform bounded verification on a C program with embedded assertions. Corral explores all possible execution paths within a bound to check if the assertions can be violated. Later, we re-implemented all the models in Boogie [9], a language for describing proof obligations that can then be tested using an SMT solver, which allowed us to fully prove the desired properties. This provides a higher assurance than the bounded verification done by Corral, but the basic ideas and approach are the same for both checking strategies. For concreteness, this section describes the Boogie version to explain our modeling.

The key Boogie language features needed to understand this paper are:

- The * symbol represents a non-deterministic Boolean value.
- HAVOC v is a statement that assigns a non-deterministic value to variable v .
- ASSERT(p) specifies an assertion that whenever the program gets to this line, p holds.
- ASSUME(p) instructs Boogie to assume that p holds whenever the program gets to this line.
- INVARIANT(p) specifies a loop invariant. Boogie checks if p is satisfied at the entry of the loop, and inductively prove p 's validity after each iteration.

If Boogie fails to prove an assertion or an invariant, it reports a counter-example. This leads us to refine the model, adding assumptions when necessary.

4.2 Modeling abstract modules

The test harness interacts with the concrete modules in a non-deterministic manner. It implements the abstract modules representing both the unknown (benign) application and the attacker's resources. The test harness consists of a loop with the loop count depth. Each iteration calls the function TestHarnessMakesCall. This function is implemented as a non-deterministic switch (i.e., a statement of “switch(*){...}”) that chooses to call FooApp_cRuns, MalApp_cMakesCall, or MalloryMakesCall. Eventually, through a series of non-deterministic choices as shown in Figure 7, one of the functions in a concrete module will be called.

Using the knowledge pool. As mentioned in Section 3.3, we use a knowledge pool to model the information

obtained by an attacker. Different types of knowledge, such as access tokens, Codes, and session IDs, are explicitly differentiated. We do not consider attacks that involve providing arguments of the incorrect type, e.g., giving a session ID to a function expecting an access token. There is an AddKnowledge function for each knowledge type. After each call to MalApp_cMakesCall and MalloryMakesCall, the function AddKnowledge_Type is called to add any acquired knowledge to the pool. There is a corresponding DrawKnowledge_Type function for non-deterministically drawing knowledge of a particular type from the knowledge pool. It is implemented using HAVOC i , where i is the array index of the piece of knowledge non-deterministically chosen.

4.3 Modeling concrete modules

Concrete modules do not have any non-determinism. The key aspects of building semantic models for the concrete modules are summarized below.

Data types. The basic data types in the models are int and several types for enumerables. We also define structs and arrays over the basic types. In the actual systems, the authentication logic is constructed using string operations such as concatenation, tokenization, equality comparison, and name/value pair parsing. We found that most string values are essentially enumerable, except those of domain names and user names, which we canonicalize as *Alice*, *Mallory*, *foo.com*, *mallory.com*, etc. Thus, the basic types, structs, and arrays are sufficient to model data used in the concrete modules.

SDKs. The sizes of these SDKs are moderate (all under 2000 lines) and their source code is public. The SDKs we modeled were implemented in HTML, JavaScript and PHP, so we needed to first translate the SDKs function-by-function into Boogie. We do this translation manually, but it is not hard to imagine tools that could mostly automate it. Table 1 shows two functions in the Facebook PHP SDK and our corresponding Boogie procedures. For getUserFromAvailableData, the changes are essentially line-by-line translations. For getLogoutUrl, the PHP code performs a string operation and re-

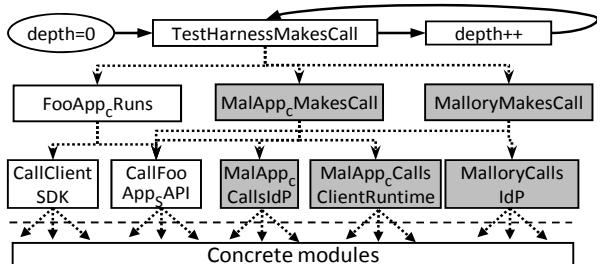


Figure 7. Test Harness.
(Dotted lines represent non-deterministic choices.)

```

protected function getUserFromAvailableData() {
    if ($signed_request) {
        ...
        $this->setPersistentData('user_id',
            $signed_request['user_id']);
        return 0;
    }
    $user = $this->getPersistentData('user_id', $default = 0);
    $persist_token =
        $this->getPersistentData('access_token');
    $access_token = $this->getAccessToken();
    if ($access_token &&
        !($user && $persist_token == $access_token)) {
        $user = $this->getUserFromAccessToken();
        if ($user)
            $this->setPersistentData('user_id', $user);
        else $this->clearAllPersistentData();
    }
    return $user;
}

public function getLogoutUrl() {
    return $this->getUrl(
        'www', 'logout.php',
        array_merge(array(
            'next' => $this->getCurrentUrl(),
            'access_token' => $this->getAccessToken(), ), ...));
}

```

```

procedure {:inline 1} getUserFromAvailableData() returns (user:User) {
    if (IdP_Signed_Request_Records__user_ID[signed_request] != _nobody) {
        ...
        user := IdP_Signed_Request_Records__user_ID[signed_request];
        call setPersistentData__user_id(user);
        return;
    }
    call user := getPersistentData__user_id();
    call persisted_access_token := getPersistentData__access_token();
    call access_token := getAccessToken();
    if (access_token >= 0 &&
        !(user != _nobody && persisted_access_token == access_token)) {
        call user := getUserFromAccessToken(access_token);
        if (user != _nobody) {
            call setPersistentData__user_id(user);
        } else {
            call clearAllPersistentData();
        }
    }
    return;
}
procedure {:inline 1} getLogoutUrl()
returns (API_id: API_ID, next__domain: Web_Domain, next__API: API_ID,
        access_token: int) {
    API_id := API_id_FBCConnectServer_login_php;
    call access_token := getAccessToken();
    call next__domain, next__API := getCurrentUrl();
}

```

Table 1. Example of a PHP function and its Boogie model.

turns a string. Our Boogie translation in this case is not obviously line-by-line. For example, our procedure returns a four-element vector instead of a string. The PHP function calls getUrl and array_merge, which concatenate substrings, therefore, are implicitly modeled by the four-element return vector.

Underlying system layer. Unlike the SDK, which is simple enough to model completely, the identity provider, client runtime, and server runtime are very complex and may not even have source code available. Completely modeling every detail of these systems is infeasible, but our analysis depends on developing suitable models of them. By studying the target SDKs, we identified three aspects of these systems that need to be carefully understood to perform verification. These aspects are the basis for the security goals the SDKs are designed to achieve:

(1) *The identity provider’s behaviors according to different input arguments and various app settings in its web portal.* Each identity provider has a web page for app developers to enter a number of app settings that the identity provider needs to know, such as app ID, app secret, service website domain, and return URL. Many of these settings are critical for the identity provider’s decision-making. Further, different inputs to the provided APIs cause different responses. Because we do not have the source code for the identity providers, we tested these behaviors by constructing different requests and app settings. For example, in the models we’ve built, the identity provider APIs dialog_

permissions_request(), RST2_srf() and oauth20_authorize_srf()³ involved 11, 8 and 6 if-statements respectively, to describe different behaviors we observed in testing.

(2) *Data passing on the client runtime.* As with the identity providers, we do not have access to source code to understand detailed behaviors of the client runtime. Our models were based on observations made during testing. We focused on the client’s decision-making about passing data from one server to another (by redirection), delivering data to FooAppc or MalAppc, and attaching cookies to outgoing requests. These decisions are important for security. We maintain a cookie structure for each client app, i.e., FooAppc or MalAppc. The cookie structure contains a session ID field and some optional fields specific to the SDK, such as signed_request and authentication_token.

(3) *Sessions, requests, and cookies on the service runtime.* In our model, the service runtime is a layer that defines data structures for sessions, requests and cookies for the service SDK and FooApps. (Note that although cookies are in the headers of requests, we separate them to flatten the data structure.) The cookie structure is the same as previously described. The request structure is defined according to the SDK’s specifica-

³ The APIs are accessed as <https://www.facebook.com/dialog/permissions.request>, <https://login.live.com/RST2.srf>, and https://login.live.com/oauth20_authorize.srf

tion. For example, requests for the Facebook PHP SDK use a structure containing a Code, a state and an optional *signed_request*. The session structure contains a session ID and a collection of session variables (keys) defined by the SDK.

4.4 Security assertions

We use ASSERT statements to document and test the desired security properties, covering each of the security violations described in Section 3.3.

Authentication violation. An authentication violation occurs when an attacker acquires some knowledge that could be used to convince FooApps that the knowledge holder is Alice. A simple example is the case we described in Section 2.2, in which the knowledge is an access token. In addition to access tokens, we also consider IdP-signed data such as Facebook’s signed messages or Live ID’s authentication tokens. To detect these violations, when a Facebook Signed Request k is added to the knowledge pool, we assert that

```
 $k.\text{user\_ID} \neq \text{_alice} \&& k.\text{app\_ID} \neq \text{_foo\_app\_ID} \&&$ 
 $\text{TokenRecordsOnIdP}[k.\text{token}].\text{user\_ID} \neq \text{_alice}$ 
```

where TokenRecordsOnIdP represents IdP’s database storing the records of access tokens.⁴

Authorization violation. To detect authorization violations, we add ASSERT statements inside each `AddKnowledge_Type` function. For example, the assertion in function `AddKnowledge_Code` is:

```
ASSERT(!(c.user_ID == _alice && c.app_ID == _foo_app_ID))
```

This checks that the Code added to the knowledge pool is not associated with Alice on FooApp. Similar assertions are added to the `AddKnowledge` functions for refresh tokens and session IDs. The app secret is different from the above knowledge types, because it is tied to the app not the user. When k is an app secret, we assert that $k \neq \text{_foo_app_secret}$.

Association violation. At the return point of every web API on FooAppS, we need to ensure the correct association of the user ID, the permission (represented by an access token or Code), and the session ID. For example, for Facebook PHP SDK, the assertion is the following. It This ensures that the three session variables of the session identified by `cookie.sessionID` all involve the same user. Concrete cases are given in Section 5.2.

```
Sessions[cookie.sessionID].user_ID ==
CodeRecordsOnIdP[ Sessions[
    cookie.sessionID].code].user_ID ==
&& Sessions[cookie.sessionID].user_ID ==
TokenRecordsOnIdP[Sessions[
    cookie.sessionID].token].user_ID
```

5 Results

We applied our approach to explicate the Facebook PHP SDK, Live Connect SDK and Windows 8 Authentication Broker. The Facebook PHP SDK is the only server-side SDK provided on Facebook’s developers’ website and is currently among the most widely used authentication/authorization SDKs. Facebook also has SDKs for Android and iOS apps, which have many concepts similar to the PHP SDK, but we have not studied them in detail. The Live Connect SDK is provided by Microsoft for developing metro apps that use Live ID as the identity provider. The Windows 8 Authentication Broker is for metro apps to use an OAuth-based (not only Live ID) identity provider, such as Facebook or Twitter.

5.1 Assumptions Explicated

The models resulting from our study formally capture what we learned about the SDKs and the systems. Our assumptions are specified in two ways: (1) all the ASSUME statements that we added; (2) when we need to assume particular program behaviors, such as a function call must always precede another, we model the behaviors accordingly, and add comment lines to state that the modeled behaviors are assumptions, rather than concrete facts. All the assumptions are added in order to satisfy the assertions that described in Section 4.4. The assertions are fairly uniform — they are all about sensitive data added to the knowledge pool and binding errors in associating sessions, users and permissions.

Verification. After all the assumptions were added, the models were automatically verified by Corral with the bound⁵ set to 5, meaning that in the test harness (Figure 7), the counter of the main loop (variable depth) does not exceed 5. Such a depth gives a reasonable confidence that the security properties are achieved by the models and the added assumptions: the properties could only be violated by attacks consisting of six or more steps. Running on a Windows server with two 2.67GHz processors and 32GB RAM, it took 11.0 hours to check the Facebook PHP SDK, 26.3 hours to check Live Connect SDK and 15.1 hours to check the Windows 8 Authentication Broker.

⁴ To improve presentation readability, the syntax of the above predicate is slightly changed from the syntax allowed by Boogie; see <https://github.com/sdk-security/> for the exact syntax.

⁵ Corral is a fully automatic tool for exploring code paths symbolically. The full automation, however, comes with the limitation that it only performs a bounded search.

Name ^a (SDK)	Assumption ^b	consequence of violation	exploit opportunity	vendor response
A1 (FB)	In <code>FooAppC MakesACall</code> , we assume <code>FooAppC.cookie.sessionID == _aliceSession</code> .	The ASSERT in Table 1 will be false. Mallory’s session is associated with Alice’s user ID.	When the SDK is used in subdomaining situations, e.g., cloud domains	Counter-measure on service platform
A2 (FB)	For any PHP page, if <code>getUser</code> is called, then <code>getAccessToken</code> must be called subsequently.	Alice’s user ID will be associated with Mallory’s access token.	When FooApps contains a PHP page that directly returns the user ID	SDK code fix
A3 (FB)	Before <code>getLogoutUrl</code> returns to client, we assume <code>logoutURL.params.access_token != getApplicationAccessToken()</code> .	App access token is added to the knowledge pool (owned by the adversary).	When a PHP page does not have the second code snippet shown in the dev guide	SDK code fix
A4 (LC)	In <code>saveRefreshToken</code> on FooApps, we assume <code>user._id != refresh_token.user._id</code> .	Alice’s refresh token will be associated with Mallory’s session on FooApps.	When the term “ <i>user id on the site</i> ” in the dev guide is interpreted as the user’s Live ID	Dev guide revision
A5 (WA)	In <code>callAuthenticateAsyncFromMalApp</code> , we assume <code>(app_id == _MalAppID user == _Mallory)</code> .	Alice’s access token or Code for FooApp is obtained by MalAppc.	When a client allows automatic login or one-click login	See Section 5.2.3
A6 (FB)	We assume FooAppC always logs in as Alice, i.e., the first argument of <code>dialog_oauth</code> is “_Alice”.	Alice’s session will be associated with Mallory’s user ID and access token.	When request forgery protection for app logon is missing or ineffective	Notifying developers

Table 2. Critical assumptions uncovered in our study.

^a FB stands for Facebook PHP SDK, LC for Live Connect and WA for Windows Auth Broker

^b Boogie syntax does not allow the dot operator to refer to a child element. For simplicity of presentation, we use it in this column.

The verification being bounded is a limitation of the models built for Corral, so we subsequently re-implemented all three models in Boogie language [9]. Verification of Boogie models is not automatic. It requires human effort to specify preconditions and post-conditions for procedures, as well as loop invariants (i.e., the invariant clauses). The Boogie verifier checks that (1) every precondition is satisfied by the caller; (2) if all preconditions of the procedure are satisfied, then all the postconditions will be satisfied when the procedure returns; (3) every loop invariant holds initially, and if it holds before an iteration then it will still hold after the iteration. By induction, the verified properties hold for an infinite number of iterations. Rewriting the three models in Boogie took 14 person-days of effort, including a significant portion on specifying appropriate loop invariants. The Boogie modeling did not find any serious case missed in the Corral modeling, but provides a higher level of confidence.

Examining the assumptions in the real world. We manually examined each assumption added to assess whether it could be violated in realistic exploits. This effort requires thinking about how apps may be deployed and executed in real-world situations. Table 2 summarizes the assumptions uncovered by our study that appear to be most critical. These assumptions can be violated in the real world, and the violations result in security compromises. Based on our experience in communicating with SDK providers, finding realistic

violating conditions is a crucial step to convincing them to treat the cases with high priority. This step requires extensive knowledge about systems, and does not appear to be easily automated. We describe these assumptions in more detail in Section 5.2. Table 3 lists some assumptions uncovered that, if violated, would also lead to security compromises. But, unlike the assumptions in Table 2, we have not found compelling realistic exploits that violate these assumptions. A few additional assumptions, listed in Appendix B, are needed to complete the verification. They correspond to some simplifications we made to the models. It is unclear if their violations lead to security compromises, but we make it explicit that we have not considered the situations violating these assumptions.

5.2 Confirmed Exploitable Assumptions

This subsection explains each of the critical assumptions in Table 2. These results show concretely how the SDK’s security assurance depends on actual system behaviors and app implementations, illustrating the importance of explicating the underlying assumptions upon which secure use of the SDK relies.

5.2.1 Facebook SDK

Assumptions A1, A2, A3, and A6 concern the Facebook PHP SDK.

name	assumption	consequence of violation	proposed fix
B1 (FB)	Result of getAccessToken returned to client is not equal to getApplicationAccessToken()	App access token is added to the knowledge pool.	Develop checker to examine the traffic from FooApps
B2 (FB)	In dialog_oauth, we assume FooApp.site_domain != Mallory_domain	Alice's access token or Code for FooApp is obtained by Mallory.	Develop checker to examine if the "Site Domain" app setting is properly set
B3 (FB)	Before FooApp sends a (non-NULL) request, we assume request.signed_request.userId == _Alice	Alice's session will be associated with Mallory's user ID and access token.	Enhance dev guide to require a runtime check on FooApp
B4 (LC)	In HandleTokenResponse, we assume auth_token.app_ID == _foo_app_ID	Alice's authentication token for MalApp will be used by Mallory to log into FooApps as Alice	Develop checker to examine if the signature in the auth_token is verified.
B5 (LC)	In constructRPCookiefromMallory, we assume (RP_Cookie.access_token.user_ID == RP_Cookie.authentication_token.user_ID)	Alice's ID associate with Mallory's access token, or vice versa	Enhance dev guide to require a runtime check on FooApps

Table 3. Assumptions uncovered that would lead to security vulnerabilities if violated but no realistic exploits known.

Assumption A1. This assumption states that the cookie associated with Alice's client must match Alice's session ID. Figure 8 is a screenshot of the usage instructions given in the *readme* file in the Facebook PHP SDK [17]. It seems straightforward to understand: the first code snippet calls `getUser` to get the logged-in user's ID (it returns null if the user is not logged in). The second snippet demonstrates how to make an API call, such as `me`. The third snippet toggles between login and logout, so that a logged-in user will get a *logoutURL* and a logged-out user will get a *loginURL* in the response.

The SDK's implementation for the `getUser` method is very simple. It calls the `getUserFromAvailableData`

Usage

The examples are a good place to start. The minimal you'll need to have is:

```
require 'facebook-php-sdk/src/facebook.php';
$facebook = new Facebook(array(
  'appId'  => 'YOUR APP ID',
  'secret' => 'YOUR APP SECRET',
));
// Get User ID
$user = $facebook->getUser();
```

To make API calls:

```
if ($user) {
  try {
    // Proceed knowing you have a logged in
    // user who's authenticated.
    $user_profile = $facebook->api('/me');
  } catch (FacebookApiException $e) {
    error_log($e);
    $user = null;
  }
}
```

Login or logout url will be needed depending on current user state.

```
if ($user) {
  $logoutUrl = $facebook->getLogoutUrl();
} else {
  $loginUrl = $facebook->getLoginUrl();
}
```

Figure 8. Facebook PHP SDK usage instructions.
(Screenshot from <https://github.com/facebook/facebook-php-sdk/blob/master/readme.md>)

function shown in Table 1. There are two statements (italicized in Table 1) calling `setPersistantData`, which is to set a PHP session variable denoted as `_SESSION['user_id']`. Setting `_SESSION['user_id']` is a binding operation because it associates the user's identity with the session, which may affect the predicate that we define against association violations — specifically, if Alice's user ID is assigned to the `_SESSION['user_id']` of Mallory's session, it would allow Mallory to act on FooApps as Alice. Because the session ID is a cookie in the HTTP request, the assertion must depend on how a client runtime handles cookies.

Violating the assumption using subdomaining.

Normally, because of the same-origin-policy of the client, cookies attached to one domain are not attached to another. However, the policy becomes interesting when we consider a cloud-hosting scenario. In fact, Facebook's developer portal makes it very easy to deploy the application server on Heroku, a cloud platform-as-a-service. Each service app runs in a subdomain under `herokuapp.com` (e.g., FooApps's sub-domain runs as `foo.herokuapp.com`). Of course, Mallory can similarly run a service as `mallory.herokuapp.com`.

The standard cookie policy for subdomains allows code on `mallory.herokuapp.com` to set a cookie for the parent domain `herokuapp.com`. When the client makes a request to `foo.herokuapp.com`, the cookie will also be attached to the request. Therefore, if Alice's client visits the site `mallory.herokuapp.com`, Mallory will be able to make the client's cookie hold Mallory's session ID. Thus, FooApps binds Alice's user ID to Mallory's session.

In response to our report, Facebook developed a countermeasure, which has been applied on the Heroku platform. It generates a new session ID (unknown to Mallory) when a client is authenticated. Facebook offered us a bounty three times the normal Bug Bounty amount for reporting this issue, as well as the same

award each for Assumptions A2 and A3 discussed next.⁶

Assumption A2. This assumption is a case in which Corral actually discovered a valid path for violating an assertion completely unexpected to us. The path indicated that if a PHP page on FooApps only calls `getUser` (e.g., only has the first code snippet from Figure 8), Mallory is able to bind her user ID to Alice’s session. The consequence is especially damaging if the session’s access token is still Alice’s. Corral precisely suggested the possibility (see Table 1): if there is a *signed_request* containing Mallory’s user ID, then the first `setPersistentData` call will be made, followed by a return. The method sets `_SESSION['user_id']` to Mallory’s ID without calling `getAccessToken`, which would otherwise keep the access token consistent with the user ID. Therefore, the association between the user ID and the access token is incorrect. The session will operate as Mallory’s account using Alice’s access token. After investigating our report about this, Facebook decided to add checking code before processing the signed request to the SDK to avoid the need for this assumption.

Assumption A3. This assumption requires that any PHP page that includes the third snippet in Figure 8 must also include the second snippet. In the example code in the figure, it is not obvious why the second snippet is required before the third snippet. However, when we modeled `getAccessToken`, as shown in Table 4, we realized that in Facebook’s authentication mechanism there are two subcategories of access token: *user access token*, which is basically what people usually refer to as “access token”, and *application access token*, which is described in Facebook’s dev guide [18]. The application access token is provided to a web service for a number of special purposes, such as “publishing instances of ‘secure Open Graph actions’”. In fact, the app secret can be derived solely from the application access token, so it is a serious authorization violation if Mallory or MalApp_C can obtain it.

Method `getLogoutUrl` in snippet 3 constructs a URL to send back to the client. The URL contains the result of `getAccessToken`. To obtain the application access token, Mallory only needs to send a request that hits a failure condition of `getUserAccessToken`, which prevents `$this->accessToken` from being overwritten in the bold line in Table 4. We confirmed that this can be done by using an invalid Code in the request.

⁶ We donated all three bounties to charities. The donations were one-to-one matched by Facebook.

```
public function getAccessToken() {
    ...
    $this->accessToken= $this->getApplicationAccessToken();
    $user_access_token = $this->getUserAccessToken();
    if ($user_access_token) {
        $this->accessToken=$user_access_token;
    }
    return $this->accessToken;
}
```

Table 4. SDK source code of `getAccessToken`

Interestingly, `getAccessToken` is also called by `getUser` in snippet 1 in Figure 8. If a PHP page includes snippet 2, the access token will be used to call a REST API. When it is an application access token, the API will raise an exception, which foils the exploit. That is why snippet 2 is required before snippet 3.

In response to our report on this issue, Facebook modified the SDK so that `getLogoutUrl` now calls `getUserAccessToken` instead of `getAccessToken`, thus avoiding the need for developers to satisfy this assumption.

Assumption A6. This assumption requires that the user on FooApp_C should not be Mallory. Otherwise, Mallory would be able to associate its access token and user id with Alice’s session. In Section 6.2, we show that many apps (14 out of 21 tested) indeed violate this assumption. Moreover, this association violation can be particularly damaging when the service app has its own credential system, and supports linking a Facebook ID to Alice’s password-protected account. Once the linking can be done in the session, Mallory will be able to sign into Alice’s account using Mallory’s Facebook ID. We confirmed that among the 14 service apps which violate the assumption, 6 of them support linking, and thus allow Mallory to login as Alice. We reported this issue to Facebook, who undertook the effort of notifying app and website developers.

5.2.2 Live Connect

Assumption A4 concerns how the Live Connect SDK handles “single sign-on for apps and websites” [27]. The sample `/LiveSDK/Samples/PHP/OAuthSample` [28] demonstrates how to implement a PHP service app that allows single sign-on. This sample code is essentially the dev guide given as a program skeleton, with comment blocks for app developers to implement. The core of the problem lies in the following function, whose implementation is empty except for a comment:

```
function saveRefreshToken($refreshToken) {
    // save the refresh token associated with the
    // user id on the site.
}
```

This is precisely what we call a *binding operation*. The refresh token is the input parameter, but it is not clear where the user id comes from. Within the scope of this

function, the only place to obtain a user ID is from a cookie called AUTHCOOKIE, which contains the user’s Live ID. However, the SDK’s logic is not sufficient to ensure that Alice’s refresh token is associated with her user ID. Appendix C of our technical report provides technical details [37].

We built a proof-of-concept exploit to send to Microsoft. The Live ID team responded that our attack is valid, but it “does not reflect the scenarios we are targeting”. The target scenario is a website which has its own credential system, such as a university website, so “the user id on the site” means, for example, the student ID. We replied to the team that an unclear context like this was exactly what we believe needs to be uncovered and at least documented clearly (indeed, explicating such assumptions is one of our main goals). In this case, the context was almost completely hidden: the *OAuthSample* sample is the only sample provided in */LiveSDK/Samples/PHP/*, so it is expected to target more generic scenarios. This is why if `saveRefreshToken` targets a specific scenario, the context must be made explicit. The team replied us that they would “*add more comments to that code to make the sample code clear on this.*” Recently we found that the comment has been revised to “*save the refresh token and associate it with the user identified by your site credential system.*” This change was also made in the ASP.NET version of the sample code.

5.2.3 Windows Authentication Broker

Assumption A5 concerns the Windows 8 Web Authentication Broker, used by Windows 8 apps with OAuth-based identity providers. For concreteness of presentation, we assume the Facebook Identity Provider. In the Auth Broker, the only function for authentication is `authenticateAsync`. Figure 9 illustrates the data passing through this function when the app requests an access token. The key observation is that the client does not conform to the same-origin policy, because the 302 response is in the context of *https://facebook.com*, while on Windows 8, an app runs in its own domain, *ms-appx://packageID*. Without the same-origin-policy, we were unable to see why Alice’s access token for FooApp is guaranteed to be passed to FooApp_C, not

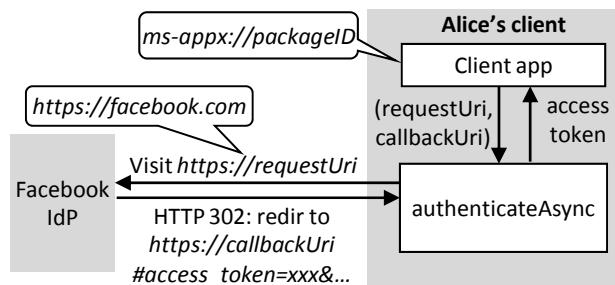


Figure 9. Data flow through `authenticateAsync`.

MalApp_C. To test this, we implemented a proof-of-concept MalApp_C. It indeed got the access token, which allowed it to do everything FooApp_C can do.

We reported this finding to Microsoft and Facebook, and learned their differing perspectives about the responsibility and severity of this issue. Microsoft considered it “*a shortcoming of the OAuth protocol and not specific to our implementation.*” Facebook pointed out that when `authenticateAsync` is called, an embedded browser window (usually called a *WebView*) is always prompted for Facebook password. This lowered the severity of the attack. We consider this a shaky security basis: if `authenticateAsync` someday allows a user to login automatically or with one click without using a password, the basis will become invalid.

We investigated how SDKs on other platforms handle the data passing, and found a similar issue with the Facebook SDK for Android. However, on Android, there is a mechanism to skip the password prompt to get the access token automatically. In response to our report, Facebook is developing a fix for its Android SDK.

6 Automated Testing

One additional value of explicating the SDKs is that it may be possible to provide tools that test apps for violations of critical assumptions. Such tests may not be able to guarantee the app always upholds the assumption, but rather focus on testing apps for common vulnerability patterns identified as a result of the explicating process. We developed a prototype to show the feasibility of building such a tester.

6.1 Design

Figure 10 shows our testing framework. For each vulnerability pattern to test, the test case defines the actions of the tester app, the proxy, and a set of server-side tester APIs (e.g., PHP or ASP.NET files). The tester app behaves as MalApp_C. The proxy does the necessary traffic manipulations for requests and responses. It also behaves as the unconstrained machine Mallory. Tester APIs implement specific checks for session states, especially for the associations we focus on.

We implemented test cases checking for violations of four assumptions: the vulnerability described in Section 2 (about using an access token for authentication), and vulnerabilities corresponding to the violations of assumptions A1 (concerning the session ID across sub-domains), A6 (about Mallory’s user ID associated with Alice’s session) and A4 (about binding the user ID with refresh token). Only the test for A4 requires a tester API on the app server.

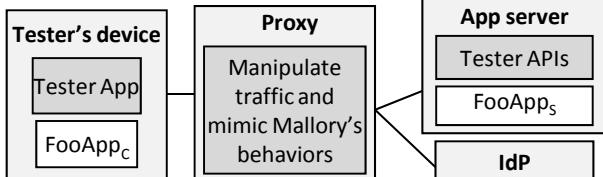


Figure 10. Testing Framework.
(Grey boxes constructed for testing.)

In the first test, the tester app performs the IdP’s sign-on steps as Alice, requests an access token, then presents the token to the app server to see if the authentication succeeded. In the second test (regarding A1), if the app server’s hostname is *foo.a.com*, the proxy creates another hostname *mallory.a.com*. The test follows the steps described in Section 5.2.1. Eventually the proxy checks if the authentication is successful, but the associated session ID is identical to that of Mallory’s session on *foo.a.com*. In the third test (for A6), the proxy observes the HTTP request that FooApp_c sends to Facebook. It finds out which type of data is used as the proof for authentication (a.k.a., the *authenticator*), which can be either a Code or signed request. The proxy also tries to find a field named state, which is an argument supported by Facebook to prevent request forgery for login [16]. The proxy then replaces the authenticator and the state field (if it exists) with the ones that Mallory’s session owns. After sending the request, the proxy checks whether Mallory can associate her Facebook ID with Alice’s session, and reports a violation if it sees a successful server response.

The fourth test (A4) requires the help of a tester API on the server because it tests whether the refresh token is associated with an appropriate user ID. The test uses the proxy to manipulate the AUTHCOOKIE in the request header so that it contains Mallory’s authentication token in Alice’s request. The proxy then mimics Mallory to call the tester API, which calls readRefreshToken and checks if it returns Alice’s refresh token.

6.2 Results

In general, the testing framework is designed for app developers so that they can avert the common pitfalls in their own implementations. Nevertheless, since some of the tests do not need tester APIs on the server, they can be used with access to the apps alone. This opens the possibility of a third party (such as the SDK provider) performing the tests on submitted apps.

We tried using the tests to check Windows 8 and Facebook apps found in the wild. The sets of apps that we tested are named Set 1, Set 2 and Set 3, corresponding to the first three aforementioned tests respectively. The test apps were obtained as objectively as possible.

Test Set	Number of Apps	Vulnerable
1 (Section 2)	27	21 (78%)
2 (assumption A1)	7	6 (86%)
3 (assumption A6)	21	14 (67%)

Table 5. Test Results.

To construct set 1, we queried “Facebook” in the free apps in Windows 8 App Store, which returned about 572 apps. We ranked the apps by user ratings and examined the apps with a rating of 3+ stars. Apps without a backend service were excluded. We then selected apps that authenticate users through identity providers. This left us with a total of 27 apps.

Set 2 was constructed by doing a Google query for “herokuapp.com login”, which gave us many URLs on *herokuapp.com*. We visited each URL to see if the website ran a PHP server and appeared reasonably functional. This gave us a list of 20 websites. We then examined the traffic of each website to determine if it used the Facebook PHP SDK. Seven of the sites did, and these were used for Set 2.

To construct Set 3, we used the Google search query “login.php” and visited the first 40 result pages⁷ to examine which URLs correspond to PHP websites that support Facebook sign-on. We found 21 candidate websites that comprise Set 3.

Table 5 shows the number and percentage of apps that matched the vulnerability pattern in each set. The results for Set 1 show that 78% of tested services with Facebook sign-on mechanism indeed use the access token for server-side authentication. The results for Set 2 reinforce the value of our SDK analysis — when we studied the SDK, we only hypothesized the possibility of this vulnerability. The vulnerability we conceived on a hypothetical service app (FooApp_s) accurately reflects the reality of 86% of services tested in Set 2. The results for Set 3 indicate that 67% of the tested apps would allow Mallory’s Facebook ID to be associated with Alice’s session. This violation is mainly due to missing or insufficient request forgery protections for user login. This association mistake can be particularly dangerous when the service apps support certain functionalities. For example, we found that many service apps have their own credential systems, and allow a user to link her Facebook ID to her password-protected account. After the linking, the user can use a Facebook login to sign into the password-protected account. When assumption A6 is violated, Mallory is able to link her Facebook ID to Alice’s account in the

⁷ We needed to examine so many result pages because most webpages matched the query “login.php” for reasons not about our intent, e.g., popular pages containing both words “login” and “php” are often considered a match.

session, and thus able to sign into Alice’s account. We confirmed that 6 of the service apps could be exploited in this way.

7 Related Work

The idea of formally verifying properties of software systems goes back to Alan Turing [34], although it only recently became possible to automatically verify interesting properties of complex, large scale systems. Our work makes use of considerable advances in model checking that have enabled model checkers to work effectively on models as complex as the ones we use here. Our work is most closely related to other work on inferring and verifying properties of interfaces such as APIs and SDKs, which we review briefly next.

API and SDK misuses. It is no longer a mystery that APIs and SDKs can be misunderstood and the results often include security problems. On various UNIX systems, setuid and other related system calls are non-trivial for programmers to understand. Chen et al. “demystified” (that is, explicated) these functions by comparing them on different UNIX versions and formally modeling these system calls as transitions in finite state automata [11]. Wang et al. showed logic bugs in how websites integrate third-party cashier services and single-sign-on services [35][36]. Many of the bugs found appear to result from website developers’ confusions about API usage. Georgiev et al. showed that SSL certificate validations in many non-browser applications are broken, which make the applications vulnerable to network man-in-the-middle attacks [19]. Our work started from a different perspective — our primary goal is not to show that SDKs can be misused, but to argue that these misuses are so reasonable that it is SDK providers’ lapse not to explicate the SDKs to make their assumptions clear. We expect that our approach could be adapted to other contexts such as third-party payment and SSL certificate validation.

Interface Verification. Many researchers have considered issues related to verifying interfaces and their use. Spinellis and Louridas [32] propose a static analysis framework for verifying Java API calls. Library developers are required to write imperative checking code for each API to assist the verification process. Henzinger et al. [1][7] propose languages and tools to help model the interfaces and find assumptions that need to be met for two APIs to be *compatible*, i.e., there is no environment for which they reach an error state. JIST [2] uses a similar approach to synthesize interface specifications for Java classes. This line of work is complementary to ours. Our main effort has been to systematically understand systems and construct semantic models. Currently, we manually add assump-

tions when counterexamples are found in the models. The assumptions could be considered as a type of “interface specifications” of the SDKs. We believe that our semantic models would be even more valuable with tools that can automatically synthesize high-quality assumptions.

Software testing. Static techniques such as the Static Driver Verifier (SDV) for Windows drivers [4] and dynamic analysis such as symbolic execution [3][12] and fuzz testing [13][20] are widely studied in software testing community. To test websites’ of single-sign-on authentications, Bai et al. developed AUTHSCAN [5], which is a technology to automatically recover an authentication protocol from concrete website implementations.

OAuth Protocol analyses. Bansal et al. [6] modeled OAuth 2.0 protocol and verified it using ProVerif [8]. They also built a library for future researchers to model web APIs into ProVerif language more easily. Pai et al. [31] used Alloy framework [23] to verify OAuth 2.0 and discovered a previously known vulnerability. Sun et al. discussed a number of website problems affecting OAuth’s effectiveness, such as not using HTTPS, having XSS and CSRF bugs [33]. Although the three SDKs we studied are based on OAuth, our work does not focus particularly on the OAuth protocol. The fact that all three studied SDKs are based on OAuth is mainly because of its widespread adoption, but the security issues we found concern the SDKs and services rather than flaws inherent in the OAuth protocol.

8 Final Remarks

Security exploits nearly always stem from attackers finding ways to violate assumptions system implementers relied upon. Such assumptions are often not carefully documented, and often only implicit in the minds of the system designers. Our study of three important authentication and authorization SDKs supports the need for systematically explicating SDKs to uncover these assumptions. We advocate that a systematic explication process should be part of the engineering process for developing SDKs. Although our current process still requires considerable manual effort in understanding and modeling system behaviors, we believe the need for this effort reveals flaws in the current engineering processes: SDK developers, including those building widely-used security-focused SDKs, have not systematically understood or documented the SDKs’ behaviors for producing secure applications. In our study, we found assumptions that were critical to secure use of the SDKs, but that were not clearly documented and were subtle enough to be missed by the majority of tested apps.

Acknowledgments

We thank Martín Abadi, Longze Chen, Cormac Herley, Kevin Sullivan, Helen Wang, Yi-Min Wang, and Westley Weimer for valuable comments on this work and an early draft of the paper. Mike Barnett offered great advice on building and checking the semantic models. We also appreciate the technical help from Eric Lawrence and David Ross about recording Live ID traffic on Windows 8. This work was partly funded by grants from the National Science Foundation and Air Force Office of Scientific Research (but does not necessarily reflect the views of the US Government). Yuchen Zhou was also supported in part by a Microsoft Research internship.

Availability

The Boogie models of the three studied SDKs are available at <https://github.com/sdk-security/>.

References

- [1] Luca de Alfaro and Thomas A. Henzinger. Interface Automata. In *8th European Software Engineering Conference* (held jointly with 9th ACM SIGSOFT International Symposium on Foundations of Software Engineering), 2001.
- [2] Rajeev Alur, Pavol Cerny, P. Madhusudan, and Wonhong Nam. Synthesis of Interface Specifications for Java Classes. In *32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (POPL), 2005.
- [3] Thanassis Avgerinos, Sang Kil Cha, Brent Lim Tze Hao, and David Brumley. AEG: Automatic Exploit Generation. In *Network and Distributed System Security Symposium* (NDSS). February 2011.
- [4] Tom Ball, Ella Bounimova, Byron Cook, Vladimir Levin, Jakob Lichtenberg, Con McGarvey, Bohus Ondrusk, Sriram K. Rajamani, and Abdullah Ustuner. Thorough Static Analysis of Device Drivers. *EuroSys*. 2006.
- [5] Guangdong Bai, Jike Lei, Guozhu Meng, Sai Sathyanarayanan Venkatraman, Prateek Saxena, Jun Sun, Yang Liu, and Jin Song Dong. AUTHSCAN: Automatic Extraction of Web Authentication Protocols from Implementations. In *Network and Distributed System Security Symposium* (NDSS). February 2013.
- [6] Chetan Bansal, Karthikeyan Bhargavan and Sergio Maffeis. Discovering Concrete Attacks on Website Authorization by Formal Analysis. *IEEE Computer Security Foundations* (CSF). 2012.
- [7] Dirk Beyer, Arindam Chakrabarti, and Thomas A. Henzinger. Web Service Interfaces. In *14th International Conference on World Wide Web* (WWW). 2005.
- [8] Bruno Blanchet. An Efficient Cryptographic Protocol Verifier Based on Prolog Rules. In *14th IEEE Computer Security Foundations Workshop* (CSFW), 2001.
- [9] Boogie: An Intermediate Verification Language. <http://research.microsoft.com/en-us/projects/boogie/>
- [10] John Bradley. *The Problem with OAuth for Authentication*. <http://www.thread-safe.com/2012/01/problem-with-oauth-for-authentication.html>
- [11] Hao Chen, David Wagner and Drew Dean. Setuid Demystified. *USENIX Security Symposium*. 2002
- [12] Chia Yuan Cho, Domagoj Babic, Pongsin Poosankam, Kevin Zhijie Chen, Edward XueJun Wu, and Dawn Song. MACE: Model-inference-assisted Concolic Exploration for Protocol and Vulnerability Discovery. In *20th USENIX Security Symposium*. 2011.
- [13] Adam Doupé, Ludovico Cavedon, Christopher Kruegel, and Giovanni Vigna. Enemy of the State: a State-aware Black-box Web Vulnerability Scanner. In *21st USENIX Security Symposium*. 2012.
- [14] Facebook. "The Facebook bounty program," <http://www.facebook.com/whitehat/bounty/>
- [15] Facebook. *SDK Reference - Facebook SDK for PHP*. <http://developers.facebook.com/docs/reference/php/>
- [16] Facebook. OAuth Dialog, <https://developers.facebook.com/docs/reference/dialogs/oauth/>
- [17] Facebook. *PHP SDK Usage*. <https://github.com/facebook/facebook-php-sdk/blob/master/readme.md>
- [18] Facebook. *Using App Access Tokens*. <http://developers.facebook.com/docs/opengraph/using-app-tokens/>
- [19] Matin Georgiev, Subodh Iyengar, Suman Jana, Rishita Anubhai, Dan Boneh, Vitaly Shmatikov. The Most Dangerous Code in the World: Validating SSL Certificates in Non-Browser Software. *ACM CCS*. 2012.
- [20] Patrice Godefroid, Michael Y. Levin and David Molnar. Automated Whitebox Fuzz Testing. *Network and Distributed System Security Symposium*. 2008
- [21] Google. *Using OAuth 2.0 to Access Google APIs*. <https://developers.google.com/accounts/docs/OAuth2>
- [22] Dick Hardt. *The OAuth 2.0 Authorization Framework (RFC6749)*. <http://tools.ietf.org/html/rfc6749>
- [23] Daniel Jackson. *Alloy: A Language and Tool for Relational Models*. <http://alloy.mit.edu/alloy/index.html>
- [24] Akash Lal, Shaz Qadeer, and Shuvendu Lahiri. Corral: A Solver for Reachability Modulo Theories. *Computer Aided Verification* (CAV). 2012
- [25] Microsoft. *Live Connect Developer Center – Metro Style Apps*. <http://msdn.microsoft.com/en-us/library/live/hh826551.aspx>
- [26] Microsoft. *Live SDK developer guide – Signing users in*. <http://msdn.microsoft.com/en-us/library/live/hh243641#signin>
- [27] Microsoft. *Live SDK Developer Guide – Single Sign-on for Apps and Websites*. <http://msdn.microsoft.com/en-us/library/live/hh826544.aspx>
- [28] Microsoft. "LiveSDK's OAuth Sample Code in PHP," <https://github.com/liveservices/LiveSDK/tree/master/Samples/PHP/OauthSample>
- [29] Microsoft. *Live Connect Developer Center – REST Reference*. <http://msdn.microsoft.com/en-us/library/live/hh243648.aspx>
- [30] Microsoft. *Windows.Security.Authentication.Web namespace*, <http://msdn.microsoft.com/library/windows/apps/BR227044>
- [31] S. Pai, Y. Sharma, S. Kumar, R.M. Pai, and S. Singh. Formal Verification of OAuth 2.0 Using Alloy Framework. In *Communication Systems and Network Technologies* (CSNT). 2011.

- [32] Diomidis Spinellis and Panagiotis Louridas. A Framework for the Static Verification of API Calls. *Journal of Systems and Software*. July 2007.
- [33] San-Tsai Sun and Konstantin Beznosov. The Devil is in the (Implementation) Details: An Empirical Analysis of OAuth SSO Systems. *ACM CCS*. 2012.
- [34] Alan Turing, “Checking a Large Routine”, presented at EDSAC Inaugural Conference, 1949. (available from <http://www.turingarchive.org/browse.php/B/8>)
- [35] Rui Wang, Shuo Chen, XiaoFeng Wang, and Shaz Qadeer. How to Shop for Free Online – Security Analysis of Cashier-as-a-Service Based Web Stores. *IEEE Symposium on Security and Privacy*. 2011
- [36] Rui Wang, Shuo Chen, XiaoFeng Wang. Signing Me onto Your Accounts through Facebook and Google: a Traffic-Guided Security Study of Commercially Deployed Single-Sign-On Web Services. *IEEE Symposium on Security and Privacy*. 2012
- [37] Rui Wang, Yuchen Zhou, Shuo Chen, Shaz Qadeer, David Evans, Yuri Gurevich. Explicating SDKs: Uncovering Assumptions Underlying Secure Authentication and Authorization. *Microsoft Research Technical Report MSR-TR-2013-37*, March 2013

Appendix A. Prevalence of SDKs.

To understand how widely-used different SDKs are we first searched for keyword “Facebook” in the Windows App Store and filtered the results by selecting free and trial apps only, which left us with a total of 572 apps. We then sorted the results by users’ rating, after which we went through the top of the list one by one to check if the app has Facebook or Live connect SSO built-in. We also monitored network traffic using Fiddler on those apps that have SSO feature, and this allows us to eliminate the ones that do not run an online service. We excluded non-English apps and also apps that do not work properly. After the selection process we came up with a total of 27 apps as listed below:

App Name	SDK(s)
<i>Soluto</i>	WA(FB)
<i>Givit</i>	Unknown
<i>Fliptoast</i>	WA(FB)
<i>Donelo</i>	Unknown
<i>IM+</i>	WA(FB)/LC
<i>Interference</i>	Live
<i>Norton Satellite</i>	Unknown
<i>Slide Ur buddy</i>	WA(FB)
<i>EuroCup</i>	Unknown
<i>Shufflr</i>	WA(FB)
<i>Social Umami</i>	Unknown
<i>SumAttack</i>	WA(FB)
<i>Guess Who</i>	WA(FB)
<i>Flixpicks</i>	WA(FB)/LC

<i>TwentyOne</i>	Unknown
<i>Apyo</i>	Unknown
<i>Where's my stuff</i>	Unknown
<i>Mahjong 31</i>	Unknown
<i>Tic Challenge</i>	WA(FB)
<i>Color orbs</i>	Unknown
<i>tagmap</i>	WA(FB)
<i>word gap</i>	LC
<i>word town</i>	Unknown
<i>noots</i>	Unknown
<i>RecipeHouse</i>	WA(FB)
<i>Alaska Airlines</i>	Unknown
<i>Captain Dash</i>	LC

WA(FB): Windows Auth Broker using Facebook IdP

LC: Live Connect

Unknown: We could not identify the observed authentication traffic.

Appendix B. Additional Assumptions.

The following assumptions were needed to complete the verification, but not included in Table 2 or Table 3 since they do not appear to have any likely security consequences.

C1: (Live Connect)

There are two sets of Live Connect APIs, one of Microsoft apps and services, such as Skydrive, the other for non-Microsoft apps and services. We assume the two sets of APIs cannot be called together, i.e., any sequence of calling these APIs is confined to only one of the two sets.

C2: (Live Connect, Windows Auth Broker)

We assume no possibility of executing a script provided by Mallory/MalAppc inside FooAppc. (Actually, we are concerned that DOM methods like InvokeScript and ScriptNotify may violate this assumption, but have not yet identified a clear security issue.)

C3: (all)

As explained in Section 4.2, we assume that access token, Code, authentication token, app secret, app ID, user ID, session ID and so on are of different types, although in reality they are all strings. We do not allow type mismatches.

Enabling Fine-Grained Permissions for Augmented Reality Applications With Recognizers

Suman Jana¹, David Molnar², Alexander Moshchuk², Alan Dunn¹, Benjamin Livshits², Helen J. Wang², and Eyal Ofek²

¹University of Texas at Austin

²Microsoft Research

Abstract

Augmented reality (AR) applications sense the environment, then render virtual objects on human senses. Examples include smartphone applications that annotate storefronts with reviews and XBox Kinect games that show “avatars” mimicking human movements. No current OS has special support for such applications. As a result, permissions for AR applications are necessarily *coarse-grained*: applications must ask for access to raw sensor feeds, such as video and audio. These raw feeds expose significant additional information beyond what applications need, including sensitive information such as the user’s location, face, or surroundings.

Instead of exposing raw sensor data to applications directly, we introduce a new OS abstraction: the *recognizer*. A recognizer takes raw sensor data as input and exposes higher-level objects, such as a skeleton or a face, to applications. We propose a *fine-grained* permission system where applications request permissions at the granularity of recognizer objects. We analyze 87 shipping AR applications and find that a set of four *core recognizers* covers almost all current apps. We also introduce *privacy goggles*, a visualization of sensitive data exposed to an application. Surveys of 962 people establish a clear “privacy ordering” over recognizers and demonstrate that privacy goggles are effective at communicating application capabilities. We build a prototype on Windows that exposes nine recognizers to applications, including the Kinect skeleton tracker. Our prototype incurs negligible overhead for single applications, while improving performance of concurrent applications and enabling secure offloading of heavyweight recognizer computation.

1 Introduction

An *augmented reality* (AR) application takes natural user interactions (such as gestures, voice, and eye gaze) as input and overlays digital content on top of the real world seen, heard, and experienced by the user. For example, on mobile phones, augmented reality “browsers” such as Layar and Juniaio allow users to look through the phone and see annotations about a magazine article or a storefront. Furniture applications on the iPad allow users to preview what a couch would look like in the context of a real room before buying [17]. The Xbox Kinect has sold over 19 million units and allows application developers to overlay avatars on top of a user’s pose, creating new kinds of games and natural user interfaces. Microsoft has released a Windows SDK for Kinect and helped incubate multiple startup companies delivering AR experiences on the PC. Even heads-up displays, previously restricted to academic and limited military/industrial use, are set to reach consumers with Google Glass [25].

Today’s AR applications are monolithic. The application itself performs sensing, rendering, and user input interpretation (e.g., for gestures), aided by user-space libraries, such as the Kinect SDK, OpenCV [6, 12], or cloud object recognition services, such as Lambda Labs or IQ Engines. Because today’s OSes are built without AR applications in mind, they offer only *coarse-grained* access to sensor streams, such as video or audio data. This raises a privacy challenge: it is difficult to build applications that follow the principle of *least privilege*, having access to only the information they need and no more. Today’s systems also do not have any AR-specific permissions, relying instead on careful pre-publication vetting of applications [5].

Motivating Example. Figure 1 illustrates the problem with coarse-grained abstractions in today’s

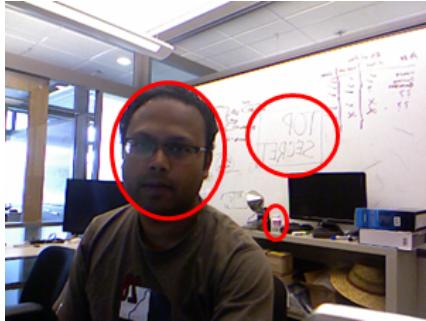


Figure 1: Giving raw sensor data to applications can compromise user privacy. This video frame captured from a Kinect contains the user’s face, private whiteboard drawings, and a bottle of medicine.



Figure 2: AR applications often need only specific objects rather than the entire sensor streams. The “Kinect Adventures!” game only needs body position to render an avatar and simulate game physics.

AR applications. The figure shows a video frame captured from a camera. Today, applications must ask for raw camera access if they want to do video-based AR, which means the application will see all sensitive information in the frame. In this frame, that information includes the user’s face, (private) drawings on the whiteboard, and a bottle of medicine with a label that reveals a medical condition.

An application, however, may not need *any* of this sensitive information to do its job. For example, Figure 2 shows a screenshot from the “Kinect Adventures!” game that ships with the Microsoft Xbox Kinect. First, the game estimates the body position of the player from the video and depth stream of the Kinect. Next, the game overlays an avatar on top of the player’s body position. Finally the game simulates interaction between the avatar and a virtual world, including a ball that bounces back and forth to hit blocks. To do its job, the game needs *only* body position, and not any other information from the video and depth stream.

Kinect is just one example of an AR system; this



Figure 3: Two examples of mobile AR applications that only need specific objects in a sensor stream. On the left, Macy’s Believe-O-Magic only needs the location in the frame of a special marker, on top of which it renders a cartoon character. On the right, Layar only needs to know the GPS location and compass position to show geo-tagged tweets.

Application	Objects recognized
Your Shape 2012	skeleton, person texture
Dance Central 3	skeleton, person texture
Nike+Kinect	skeleton, person texture
Just Dance 4	skeleton, video clip
NBA 2K13	voice commands
Xbox Dashboard	pointer, voice commands
Layar	GPS “points of interest”
Red Bull Racing	Red Bull Cans
Macy’s Believe-O-Magic	Macy’s store display

Figure 4: Sample AR applications and the objects they recognize. Kinect apps are above the line, mobile below.

principle of AR applications benefiting from “least privilege” is more general. We show two mobile phone examples in Figure 3. On the left, the Macy’s Believe-O-Magic application shows a view of a child standing next to a holiday-themed cartoon character. While the application today must ask for raw video access, which includes the face of the child and of all bystanders, the only information the application needs is the location of a special marker to enable rendering the cartoon in the correct place. On the right, Layar is an “AR browser” for mobile phones, here showing a visualization of where recent tweets have originated near the user. Again, Layar must ask for raw video and location access, but in fact it needs to only know the GPS position of the tweet relative to the user.

Beyond these examples, Figure 4 shows the top 5 Amazon best-selling Kinect-enabled applications for the Xbox 360, along with the Xbox Dashboard and representative AR apps on mobile phones. For each application, as well as the Xbox Dashboard, we enumerate the objects recognized; in Section 5 we carry out a similar analysis for all shipping Xbox Kinect applications. *None* of these applications need con-

tinuous access to raw video and depth data, but no current OS allows a user to restrict access at finer granularity.

The Recognizer Abstraction. To address this problem, we introduce a new least-privilege OS abstraction called a *recognizer*. A recognizer takes as input a sensor stream and creates events when objects are recognized. These events contain information about the recognized object, such as its position in the video frame, but not the raw sensor information. By making access to recognizer-exposed objects a first-class permission in an operating system, we enable least privilege for AR applications. We assume a fixed set of system-provided recognizers in this work. This is justified by our analysis of over 87 shipping applications, which shows a set of four “core recognizers” is sufficient for the vast majority of such applications (Section 5).

Supporting recognizers in the OS incurs several benefits. Besides enabling least privilege, recognizers lead to a *performance improvement*, as heavyweight object recognition can be shared among multiple applications. We show how an OS can compose recognizers in a *dataflow graph*, which enables precise reasoning about which recognizers should be run, depending on the set of running applications. Finally, we show how making dataflow explicit allows us to prune spurious permission requests. These benefits extend beyond AR applications and to any set of applications that must interpret higher-level objects from raw sensor data, such as building monitoring, stored video analysis, and health monitoring.

Challenges. We faced several challenges designing our recognizer-based AR platform. First, other fine-grained permission systems, such as Android, have been shown to be difficult to interpret for users [11]. To address this problem, we introduce *privacy goggles*: an “application’s-eye view” of the world that shows users which recognizers are available to an application. Users see a video representation of sensitive data that will be shown to the application (Figure 9). This, in turn, lays the foundation for informed permission granting or permission revocation. Our surveys of 462 people show that privacy goggles are effective at communicating capabilities to users.

Another challenge concerned *recognizer errors*. For example, an application may have permission for a skeleton recognizer. If that recognizer mistakenly finds a skeleton in a frame, the application may obtain information even though there is no person present. This information leakage violates a user’s expectations, even though the application sees only a higher-level object such as the skeleton.

We address recognizer errors with a new OS component, *recognizer error correction*. We evaluate three approaches: *blurring*, *frame subtraction*, and *recognizer combination*. The first two manipulate raw sensor data to reduce false positives in a recognizer-independent way. The last reduces false positives by using context information available to the OS from its use of multiple recognizers that could not be available to any individual recognizer author. We show that our techniques reduce false positives across a set of seven recognizers implemented in the OpenCV library [12].

Our final challenge concerned recognizers that require heavyweight object recognition algorithms which may run poorly or not at all on performance-constrained mobile devices [23, 21]. We thus build and evaluate support for *offloading* of particularly heavyweight recognizers to a remote machine.

We have implemented a prototype of our system on Windows, using the Kinect for Windows SDK. Our system includes nine recognizers, including face detection, skeleton detection, and a “plane recognizer” built on top of KinectFusion [23].

Contributions. We make the following contributions:

- We introduce a new OS abstraction, the *recognizer*, which captures the core object recognition capabilities of AR applications. Our novel fine-grained permission system for recognizers enables least privilege for AR applications. We show that all shipping Kinect applications would benefit from least privilege. Based on surveys of 500 people, we determine a privacy ordering on common recognizers.
- We introduce a novel visualization of sensitive data provided to AR applications, which we call *privacy goggles*. Privacy goggles let users inspect sensitive information flowing to an application, to aid in permission granting, inspection, and revocation. Our surveys of 462 people show that privacy goggles are effective at communicating capabilities to users.
- We recognize the problem of granting permissions in the presence of object recognition errors and propose techniques to mitigate it.
- We demonstrate that raising the level of abstraction to the “recognizer” enables the OS to offer services such as *offloading* and *cross-application recognizer sharing* that improve performance. Our implementation has negligible

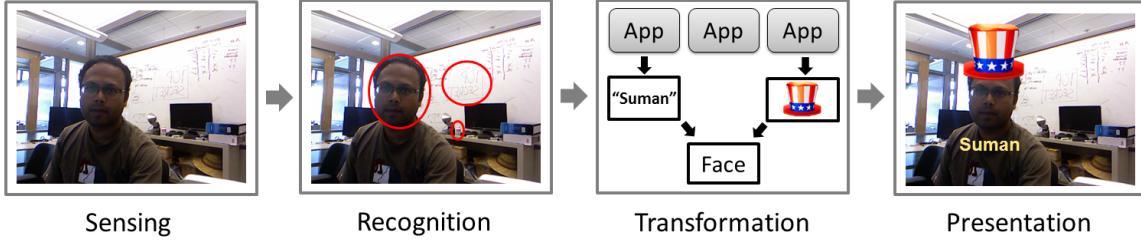


Figure 5: AR application pipeline: (1) reading raw data from hardware, (2) parsing raw data into recognized objects, (3) manipulating these objects to add augmentations to the scene, and (4) resolving conflicts and rendering.

overhead for single applications, yet greatly increases performance for concurrent applications and allows the OS to offload heavyweight recognizer computation.

In the rest of the paper, Section 2 provides background on AR, Section 3 discusses our recognizer abstraction, and Section 4 describes our implementation. Section 5 evaluates privacy goggles, recognizers required for shipping AR applications, recognizer error correction, and performance of our prototype. Sections 6 and 7 present related and future work, and Section 8 concludes.

2 AR Overview

We characterize AR applications using a pipeline shown in Figure 5. First, the *sensing stage* acquires raw video, audio, and other sensor data from platform hardware. In the figure, we show an RGB video frame as an example. The frame was captured from a Kinect, which also exposes a depth stream and a high-quality microphone.

Next, the *recognition stage* applies object recognition algorithms to the raw sensor data. For example, voice recognition may run on audio to look for specific keywords spoken, or a skeleton detector may estimate the presence and pose of a human in the video. As new advances in computer vision and machine learning make it possible to reliably recognize different objects, the resulting algorithms can be added to this stage. The code performing object recognition is similar to drivers in traditional operating systems: code running with high privilege maintains an abstraction between “bare sensing” and applications. Just as with devices in traditional OSes, an OS with support for AR could multiplex applications across multiple object recognition components; we will describe a new OS abstraction that enables this in the next section. In the figure, a face and two areas of text are recognized, one on the whiteboard and another on a bottle of medicine. The output of the recognition stage is a set of *software objects* that

“mirror” recognized real-world objects.

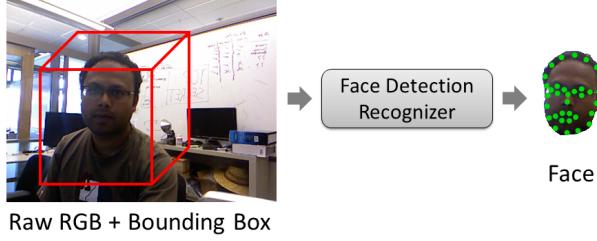
In the *transformation stage*, applications consume the recognized objects and add virtual objects of their own. Finally, the *presentation stage* creates the final view for the user, taking as input all current software objects and the current state of the world. This stage must resolve any remaining logical conflicts, as well as check that desired placement of objects is feasible. Today, this rendering is done using standard OS abstractions, such as DirectX or OpenGL.

3 The Recognizer OS Abstraction

We propose a new OS abstraction called a *recognizer*. A recognizer is an OS component that takes a sensor stream, such as video or audio, and “recognizes” objects in the sensor stream. For example, Figure 6 shows a recognizer that wraps face detection logic. This recognizer takes a raw RGB image and outputs a face object if a face is present. The recognizer abstraction lets us capture that most AR applications operate on specific entities with high-level semantics, such as the face or the skeleton. To enable least privilege, the OS exposes higher level entities through recognizers.

Recognizers create *events* when objects are recognized. A recognizer event contains structured data that encodes information about the objects. Each recognizer declares a public type for this structured data that is available to applications. Applications register callbacks with the OS that fire for events from a particular recognizer; the callbacks accept arguments of the specified type. For example, the recognizer in Figure 6 declares that it will return a list of points corresponding to facial features, plus an RGB texture for the face itself. A callback for an application receives the points and texture in its arguments, but not the rest of the raw RGB frame.

The recognizer is the unit of permission granting. Every time an application attempts to register a callback with the OS for a specific recognizer, the application must be authorized by the user. Different



Raw RGB + Bounding Box

Figure 6: Example of a recognizer for face detection. The input is a feed of raw RGB video plus a region within that video. The recognizer outputs an event if a face is recognized in the region. Applications register callbacks that fire on the event and are called with a list of points outlining the face plus an RGB texture, but not the rest of the video frame.

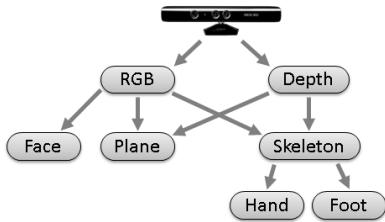


Figure 7: A sample directed acyclic graph of recognizers. Arrows denote how recognizers subscribe to events from other recognizers.

applications can, depending on the user’s authorization, have access to different recognizers. This gives us a *fine-grained* permission mechanism.

Users can restrict applications to only “see” a subset of the raw data stream. For example, Figure 6 shows a bounding box in the raw RGB frame that can be associated with a specific application. If a face happened to be present outside this bounding box, that application would not see the resulting event. Such regions are useful to (1) prevent an application from seeing sensitive information in the environment, and (2) improve efficiency and accuracy of recognizers (e.g., by skipping a region that generates false positives). This bounding box works for sensors where the data is spatial, such as RGB, depth, or skeleton feeds. Other cutoffs would work for other sensors, such as filtering audio to a certain frequency range to ensure voice data is not leaked while other sounds are kept.

Recognizers can also *subscribe* to events from other recognizers, just like applications. The OS includes recognizers for raw sensor streams, such as RGB frames from a camera. Because subscribing to events is an explicit call to the OS, the OS can construct a *dataflow graph* showing how raw sensor streams are progressively refined into objects. Figure 7 shows an example. Having explicit data flow

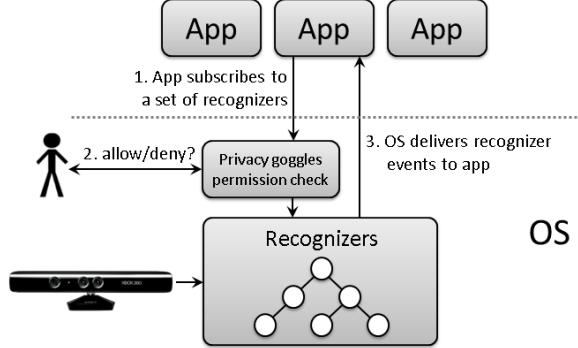


Figure 8: Recognizer-based OS architecture. Applications request subscriptions to sets of recognizers, which the OS then confirms with the user using privacy goggles (Figure 9). Once the user grants permission, the OS delivers recognizer events to subscribed applications.

helps the OS with both security and performance, as we describe below.

Architecture and Threat Model: Figure 8 shows the core architecture of an OS with multiple applications and multiple recognizers. “Root” recognizers acquire raw input from sensors such as the Kinect, then raise events that are consumed by other recognizers. An application may request a subscription for a set of recognizers. The OS confirms this request with the user using our “Privacy Goggles” visualization (Section 3.3). If the user agrees to the request, the OS then delivers events from appropriate recognizers to the application. While our implementation and example focuses on the Kinect, our architecture applies to all forms of object recognition across different platforms such as mobile phones.

The applications are not trusted, while the OS, recognizer implementations, and hardware are trusted. This is similar to the threat model in today’s mobile devices. Third-party recognizer implementations are out of scope of this paper, but we describe in Section 7 key new challenges they raise.

3.1 Security Benefits

The recognizer abstraction has two key security benefits:

Least privilege: Applications can be given access only to the recognizers they need, instead of to raw sensor streams. Before recognizers, OSes could expose permissions only at a coarse granularity. As we will see in Section 5, a small set of recognizers is sufficient to cover most shipping AR applications.

Reducing permission requests: If an application requests access to the skeleton and hand recognizers from the DAG shown in Figure 7, a user only needs to grant access to the skeleton recognizer.

More generally, the recognizer DAG allows us to find such dependencies efficiently. This helps with warning fatigue, which is one of the major problems with existing permission systems [11].

3.2 Performance Benefits

Besides the security benefits described above, recognizer DAGs also allow us to achieve significant performance gains.

Sharing recognizer output: Most computer vision algorithms used in recognizers are computationally intensive. Since concurrently running AR applications may access the same recognizers, our recognizer DAG allows us to run such shared recognizers only once and send the output to all subscribed applications. Our experiments show that this results in significant performance gains for concurrent applications.

On-demand invocation: The recognizer DAG allows us to find all recognizers being accessed by currently active applications at all times. We can then prevent scheduling inactive recognizers.

Concurrent execution: The recognizer DAG also allows us to find true data dependencies between the recognizers. We leverage this to schedule independent recognizers in multiple threads/cores and thus minimize inter-thread/core communication.

Offloading: Some recognizers require special-purpose hardware such as a powerful GPU that may not be available in mobile devices. These recognizers must be outsourced to a remote server. For example, the real-time 3D model generation of KinectFusion [23] requires a high-end nVidia desktop graphics card, such as a GeForce GTX 680. Therefore, if we want to use a commodity tablet with a Kinect attached to scan objects and create models, we must run the recognizer on a remote machine. While applications could implement offloading themselves, adding offloading support to the OS preserves least privilege. For example, the OS can offload KinectFusion without giving applications access to raw RGB and depth inputs, which would be required if an application were to offload it manually.

3.3 Privacy Goggles

We introduce *privacy goggles*, an “application-eye view” of the world for running applications. For example, if the application has access to a skeleton recognizer, a stick figure in the “privacy goggles view” mirrors the movements of any person in view of the system, as shown in Figure 9. A trusted visualization method for each recognizer communicates

the capabilities of applications that have access to this recognizer. If an application requests access to more than one recognizer, the OS will compose the appropriate visualizations. In Section 5 we survey 462 people to demonstrate that privacy goggles do effectively communicate capabilities for “core recognizers” derived from analyzing shipping AR applications. Privacy goggles are complementary to existing permission widgets, such as those of Howell and Schechter [16], which allow users to understand how apps perceive them in real time.

Permission Granting and Revocation. Privacy goggles lay a foundation for permission granting, inspection, and revocation experiences. For example, we can generalize existing install-time manifests to use privacy goggles visualizations. At installation time, a short prepared video could play showing a “raw” data stream side by side with the privacy goggles view. The user can then decide to allow access to all, some, or none of the recognizers. We are currently evaluating this approach. Because manifest-based systems have known problems with user attention [11], we are also exploring how access-control gadgets might interact with privacy goggles [27].

A major difference between privacy goggles and existing permission granting systems like Android manifests is the visual representation of the sensitive data. The visual representation helps users to make informed decisions about granting and revoking an application’s access to different recognizers. Traditional systems do not need this representation because they ask for permissions about well-understood low-level hardware, such as the camera and microphone. Because we are fine-grained and must consider higher-level semantics, we need privacy goggles to show the impact of allowing applications access to specific recognizers.

After installation, privacy goggles are a natural way to inspect sensitive data exposed to applications. The user can trigger a “privacy goggles control panel” to zero in on a particular application or view a composite for all applications at once. From the control panel, a user can then turn off an application’s access to a recognizer or even uninstall the application.

3.4 Handling Recognizer Errors

Because our permission system depends on recognizer outputs, we have a new challenge: *recognizer errors*. Object recognition algorithms inside recognizers have both false positives and false negatives. A false negative means that applications will not “see” an object in the world, impacting functionality.

False negatives, however, do not concern privacy.

A false positive, on the other hand, means that an application will see more information than was intended. In some cases the damage will be limited, because the recognizer will return information that is not sensitive. For example, a false positive from a recognizer for hand positions is unlikely to be a problem. In others, false positives could leak portions of raw RGB frames or other more sensitive data.

To address recognizer errors, we introduce a new OS component for *recognizer error correction*. While recognizers themselves implement various techniques to decrease errors, in our setting false positives are damaging, while false negatives are less important. Therefore, we are willing to tolerate more false negatives and fewer false positives than a recognizer developer who is not concerned with basing permission decisions on a recognizer’s output.

For recognizer error correction, we first considered two techniques: *blurring* and *frame subtraction*, both of which are well-known graphics techniques that can be applied in a *recognizer-independent* way. We apply these techniques to recognizer inputs to reduce potential false positives, accepting that they may raise false negatives. We discuss the results and show data in Section 5.

In addition, the OS has information not available to an individual recognizer developer: results from *other recognizers* in the same system on the same environment. Recognizer error correction can therefore employ *recognizer combination* to reduce false positives. For example, if a depth camera is available, the OS can use the depth camera to modify the input to a face detection recognizer. By blanking out all pixels past a certain depth, the OS can ensure a face recognizer focuses only on possible faces near the system. While combination does require knowing something about what a recognizer does, it is independent of the internals of the recognizer implementation. For another example, the OS can combine a skeleton recognizer and a face recognizer to release a face image only if there is also a skeleton with its head in the appropriate place.

3.5 Adding New Recognizers.

Today’s AR platforms ship with a small fixed set of recognizers. Applications that want capabilities outside that set need to both innovate on object recognition and on app experience, which is rare. As the platforms mature, we expect additional recognizers to appear. The main incremental costs for new recognizers are 1) coming up with a privacy goggles visualization, 2) measuring the effectiveness of this

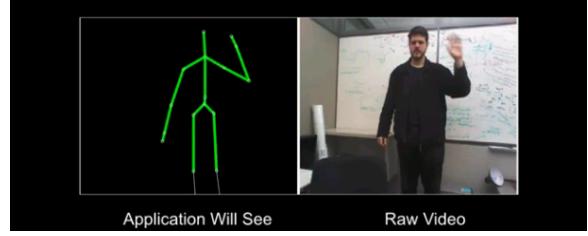


Figure 9: Example of “privacy goggles.” The user sees the “application-eye view” for a skeleton recognizer.

API	Purpose
<code>init</code>	Register
<code>destruct</code>	Clean up
<code>event_generate</code>	Notify apps of recognized objects
<code>visualize</code>	Render recognized objects
<code>filter</code>	Restrict domain for recognition
<code>cache_compare</code>	Compare to previous inputs

Figure 10: The APIs implemented by each recognizer. The first four are required, while `filter` and `cache_compare` are optional.

visualization at informing users (and re-designing if not effective), and 3) defining relationships with existing recognizers to support recognizer error correction. For example, a new “eye recognizer” would have the invariant that every eye detected should be on a head detected by the skeleton recognizer. Third-party recognizers raise additional security issues outside the scope of this paper; we discuss them briefly in Section 7.

4 Implementation

We have built a prototype implementation of our architecture. Our prototype consists of a *multiplexer*, which plays the role of an OS “kernel”, and *ARLib*, a library used by AR applications to communicate to the multiplexer. Our system uses the Kinect RGB and depth cameras for its sensor inputs.

Multiplexer. The multiplexer handles access to the sensors and also contains implementations of all recognizers in the system. Our applications no longer have direct access to Kinect sensor data and must instead interact with the multiplexer and retrieve this data from recognizers. The multiplexer supports simultaneous connections from multiple applications. To simplify implementation, we built the multiplexer as a user-space program in Windows that links against the Kinect for Windows SDK.

The multiplexer registers each recognizer using a static, well-known name. Applications use these names to request access to one or more recogniz-

```

var client = new MultiplexerClient();
client.Connect();
client.OnFace += new FaceEventCallback(ProcessFace);
...
public void ProcessFace(FTPoint[] points)
{
    if (points.Length > 0) {
        DrawFace(points);
    } else {
        RemoveFace();
    }
}

```

Figure 12: Code used by a sample C# application to connect to the multiplexer, subscribe to events from the face recognizer, and use those events to update its face visualization.

ers. When the multiplexer receives such an access request, it asks the user whether or not permission should be granted using privacy goggles (Section 3.3). If the user grants permission, the multiplexer will forward future recognizer events, such as face mesh points from a face recognizer, to the application.

The multiplexer interacts with recognizers via an API shown in Figure 10. All recognizers must implement the first four API calls. The multiplexer calls `init` to initialize a recognizer and `destruct` to let a recognizer release its resources. In our current implementation, the multiplexer calls the `event_generate` function of each recognizer in a loop, providing prerequisite recognizer inputs as parameters, to check if any new objects have been recognized. If so, the recognizer will return data that the multiplexer will then package in an event data structure and pass to all subscribed applications. We plan to implement a more efficient interrupt-driven multiplexer in the future.

The next two API calls are optional. The `filter` call allows the multiplexer to tell the recognizer that only a specific subset of the raw inputs should be used for recognition. For example, only a sub-rectangle of the video frame should be considered for a face detector. Finally, `cache_compare` is a recognizer-specific comparator function that takes two sets of recognizer inputs and determines whether they are considered equal. The multiplexer uses this comparator to implement per-recognizer caching. For example, the multiplexer may pass the previous and current RGB frames to the `cache_compare` function of the face recognizer and potentially avoid a recomputation of the face model if the two frames have not sufficiently changed.

Our multiplexer and recognizers consisted of about 3,000 lines of C++ code. We wrote a total of nine recognizers, which we summarize in Figure 11.

Application support. Applications targeting our multiplexer run in separate Windows processes. Each application links against the *ARLib* library we have built. *ARLib* communicates with the multiplexer over local sockets and handles marshaling and unmarshaling of recognizer event data. By calling *ARLib* functions, an application can request access to specific recognizers and register callbacks to handle recognizer events. *ARLib* provides two kinds of interfaces: a low-level interface for applications written in C++ and higher-level wrappers for .NET applications written in C# or other managed languages. *ARLib* consists of about 500 lines of C++ code and 400 lines of C# code.

Sample code in Figure 12 shows a part of a test application we wrote that detects faces and draws pictures on the screen which follow face movements. The application connects to the multiplexer and subscribes to face recognizer events. In our implementation, these events contain approximately 100 points corresponding to different parts of the face, or 0 points if a face is not present. The application handles these events in the `ProcessFace` callback by checking if a face is present and calling a separate function (not shown) that updates the display.

In addition to face visualization, we ported a few other sample applications bundled with the Kinect SDK to our system. These included a skeleton visualizer and raw RGB and depth visualizers. We found the porting effort to be modest, aided in part by the fact that we modeled our event data formats on existing Kinect SDK APIs. In each case, we only changed a handful of lines dealing with event subscription. We additionally wrote two applications from scratch: a 500-line C++ application that translates hand gestures into mouse cursor movements, and a 300-line C# application that uses face recognition to annotate people with their names. Overall, we found our multiplexer interface simple and intuitive to use for building AR applications.

5 Evaluation

We first evaluate how recognizers are used by an analysis of 87 shipping AR applications and users' mental models of AR applications. A survey of 462 respondents shows that users expect AR applications to have limited access to raw data. Furthermore, no shipping application needs continuous RGB access, and in fact a set of four recognizers is sufficient for almost all applications. For these "core" recognizers, we design privacy goggles visualizations and evaluate how well users understand them. Next, we look at how the OS can mitigate recognizer er-

Recognizer	Input dependencies	Output
RGB	<i>Kinect</i>	RGB camera frames
Depth	<i>Kinect</i>	Depth camera frames
Skeleton	<i>Kinect</i>	Computed skeleton model(s)
Hand	Skeleton	Hand positions
FaceDetect	RGB	2D face models for faces in current view
PersonTexture	Depth, Skeleton	Depth “cutout” of a person
Plane	RGB, Depth	3D polygon coordinates constructed with KinectFusion (see Section 5.3)
FaceRecognize	RGB, FaceDetect	Name of person in current view (see Section 5.3)
CameraMotion	<i>Kinect</i>	Camera movements detected using an accelerometer/gyro

Figure 11: The nine recognizers implemented by our multiplexer. A “Kinect” input dependency means that the recognizer obtains data directly from the Kinect rather than other recognizers.

rors once an application has access to recognizers. Finally, we show that our abstraction enables performance improvements, making this a rare case when improved privacy leads to improved performance.

5.1 Recognizers

Core Recognizers. We analyzed 87 AR applications on the Xbox Kinect platform, including all applications sold on Amazon.com. We focused on Kinect because it is widely adopted and sits in a user’s home. For each application, we manually reviewed their functionality, either through reading reviews or by using the application. From this, we extracted “recognizers” that would be sufficient to support the application’s functionality.

Figure 13 shows the results. Four *core recognizers* are sufficient to support around 89% of shipping AR applications. The set consists of skeleton tracking, hand position, *person texture*, and keyword voice commands. Person texture reveals a portion of RGB video around a person detected through skeleton tracking, but with the image blurred or otherwise transformed to hide all details. Fitness applications, in particular, use person texture when instructing the user on proper form.

After the core set, there is a “long tail” of seven recognizers. For example, the Alvin and the Chipmunks game uses voice modulation to “Alvin-ize” the player’s voice, and NBA Baller Beats actually tracks the location of a basketball to check that the player dribbles in time to music. *None* of the applications in our set, however, require continuous access to RGB data. Instead, applications take a short video or photo of the player so that she can share how silly she looks with friends; this could be handled via user-driven access control [27]. Only 3 applications require audio access beyond voice command triggers. There is plenty of room to improve privacy with least privilege enabled by the recognizer abstraction.

Privacy Expectations for Applications. To

Recognizer	% Apps
Skeleton	94.3%
Person Texture (PT)	25.3%
Voice Commands (VC)	3.44%
Hand Position (HP)	5.74%
Video Clip	3.4%
Picture Snap	1.1%
Voice Intensity	1.1%
Voice Modulation	1.1%
Speaker Recognition	1.1%
Sound Recognition	1.1%
Basketball Tracking	1.1%
Skeleton+PT+VC	82.75%
Skeleton+PT+VC+HP	89.65%

Figure 13: Analysis of all recognizers used by 87 shipping Xbox applications. For each recognizer, we show what percentage of apps use that recognizer (and possibly others). We also show two sets of recognizers, and for each set, the percentage of apps that use recognizers in this set and no others. A set of four recognizers covers 89.65% of all applications. No application needs continuous raw RGB access, and only 3 need audio access beyond voice commands.

learn users’ mental models of AR application capabilities, we showed 462 survey respondents a video of a Kinect “foot piano” application in action: the Kinect tracks foot positions and plays music. We then asked about the capabilities of the application. Figure 17(A) shows the results. Over 86% of all users responded that the application could see the foot positions, while a much smaller number believed this application had other capabilities. Overall, users expect applications will not see the entire raw sensor stream.

Privacy Goggles for Core Recognizers. As we discussed in Section 3, every recognizer must implement a visualization method to enable the privacy goggles view. The OS uses these visualizations to display to the user what information is obtained by each application. We developed privacy goggles visualizations for three of the four core recognizers: skeleton, hand position, and person texture. While voice commands are also a core recognizer,

You have found an app that you want to install. Right before installing, the video below plays. On the right, a sample “raw” video. On the left, a view of what the app will see if it is installed. Which of the following can the app do?

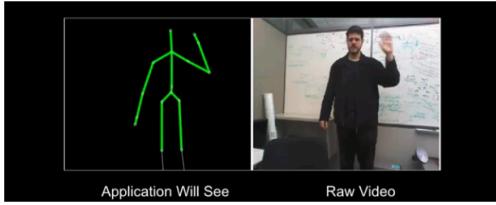


Figure 14: Example survey question for privacy goggles. An embedded warning video shows two views: the raw video on the right, and what the application will see on the left. Survey respondents watched the warning video, then answered questions about what the app could or could not do after installation. Out of 152 respondents, 80% correctly identified that the app could see body position, and 47% correctly determined the app could see hand positions.

Consider the two pictures below. Which picture contains “more sensitive” information?

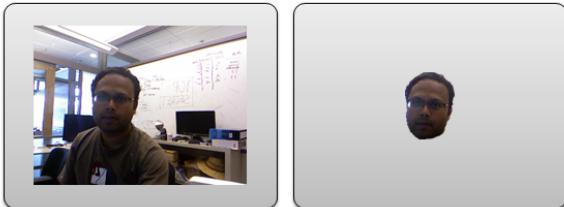


Figure 15: Example survey on relative sensitivity. Respondents indicated which picture is more sensitive: the “raw” RGB video frame or an image showing only the output of a face detector. Out of 50 respondents, 86% indicated the raw image was more sensitive.

we decided to focus first on the visual recognizers and leave visualization of voice commands for future work.

Privacy Attitudes for Core Recognizers. We then conducted surveys to measure the relative sensitivity of the information released by the core recognizers. We also added the “face detector” recognizer, because intuitively the face is private information, and a “Raw” video recognizer that represents giving all information to the application. For each pair of recognizers, we showed a visualization from the same underlying video frame, then asked the participant to state which picture was “more sensitive” and why. Figure 15 shows an example comparing raw RGB and face detector recognizers.

For each pair of recognizers, we asked 50 people to rate which picture contained information that was “more sensitive.” Figure 16 shows the results. In total we had 500 survey respondents, all from the United States. As expected, respondents find that the raw RGB frame is more sensitive than any other

Recognizers		Left more sensitive	95% CI
Left	Right		
Raw	Face	86%	± 9.6%
Raw	Skeleton	78%	± 11.48%
Raw	Texture	88%	± 9.01%
Raw	Hand	88%	± 9.01%
Texture	Skeleton	82%	± 10.65%
Texture	Face	35%	± 13.22%
Texture	Hand	84%	± 10.16%
Skeleton	Face	24%	± 11.84%
Skeleton	Hand	84%	± 10.16%
Hand	Face	22%	± 11.48%

Figure 16: Results from relative sensitivity surveys. Users were shown two pictures, one from each recognizer, here shown as the “left” and the “right” recognizer. The table reports which picture respondents thought contained “more sensitive” information and the 95% confidence interval. For example, in the first line, 86% of people thought that the view from the “Raw” RGB recognizer was more sensitive than the view from a face detector, with a 95% confidence interval of ± 9.6%.

recognizer. Based on the responses, we can order recognizers from “most sensitive” to “least sensitive”, as follows: Raw, Face, Person Texture, Skeleton, and finally the least sensitive is Hand Position.

Effectiveness of Privacy Goggles. Finally, we evaluated whether our “privacy goggles” visualizations successfully communicate the capabilities of applications. We created three surveys, one for each of the skeleton, person texture, and hand recognizers. We had at least 150 respondents to each survey, with a total of 462 respondents. Our surveys are inspired by Felt *et al.*’s Android permission “quiz.” [11]

We showed a short video clip of the privacy goggles visualization for the target recognizer. Figure 14 shows an example for the skeleton recognizer. The right half shows the raw RGB video of a person writing on a whiteboard and handling a small ceramic cat figurine. The left half shows the “application-eye view” showing the detected skeleton. We then asked users what they believed the capabilities of the application would be if installed. Figure 17 shows the results, with a check mark next to correct answers. We see that a large number of respondents (over 80%) picked the correct result and relatively few picked incorrect results. This shows that privacy goggles are effective at communicating application capabilities to the user.

Respondent Demographics. Our survey participants were recruited from the U.S. through uSample [30], a professional survey service, via the Instant.ly web site. We did not specify any restrictions on demographics to recruit. As reported by uSample, participants are 66% female and 33%

A. Foot Piano (462 respondents)		B. Skeleton (152 respondents)	
See my body position	76 (16%)	See what I look like	17 (11%)
See my foot positions ✓	400 (86%)	See my body position ✓	122 (80%)
See what I look like	28 (6%)	See my location	24 (16%)
See the entire video	52 (11%)	Read the contents of the whiteboard	14 (9%)
Learn my heart rate	21 (4%)	Send premium SMS messages on my behalf	4 (3%)
None of the above	20 (4%)	Track the position of my hands ✓	71 (47%)
I don't know	20 (4%)	None of the above	4 (3%)
		I don't know	1 (1%)
C. Person Texture (156 respondents)		D. Hand Position (154 respondents)	
See what I look like	36 (23%)	See what I look like	17 (11%)
See my body position ✓	137 (88%)	See my body position	32 (21%)
See my location	25 (16%)	See my location	14 (9%)
See the ceramic cat	19 (12%)	See the ceramic cat	12 (8%)
Read the contents of the whiteboard	5 (3%)	Read the contents of the whiteboard	7 (5%)
Send premium SMS messages on my behalf	0 (0%)	Send premium SMS messages on my behalf	2 (1%)
Track the position of my hands ✓	60 (38%)	Track the position of my hands ✓	125 (81%)
None of the above	2 (1%)	None of the above	3 (2%)
I don't know	5 (3%)	I don't know	4 (3%)

Figure 17: Results from privacy goggles effectiveness surveys. For each of our three core recognizers, we first asked respondents to answer questions about the capabilities of a Kinect “foot piano” application based on a short video of the application in use (A). We next showed a privacy goggles “permission warning video” and asked questions about what the application could do if installed (B-D).

male, with 10.2% in the 0–22 age range, 12.9% 22–26, 21.2% 26–34, 16.8% 34–42, 13.5% 42–50, 15.1% 50–60, 8.1% 60–70, and 1.8% 70 or older.

Human Ethics Statement. Our experiments include surveys of anonymous human participants. Our institution does not have an Institutional Review Board (IRB), but it does have a dedicated team whose focus is privacy and human protection. This team has pre-approved survey participant vendors to ensure that they have privacy policies which protect participants. We followed the guidelines of this team in choosing our survey vendor. We also discussed our surveys with a member of the team to ensure that our questions did not ask for personally identifiable information, that they were not overly intrusive, and that no other issues were present.

5.2 Noisy Permissions

While privacy goggles are effective at communicating what an app should and should not see to the user, the recognizers we use can have false positives. These could leak information to applications. We first evaluated a representative set of recognizers on well-known vision data sets to quantify the problem. Next, we evaluated OS-level mitigations for false positives.

Recognizer Accuracy. We picked three well-known data sets for our evaluations: (1) a Berkeley dataset consisting of pictures of objects, (2) an INRIA dataset containing pictures of a talking head, and (3) a set of pictures of a face turning toward the

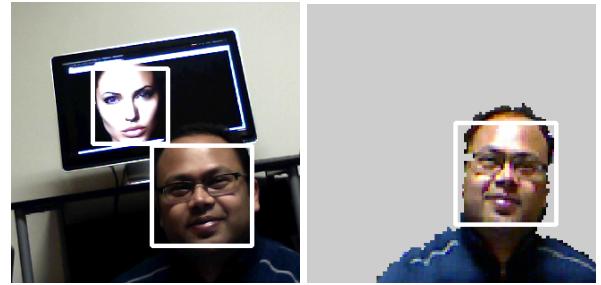


Figure 19: Recognizer combination in action. The left figure shows results of running a face detector on a raw RGB video frame. Two faces are detected, but only one belongs to a real person. On the right, face detection is run after combining RGB and depth. Only the real person is detected.

camera and then away. We then evaluated baseline false positive and false negative rates for seven object recognition algorithms contained in the widely adopted OpenCV library. All seven had false positives on at least one of the data sets.

Input Massaging. We then implemented *pre-permission blurring*, in which frames are put through a blurring process using a box filter before being passed to the face detection algorithm. We used a 12×12 box filter. We also used *frame subtraction* as a heuristic to suppress recognizer false positives. In frame subtraction, when a recognizer detects an object with a bounding box b in a frame F_1 that it did not detect in the previous frame F_0 , we compute the difference $Crop(F_1, b) - Crop(F_0, b)$ and check the number of pixels that have a difference. If this num-

Recognizer	Data Set	False Positive	False Negative	BlurFP	BlurFN	SubFP	SubFN
Face	Objects	10.6%	0%	6%	0%	9.6%	0%
Face	Talking Head	0.2%	0%	0%	0%	0%	0%
Face	Turning Face	19.1%	16.1%	15%	16.1%	17.64 %	16.1%
FullBody	Objects	14.8%	0%	3.5%	0%	9.6%	0%
FullBody	Talking Head	0.2 %	0%	0%	0%	0%	0%
FullBody	Turning Face	24.6%	0%	22.7 %	0%	20%	0%
LowBody	Objects	19.5%	0%	4.6%	0%	17.9%	0%
LowBody	Talking Head	6.2%	0%	0.3%	0%	0%	0%
LowBody	Turning Face	33%	0%	25%	0%	28.3%	0%
UpperBody	Objects	41%	0%	10%	0%	38.1%	0%
UpperBody	Talking Head	5.3%	0%	0.1%	0%	0.2%	0%
UpperBody	Turning Face	86%	0%	0%	0%	19.9%	0%
Eye	Object	35%	0%	83%	0%	32 %	0%
Eye	Talking Head	64 %	0 %	100 %	0%	30%	2%
Eye	Turning Face	23 %	5%	100%	0%	9%	10%
Nose	Object	17.8%	0%	57%	0%	17.1 %	0%
Nose	Talking Head	90 %	0%	86%	0%	90%	0%
Nose	Turning Face	24.5 %	0%	43%	7%	24%	0%
Mouth	Object	61%	0%	75%	0%	59 %	0%
Mouth	Talking Head	100 %	0%	75%	0%	100%	0%
Mouth	Turning Face	75 %	0%	82%	0%	74%	0%

Figure 18: False positive and false negative rates for OpenCV recognizers on common data sets. False positives are important because they could leak unintended information to an application. We also show the effect of blurring and frame subtraction. For blurring we used a 12x12 box filter.

ber does not exceed a threshold, we ignore the detected object as a false positive.

For three out of our seven recognizers, blurring decreases false positives with no effect on false negatives, with a maximum reduction for our lower body recognizer from 19.5% false positives to 4.6% false positives. For the remaining recognizers, false positives decrease but false negatives also increase. Frame subtraction decreases false positives for six out of seven recognizers and has no effect on the seventh, with no impact on false negatives. This is in line with our goals, because false positives are more damaging to privacy than false negatives. The full results are in Figure 18.

Recognizer Combination. Finally, we implemented *recognizer combination*, in which the OS can take advantage of the fact that multiple recognizers are available. Specifically, we combined the OpenCV face detector with the Kinect depth sensor. We chose the OpenCV face detector because its developers could depend only on the presence of RGB video data. We ran an experiment that first acquires an RGB and depth frame, then blanks out all pixels with depth data that is further away than a threshold. Next, we fed the resulting frame to the face detector. An example result is shown in Figure 19. On the left, the original frame shows a false positive detected behind the real person. On the right, recognizer combination successfully avoids the false

Recognizers	Kinect SDK	Our framework
RGB Video	29.87 fps	30.02 fps
Skeleton	29.59 fps	28.65 fps
Face	28.24 fps	28.00 fps

Figure 20: Frame rates for a single application using the Kinect SDK vs. using recognizers from our system. Our system incurs negligible overhead.

positive.

5.3 Performance

In our performance evaluation, we (1) measure the overhead of using our system compared to using the Kinect SDK directly, (2) quantify the benefits of recognizer sharing for multiple concurrent applications, and (3) evaluate the benefit of recognizer offloading.

Overhead over Kinect SDK. Compared to directly using the Kinect SDK, an application that uses our multiplexer will face extra overhead due to recognizer event processing in the multiplexer as well as data marshaling and transfer over local sockets. To quantify this overhead, we wrote two identically functioning applications to obtain and display a raw 640x480 RGB video feed, a skeleton model, and points from a face model. The first application used the Kinect SDK APIs directly, while the second

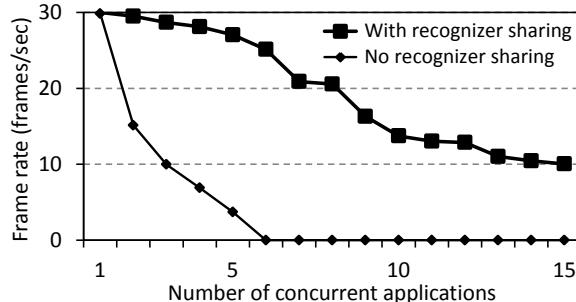


Figure 21: Effect of sharing a concurrent RGB video stream between applications. Our framework enables 25 frames per second or higher for up to six applications, while without sharing the frame rate drops.

used our multiplexer with RGB, skeleton, and face detection recognizers.

Figure 20 shows the frame rates when running these two applications on a desktop HP xw8600 machine with a 4-core Core i5 processor and 4 GB of RAM. We see that, fortunately, our current prototype incurs negligible overhead over the Kinect SDK when used by a single application.

Recognizer Sharing. Next, we ran multiple concurrent copies of the two applications above to evaluate the benefits of recognizer sharing as well as the scalability of our prototype. Since the Kinect SDK does not permit concurrent applications, we wrote a simple wrapper for simulating that functionality, i.e., allowing multiple applications as if they were linking to independent copies of the Kinect SDK.

Figure 21 shows the average frame rate for multiple concurrent applications using the RGB recognizer. We see that without recognizer sharing, frame rates quickly stall as the number of concurrent applications increases, becoming unusable beyond five applications. In contrast, our approach maintains at least 25 frames per second up to six concurrent applications and degrades gracefully thereafter. We experienced similar recognizer sharing benefits for skeleton and face recognizers.

While currently shipping AR platforms do not yet support multiple concurrent applications, the above experiment demonstrates that our system is ready to *efficiently* embrace such support. Indeed, we believe this to be the future of AR platforms. Mobile phone “AR Browsers” such as Layar already expose APIs for third-party developers, with over 5,000 applications written for Layar alone [20]. Users will benefit from running these applications concurrently; for example, looking at a store front, one application may show reviews of the store, while another shows information about its online catalog, and yet a third

Recognizer	Throughput (frames/sec)		
	Tablet	Offloaded	Server
Plane detection	0	4.17	4.46
Face recognition	2.04	2.73	2.84

Figure 22: Frames processed per second when running recognizers (1) locally on a client tablet, (2) offloaded to the server and shipping results back to the tablet, and (3) locally on the server.

application attaches a name to the face of someone walking by.

Recognizer Offloading. We evaluated offloading of two resource-intensive recognizers: plane and face recognition. The plane recognizer reconstructs planes in the current scene using KinectFusion, which computes 3D models from Kinect depth data [23]. The face recognizer uses the Microsoft Face SDK [21] to identify the name of the person in the scene using a small database of known faces.

We implemented offloading across two devices linked by an 802.11g wireless network. For face recognition, the client sends RGB bitmaps of the current scene to the server as often as possible; the client additionally includes the depth bitmap for the plane recognizer.

Our client device was a Samsung Series 7 tablet running Windows 8 Pro 64-bit with a 2-core Core i5 processor and 4 GB of RAM, hooked up to a Kinect. Our server device was a desktop HP Z800 machine running Windows 8 Pro 64-bit with two 4-core Xeon E5530 processors, 48 GB of RAM, and an Nvidia GTX 680 GPU.

The first two columns of Figure 22 show throughputs experienced by the client when running recognizers locally and when offloading them to the server. The plane recognizer requires a high-end Nvidia GPU, which prevented it from running on our client at all; we report this as zero frames per second. With offloading, however, the client is able to detect planes 4.2 times per second. For face recognition, the client processed 2.73 frames per second when offloading, a 34% improvement in response time compared to running face recognition locally. In addition, when run locally, face recognition placed heavy CPU load on the client, completely consuming one of its two cores. With offloading, the client’s CPU consumption dropped to 15% required to send bitmaps, saving battery and freeing resources for processing other recognizers. Note that our setup allows the offloading server to service multiple clients in parallel. For example, the server was able to handle eight concurrent face recognition clients before saturating.

We also considered the overhead of our offload-

ing mechanism by plugging a Kinect into our server and running the recognizer framework directly on it. Column 3 of Figure 22 shows these results. We see that offloading with the Kinect on the client is only 4–7% slower than running the Kinect on the server, meaning that the offloading overhead of transferring bitmaps and recognition results is reasonable.

6 Related Work

Augmented Reality. Azuma surveyed augmented reality, defining it as real-time registration of 3-D overlays on the real world [1], later broadening it to include audio and other senses [2]. We take a broader view and also consider systems that take input from the world. Qualcomm now has an SDK for augmented reality that includes features such as marker-based tracking for mobile phones [26]. Previous work by our group has laid out a case for adding OS support for augmented reality applications and highlighted key challenges [7].

Common shipping object recognition algorithms include skeleton detection [29], face and headpose detection [31, 21], and speech recognition [22]. More recently, Poh *et al.* showed that heart rate can be extracted from RGB video [24]. Our recognizer graph and simple API allow quickly adding new recognizers to our system.

Sensor Privacy. There are several parts to sensor privacy: access control on sensors, sensor data usage control once an application obtains access to sensor data, and access visualization; we discuss related work for each.

Access control can take the form of user permissions. iOS’s permission system is to prompt a user at the first time of the sensor access (such as a map application first accessing GPS). Android and latest Windows OSes use manifests at application installation time to inform the user of sensor usage among other things; the installation proceeds only if the user permits the application to permanently access all the requested permissions. These existing permission systems are either disruptive or ask users’ permissions out-of-context. They are not least-privilege; permanent access is often granted unnecessarily. Felt et al [11] has shown that most people ignore manifests, and the few who do read manifests do not understand them. To address these issues, access control gadgets (ACGs) [27] were introduced to be trusted UI elements for sensors, which are embeddable by applications; users’ authentic actions on an ACG (e.g., a camera trusted UI) grants the embedding application permission to access the

represented sensor. In this paper, we argue that even the ACG style of permission granting is too coarse-grained for augmented reality systems because most AR applications only require specific objects rather than the entire RGB streams (Section 5.1).

Another form of access control is to reduce the sensitivity of private data (e.g., GPS coordinates) available to applications. MockDroid [3] and AppFence [14] allow using fake sensor data. Krumm [19] surveys methods of reducing sensitive information conveyed by location readings. Differential privacy [9] uses well-known methods for computing the amount of noise to add to give strong guarantees against an adversary’s ability to learn about any specific individual. Similarly, we proposed modifying sensor inputs to recognizers in specific ways to reduce false positives that could result in privacy leaks. Darkly [18] transforms output from computer vision algorithms (such as contours, moments, or recognized objects) to blur the identity of the output. Darkly can be applied to the output of our recognizers.

Once an application obtains access to sensors, information flow control approaches can be used to control or monitor an application’s usage of the sensitive data as in TaintDroid [10] and AppFence [14].

In access visualization, sensor-access widgets [15] were proposed to reside within an application’s display with an animation to show sensor data being collected by the application. Darkly [18] also gives a visualization on its transforms (see above). Our privacy goggles apply similar ideas to the AR environment, allowing a user to visualize an application’s eye view of the user’s world.

Abstractions for Privacy. Our notion of taking raw sensor data and providing the higher-level abstraction of recognizers is similar to CondOS [4]’s notion of Contextual Data Units. However, they neither choose a set of concrete Contextual Data Units that are suitable for a wide variety of real-world applications nor address privacy concerns that arise from applications having access to Contextual Data Unit values. Koi [13] provides a location matching abstraction to replace raw GPS coordinates in applications. The approach in Koi is limited to location data and may require significant work to integrate into real applications, while our recognizers cover many types of sensor data and were specifically chosen to match application needs.

7 Future Work

Further Recognizer Visualization. The recognizers we evaluated had straightforward visualiza-

tions, such as the Kinect skeleton. As we noted, some recognizers, such as voice commands, do not have obvious visualizations. Other recognizers might extract features from raw video or audio for use by a variety of object recognition algorithms, but not in themselves have an easily understood semantics, such as a fast Fourier transform of audio. One key challenge here is to design visualizations for privacy goggles that clearly communicate to users the impact of allowing application access to the recognizer. For example, with voice commands we might try showing a video with sound where detected words are highlighted with subtitles. A second key challenge is characterizing the privacy impact of algorithmic transforms on raw data, especially in the case of computer vision features that have not been considered from a privacy perspective.

Third-Party Recognizers. All the recognizers in this paper are assumed trusted. To enable new experiences, we would like to support extension of the platform with third-party recognizers. Supporting third-party recognizers raises challenges, including permissions for recognizers as well as sandboxing untrusted GPU code without sacrificing performance. We have developed recognizers in a domain-specific language that enables precise analysis [8]. Dealing with such challenges is intriguing future work, similar in spirit to research on third-party driver isolation in an OS. For example, we might require such recognizers to go through a vetting program and then have their code signed, similar to drivers in Windows or applications on mobile phone platforms.

Sensing Applications. Besides traditional AR applications, other applications employ rich sensing but do not necessarily render on human senses. For example, robots today use the Kinect sensor for navigating environment, and video conferencing can use the “person texture” recognizer we describe. One of our colleagues has also suggested that video conferencing can benefit from a depth-limited camera [28]. These applications may also benefit from recognizers.

Bystander Privacy. Our focus is on protecting a user’s privacy against untrusted applications. Mobile AR systems such as Google Glass, however, have already raised significant discussion of *bystander privacy* — the ability of people around the user to opt out of recording and object recognition. Our architecture allows explicitly identifying all applications that might have access to bystander information, but it does not tell us when and how to stop sending recognizer events to applications. Making the system aware of these issues is important future work.

8 Conclusions

We introduced a new abstraction, the *recognizer*, for operating systems to support augmented reality applications. Recognizers allow applications to raise the level of abstraction from raw sensor data, such as audio and video streams, to ask for access to specific recognized objects. This enables applications to act with the least privilege needed. Our analysis of existing applications shows that all of them would benefit from least privilege enabled by an OS with support for recognizers. We then introduced a “privacy goggles” visualization for recognizers to communicate the impact of allowing access to users. Our surveys establish a clear privacy ordering on core recognizers, show that users expect AR apps to have limited capabilities, and demonstrate privacy goggles are effective at communicating capabilities of apps that access recognizers. We built a prototype on top of the Kinect for Windows SDK. Our implementation has negligible overhead for single applications, enables secure OS-level offloading of heavyweight recognizer computation, and improves performance for concurrent applications. In short, the recognizer abstraction improves privacy *and* performance for AR applications, laying the groundwork for future OS support of rich sensing and AR application rendering.

9 Acknowledgements

We thank Janice Tsai, our Privacy Manager, for reviewing our survey. We thank Doug Burger, Loris D’Antoni, Yoshi Kohno, Franziska Roesner, Stuart Schechter, Margus Veanes, and John Vilk for helpful discussions and review of drafts. Stuart Schechter suggested the idea of a depth-limited camera for teleconferencing scenarios. This work was carried out while the first and fourth author were interning at Microsoft Research.

References

- [1] R. T. Azuma. A survey of augmented reality. *Presence: Teleoperators and Virtual Environments*, 6(4):355–385, August 1997.
- [2] R. T. Azuma, Y. Baillot, R. Behringer, S. Feiner, S. Julier, and B. MacIntyre. Recent advances in augmented reality. *Computer Graphics and Applications*, 21(6):34–47, 2001.
- [3] A. R. Beresford, A. Rice, N. Skehin, and R. Sohan. MockDroid: Trading privacy for application functionality on smartphones. In *Workshop on Mobile Computing Systems and Applications (HotMobile)*, 2011.
- [4] D. Chu, A. Kansal, J. Liu, and F. Zhao. Mobile apps: It’s time to move up to condOS. May

2011. <http://research.microsoft.com/apps/pubs/default.aspx?id=147238>.
- [5] M. Corporation. Kinect for xbox 360 privacy considerations, 2012. <http://www.microsoft.com/privacy/technologies/kinect.aspx>.
- [6] M. Corporation. Kinect for Windows SDK, 2013. <http://www.microsoft.com/en-us/kinectforwindows/>.
- [7] L. D’Antoni, A. Dunn, S. Jana, T. Kohno, B. Livshits, D. Molnar, A. Moshchuk, E. Ofek, F. Roesner, S. Saponas, M. Veane, and H. J. Wang. Operating system support for augmented reality applications. In *Hot Topics in Operating Systems (HotOS)*, 2013.
- [8] L. D’Antoni, M. Veane, B. Livshits, and D. Molnar. FAST: A transducer-based language for tree manipulation, 2012. MSR Technical Report 2012-123 <http://research.microsoft.com/apps/pubs/default.aspx?id=179252>.
- [9] C. Dwork. The differential privacy frontier. In *6th Theory of Cryptography Conference (TCC)*, 2009.
- [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 *Conference on Operating System Design and Implementation*, 2010.
- [11] A. P. Felt, E. Ha, S. Egelman, A. Haney, E. Chin, and D. Wagner. Android permissions: User attention, comprehension, and behavior. In *Symposium on Usable Privacy and Security (SOUPS)*, 2012.
- [12] W. Garage. OpenCV, 2013. <http://opencv.org/>.
- [13] S. Guha, M. Jain, and V. N. Padmanabhan. Koi: A location-privacy platform for smartphone apps. In *NSDI*, 2012.
- [14] 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 *Conference on Computer and Communications Security*, 2011.
- [15] J. Howell and S. Schechter. What You See is What They Get: Protecting users from unwanted use of microphones, cameras, and other sensors. In *Web 2.0 Security and Privacy, IEEE*, 2010.
- [16] J. Howell and S. Schechter. What you see is what they get: Protecting users from unwanted use of microphones, cameras, and other sensors. In *Web 2.0 Security and Privacy Workshop*, 2010.
- [17] E. Hutchings. Augmented reality lets shoppers see how new furniture would look at home, 2012. <http://www.psfk.com/2012/05/augmented-reality-furniture-app.html>.
- [18] S. Jana, A. Narayanan, and V. Shmatikov. DARKLY: Privacy for perceptual applications. In *IEEE Symposium on Security and Privacy*, 2013.
- [19] J. Krumm. A survey of computational location privacy. *Personal Ubiquitous Computing*, 13(6):391–399, Aug 2009.
- [20] Layar. Layar catalogue, 2013. <http://www.layar.com/layers>.
- [21] Microsoft Research Face SDK Beta. <http://research.microsoft.com/en-us/projects/facesdk/>.
- [22] Microsoft Speech Platform. [http://msdn.microsoft.com/en-us/library/hh361572\(v=office.14\).aspx](http://msdn.microsoft.com/en-us/library/hh361572(v=office.14).aspx).
- [23] R. A. Newcombe, S. Izadi, O. Hilliges, D. Molyneaux, D. Kim, A. J. Davison, P. Kohli, J. Shotton, S. Hodges, and A. Fitzgibbon. KinectFusion: Real-time dense surface mapping and tracking. In *10th IEEE International Symposium on Mixed and Augmented Reality*, 2011.
- [24] M. Poh, D. MacDuff, and R. Picard. Advancements in non-contact, multiparameter physiological measurements using a webcam. *IEEE Trans Biomed Engineering*, 58(1):7–11, 2011.
- [25] Project Glass. <https://plus.google.com/+projectglass/posts>.
- [26] Qualcomm. Augmented Reality SDK, 2011. http://www.qualcomm.com/products_services/augmented_reality.html.
- [27] F. Roesner, T. Kohno, A. Moshchuk, B. Parno, H. J. Wang, and C. Cowan. User-driven access control: Rethinking permission granting in modern operating systems. In *IEEE Symposium on Security and Privacy*, 2011.
- [28] S. Schechter. Depth-limited camera for skype - personal communication, 2012.
- [29] J. Shotton, A. Fitzgibbon, M. Cook, T. Sharp, M. Finocchio, R. Moore, A. Kipman, and A. Blake. Real-time human pose recognition in parts from a single depth image. In *Computer Vision and Pattern Recognition*, June 2011.
- [30] uSample. Instant.ly survey creator, 2013. <http://instant.ly>.
- [31] P. Viola and M. Jones. Robust Real-time Object Detection. In *International Journal of Computer Vision*, 2001.

CacheAudit: A Tool for the Static Analysis of Cache Side Channels

Goran Doychev¹, Dominik Feld², Boris Köpf¹, Laurent Mauborgne¹, and Jan Reineke²

¹IMDEA Software Institute

²Saarland University

Abstract

We present CacheAudit, a versatile framework for the automatic, static analysis of cache side channels. CacheAudit takes as input a program binary and a cache configuration, and it derives formal, quantitative security guarantees for a comprehensive set of side-channel adversaries, namely those based on observing cache states, traces of hits and misses, and execution times.

Our technical contributions include novel abstractions to efficiently compute precise overapproximations of the possible side-channel observations for each of these adversaries. These approximations then yield upper bounds on the information that is revealed. In case studies we apply CacheAudit to binary executables of algorithms for symmetric encryption and sorting, obtaining the first formal proofs of security for implementations with countermeasures such as preloading and data-independent memory access patterns.

1 Introduction

Side-channel attacks recover secret inputs to programs from non-functional characteristics of computations, such as time [31], power [32], or memory consumption [27]. Typical goals of side-channel attacks are the recovery of cryptographic keys and private information about users.

Processor caches are a particularly rich source of side-channels because their behavior can be monitored in various ways, which is demonstrated by three documented classes of side-channel attacks: (1) In *time-based attacks* [31, 10] the adversary monitors the overall execution time of a victim, which is correlated with the number of cache hits and misses during execution. Time-based attacks are especially daunting because they can be carried out remotely over the network [6]. (2) In *access-based attacks* [40, 39, 23] the adversary probes the cache state by timing its own accesses to memory. Access-based attacks require that attacker and victim share the

same hardware platform, which is common in the cloud and has already been exploited [41, 49]. (3) In *trace-based attacks* [5] the adversary monitors the sequence of cache hits and misses. This can be achieved, e.g., by monitoring the CPU’s power consumption and is particularly relevant for embedded systems.

A number of proposals have been made for countering cache-based side-channel attacks. Some proposals focus entirely on modifications of the hardware platform; they either solve the problem for specific algorithms such as AES [2] or require modifications to the platform [46] that are so significant that their rapid adoption seems unlikely. The bulk of proposals rely on controlling the interactions between the software and the hardware layers, either through the operating system [23, 48], the client application [10, 39, 15], or both [29]. Reasoning about these interactions can be tricky and error-prone because it relies on the specifics of the binary code and the microarchitecture.

In this paper we present CacheAudit, a tool for the automatic, static exploration of the interactions of a program with the cache. CacheAudit takes as input a program binary and a cache configuration and delivers formal security guarantees that cover all possible executions of the corresponding system. The security guarantees are quantitative upper bounds on the amount of information that is contained in the side-channel observations of timing-, access-, and trace-based adversaries, respectively. CacheAudit can be used to formally analyze the effect on the leakage of software countermeasures and cache configurations, such as preloading of tables or increasing the cache’s line size. The design of CacheAudit is modular and facilitates extension with any cache model for which efficient abstractions are in place. The current implementation of CacheAudit supports caches with LRU, FIFO, and PLRU replacement strategies.

We demonstrate the scope of CacheAudit in case studies where we analyze the side-channel leakage of representative algorithms for symmetric encryption and sort-

ing. We highlight the following two results: (1) For the reference implementation of the Salsa20 [11] stream cipher (which was designed to be resilient to cache side-channel attacks) CacheAudit can formally prove non-leakage on the basis of the binary executable, for all adversary models and replacement strategies. (2) For a library implementation of AES 128 [3], CacheAudit confirms that the preloading of tables significantly improves the security of the executable: for most adversary models and replacement strategies, we can in fact prove non-leakage of the executable, whenever the tables fit entirely in the cache. However, for access-based adversaries and LRU caches, CacheAudit reports small, non-zero bounds. And indeed, with LRU (in contrast to, e.g., FIFO), the *ordering* of blocks within a cache set reveals information about the victim’s final memory accesses.

On a technical level, we build on the fact that the amount of leaked information corresponds to the number of possible side-channel observations, which can be over-approximated by abstract interpretation¹ and counting techniques [35, 34]. To realize CacheAudit based on this insight, we propose three novel abstract domains (i.e. data structures that approximate properties of the program semantics) that keep track of the observations of access-based, time-based, and trace-based adversaries, respectively. In particular:

1. We propose an abstract domain that tracks relational information about the memory blocks that may be cached. In contrast to existing abstract domains used in worst-case execution time analysis [21], our novel domain can retain analysis precision in the presence of array accesses to unknown positions.

2. We propose an abstract domain that tracks the traces of cache hits and misses that may occur during execution. We use a technique based on prefix trees and hash consing to compactly represent such sets of traces, and to count their number.

3. We propose an abstract domain that tracks the possible execution times of a program. This domain captures timing variations due to control flow and caches by associating hits and misses with their respective latencies and adding the execution time of the respective commands.

We formalize the connection of these domains in an abstract interpretation framework that captures the relationship between microarchitectural state and program code. We use this framework to formally prove the correctness of the derived upper bounds on the leakage to the corresponding side-channel adversaries.

In summary, our main contributions are both theoretical and practical: On a theoretical level, we define novel abstract domains that are suitable for the analysis of cache side channels, for a comprehensive set of

adversaries. On a practical level, we build CacheAudit, the first tool for the automatic, quantitative information-flow analysis of cache side-channels, and we show how it can be used to derive formal security guarantees from binary executables of sorting algorithms and state-of-the-art cryptosystems.

Outline The remainder of the paper is structured as follows. In Section 2, we illustrate the power of CacheAudit on a simple example program. In Section 3 we define the semantics and side channels of programs. We describe the analysis framework, the design of CacheAudit, and the novel abstract domains in Sections 4, 5 and 6, respectively. We present experimental results in Section 7, before we discuss prior work and conclude in Sections 8 and 9. The source code and documentation of CacheAudit are available at

<http://software.imdea.org/cacheaudit>

2 Illustrative Example

In this section, we illustrate on a simple example program the kind of guarantees CacheAudit can derive. Namely, we consider an implementation of BubbleSort that receives its input in an array a of length n . We assume that the contents of a are secret and we aim to deduce how much information a cache side-channel adversary can learn about the relative ordering of the elements of a .

```

1 void BubbleSort(int a[], int n)
2 {
3     int i, j, temp;
4     for (i = 0; i < n - 1; ++i)
5         for (j = 0; j < n - 1 - i; ++j)
6             if (a[j] > a[j+1])
7                 {
8                     temp = a[j+1];
9                     a[j+1] = a[j];
10                    a[j] = temp;
11                }
12 }
```

To begin with, observe that the conditional swap in lines 6–11 is executed exactly $\frac{n(n-1)}{2}$ times. A *trace-based* adversary that can observe, for each instruction, whether it corresponds to a cache hit or a miss is likely to be able to distinguish between the two alternative paths in the conditional swap, hence we expect this adversary to be able to distinguish between $2^{\frac{n(n-1)}{2}}$ execution traces. A *timing-based* adversary who can observe the overall execution time is likely to be able to distinguish between $\frac{n(n-1)}{2} + 1$ possible execution times, corresponding to the number of times the swap has been carried out. For an

¹A theory of sound approximation of program semantics [16]

access-based adversary who can probe the final cache state upon termination, the situation is more subtle: evaluating the guard in line 6 requires accessing both $a[j]$ and $a[j+1]$, which implies that both will be present in the cache when the swap in lines 8–10 is carried out. Assuming we begin with an empty cache, we expect that there is only one possible final cache state.

CacheAudit enables us to perform such analyses (for a particular n) formally and automatically, based on actual x86 binary executables and different cache types. CacheAudit achieves this by tracking compact representations of supersets of possible cache states and traces of hits and misses, and by counting the corresponding number of elements. For the above example, CacheAudit was able to precisely confirm the intuitive bounds, for a selection of several n in $\{2, \dots, 64\}$.

In terms of security, the number of possible observations corresponds to the factor by which the cache observation increases the probability of correctly guessing the secret ordering of inputs. Hence, for $n = 32$ and a uniform distribution on this order (i.e. an initial probability of $\frac{1}{32!} = 3.8 \cdot 10^{-36}$), the bounds derived by CacheAudit imply that the probability of determining the correct input order from the side-channel observation is 1 for a trace-based adversary, $3.7 \cdot 10^{-33}$ for a time-based adversary, and remains $\frac{1}{32!}$ for an access-based adversary.

3 Caches, Programs, and Side Channels

3.1 A Primer on Caches

Caches are fast but small memories that store a subset of the main memory’s contents to bridge the latency gap between the CPU and main memory. To profit from spatial locality and to reduce management overhead, main memory is logically partitioned into a set of *memory blocks* \mathcal{B} . Each block is cached as a whole in a cache line of the same size.

When accessing a memory block, the cache logic has to determine whether the block is stored in the cache (“cache hit”) or not (“cache miss”). To enable an efficient look-up, each block can only be stored in a small number of cache lines. For this purpose, caches are partitioned into equally-sized *cache sets*. The size of a cache set is called the *associativity* k of the cache. There is a function set that determines the cache set a memory block maps to.

Since the cache is much smaller than main memory, a *replacement policy* must decide which memory block to replace upon a cache miss. Usually, replacement policies treat sets independently, so that accesses to one set do not influence replacement decisions in other sets. Well-known replacement policies in this class are least-recently used (LRU), used in vari-

ous Freescale processors such as the MPC603E and the TriCore17xx; pseudo-LRU (PLRU), a cost-efficient variant of LRU, used in the Freescale MPC750 family and multiple Intel microarchitectures; and first-in first-out (FIFO), also known as ROUND ROBIN, used in several ARM and Freescale processors such as the ARM922 and the Freescale MPC7450 family. A more comprehensive overview can be found in [22].

3.2 Programs and Computations

A program $P = (\Sigma, I, F, \mathcal{E}, \mathcal{T})$ consists of the following components:

- Σ - a set of *states*
- $I \subseteq \Sigma$ - a set of *initial states*
- $F \subseteq \Sigma$ - a set of *final states*
- \mathcal{E} - a set of *events*
- $\mathcal{T} \subseteq \Sigma \times \mathcal{E} \times \Sigma$ - a *transition relation*

A *computation* of P is an alternating sequence of states and events $\sigma_0 e_0 \sigma_1 e_1 \dots \sigma_n$ such that $\sigma_0 \in I$ and that for all $i \in \{0, \dots, n-1\}$, $(\sigma_i, e_i, \sigma_{i+1}) \in \mathcal{T}$. The set of all computations of P is its *trace collecting semantics* $Col(P) \subseteq Traces$, where $Traces$ denotes the set of all alternating sequences of states and events. When considering terminating programs, the trace collecting semantics can be formally defined as the least fixpoint of the *next* operator containing I :

$$Col(P) = I \cup next(I) \cup next^2(I) \cup \dots ,$$

where *next* describes the effect of one computation step:

$$next(S) = \{t. \sigma_n e_n \sigma_{n+1} \mid t. \sigma_n \in S \wedge (\sigma_n, e_n, \sigma_{n+1}) \in \mathcal{T}\}$$

In the rest of the paper, we assume that P is fixed and abbreviate its trace collecting semantics by Col .

3.3 Cache Updates and Cache Effects

For reasoning about cache side channels, we consider a semantics in which the cache is part of the program state. Namely, the state will consist of logical memories in \mathcal{M} (representing the values of main memory locations and CPU registers, including the program counter) and a cache state in \mathcal{C} , i.e., $\Sigma = \mathcal{M} \times \mathcal{C}$.

The *memory update* $upd_{\mathcal{M}}$ is a function $upd_{\mathcal{M}}: \mathcal{M} \rightarrow \mathcal{M}$ that is determined solely by the instruction set semantics. The memory update has effects on the cache that are described by a function $eff_{\mathcal{M}}: \mathcal{M} \rightarrow \mathcal{E}_{\mathcal{M}}$. The memory effect is an argument to the *cache update* function $upd_{\mathcal{C}}: \mathcal{C} \times \mathcal{E}_{\mathcal{M}} \rightarrow \mathcal{C}$.

In the setting of this paper, $eff_{\mathcal{M}}$ determines which block of main memory is accessed, which is required to compute the cache update $upd_{\mathcal{C}}$, i.e., $\mathcal{E}_{\mathcal{M}} = \mathcal{B} \cup \{\perp\}$, where \perp denotes that no memory block is accessed.

We formally describe upd_C only for the LRU strategy. For formalizations of other strategies, see [22]. Upon a cache miss, LRU replaces the least-recently-used memory block. To this end, it tracks the ages of memory blocks within each cache set, where the youngest block has age 0 and the oldest cached block has age $k - 1$. Thus, the state of the cache can be modeled as a function that assigns an age to each memory block, where non-cached blocks are assigned age k :

$$\mathcal{C} := \{c \in \mathcal{B} \rightarrow A \mid \forall a, b \in \mathcal{B}: a \neq b \Rightarrow ((set(a) = set(b)) \Rightarrow (c(a) \neq c(b) \vee c(a) = c(b) = k))\},$$

where $A := \{0, \dots, k-1, k\}$ is the set of ages. The constraint encodes that no two blocks in the same cache set can have the same age. For readability we omit the additional constraint that blocks of non-zero age are preceded by other blocks, i.e. that cache sets do not contain ‘holes’.

The cache update for LRU is then given by

$$upd_C(c, b) := \lambda b' \in \mathcal{B}. \begin{cases} 0 & : b' = b \\ c(b') & : set(b') \neq set(b) \\ c(b') + 1 & : set(b') = set(b) \wedge c(b') < c(b) \\ c(b') & : set(b') = set(b) \wedge c(b') \geq c(b) \end{cases}$$

In the setting of this paper, the events \mathcal{E} consist of cache hits and misses, which are described by the cache effect $eff_C: \mathcal{C} \times \mathcal{B} \rightarrow \mathcal{E}$:

$$eff_C(c, m) := \begin{cases} hit & : c(m) < k \\ miss & : \text{else} \end{cases}$$

Both upd_C and eff_C are naturally extended to the case where no memory access occurs. Then, the cache state remains unchanged and the cache effect is \perp , so $\mathcal{E} = \{hit, miss, \perp\}$.

With this, we can now connect the components and obtain the global transition relation $\mathcal{T} \subseteq \Sigma \times \mathcal{E} \times \Sigma$ by

$$\begin{aligned} \mathcal{T} = \{((m_1, c_1), e, (m_2, c_2)) \mid & m_2 = upd_M(m_1) \\ & \wedge c_2 = upd_C(c_1, eff_M(m_1)) \\ & \wedge e = eff_C(c_1, eff_M(m_1))\}, \end{aligned}$$

which formally captures the asymmetric relationship between caches, logical memories, and events.

3.4 Side Channels

For a deterministic, terminating program P , the transition relation is a function, and the program can be modeled as a mapping $P: I \rightarrow Col$.

We model an adversary’s view on the computations of P as a function $view: Col \rightarrow O$ that maps computations to a finite set of observations O . The composition

$$C = (view \circ P): I \rightarrow O$$

defines a function from initial states to observations, which we call a *channel* of P . Whenever $view$ is determined by the cache and event components of traces, we call C a *side channel* of P .

We next define views corresponding to the observations of access-based, trace-based, and timing-based side-channel adversaries.

The view of an *access-based* adversary that shares the memory space with the victim is defined by

$$view^{acc}: (m_0, c_0)e_0 \dots e_{n-1}(m_n, c_n) \mapsto c_n$$

and captures that the adversary can determine (by probing) which memory blocks are contained in the cache upon termination of the victim. An adversary that does *not* share the memory space with the victim can only observe how many blocks the victim has loaded in each cache set (by probing how many of its own blocks have been evicted), but not which. We denote this view by $view^{accd}$. The view of a *trace-based* adversary is defined by

$$view^{tr}: \sigma_0 e_0 \dots e_{n-1} \sigma_n \mapsto e_0 \dots e_{n-1}$$

and captures that the adversary can determine for each instruction whether it results in a hit, miss, or does not access memory. The view of a *time-based* adversary is defined by

$$\begin{aligned} view^{time}: \sigma_0 e_0 \dots e_{n-1} \sigma_n \mapsto \\ t_{hit} \cdot |\{i \mid e_i = hit\}| + t_{miss} \cdot |\{i \mid e_i = miss\}| + \\ t_{\perp} \cdot |\{i \mid e_i = \perp\}| \end{aligned}$$

and captures that the adversary can determine the overall execution time of the program. Here, t_{hit} , t_{miss} , and t_{\perp} are the execution times (e.g. in clock cycles) of instructions that imply cache hits, cache misses, or no memory accesses at all. While the view of the time-based adversary as defined above is rather simplistic, e.g. disregarding effects of pipelining and out-of-order execution, notice that our semantics and our tool can be extended to cater for a more fine-grained, instruction- and context-dependent modeling of execution times. We denote the side channels corresponding to the four views by C^{acc} , C^{accd} , C^{tr} , and C^{time} , respectively. Figure 1 gives an overview.

3.5 Quantification of Side Channels

We characterize the security of a channel $C: I \rightarrow O$ as the difficulty of guessing the secret input from the channel output.

C^{acc}	Access-based adversary whose memory space is shared with the victim's.
C^{accd}	Access-based adversary whose memory space is disjoint from the victim's.
C^{tr}	Adversary who observes the trace of cache hits and misses.
C^{time}	Adversary who observes the overall execution time.

Figure 1: Channels corresponding to different adversary models.

Formally, we model the choice of a secret input by a random variable X with $\text{ran}(X) \subseteq I$ and the corresponding observation by a random variable $C(X)$ with $\text{ran}(C(X)) \subseteq O$. We model the attacker as another random variable \hat{X} . The goal of the attacker is to estimate the value of X , i.e. it is successful if $\hat{X} = X$. We make the assumption that the attacker does not have information about the value of X beyond what is contained in $C(X)$, which we formalize as the requirement that $X \rightarrow C(X) \rightarrow \hat{X}$ form a Markov chain. The following theorem expresses a security guarantee as an upper bound on the attacker's success probability in terms of the size of the range of C .

Theorem 1. *Let $X \rightarrow C(X) \rightarrow \hat{X}$ be a Markov chain. Then*

$$P(X = \hat{X}) \leq \max_{\sigma \in I} P(X = \sigma) \cdot |\text{ran}(C)|$$

For the interpretation of the statement observe that if the adversary has no information about the value of X (i.e., if \hat{X} and X are statistically independent), its success probability is bounded by the probability of the most likely value of X , i.e. $P(X = \hat{X}) \leq \max_{\sigma \in I} P(X = \sigma)$, where equality can be achieved. Theorem 1 hence states that the size of the range of C is an upper bound on the factor by which this probability is increased when the attacker sees $C(X)$ and is, in that sense, an upper bound for the amount of information leaked by C . We will often give bounds on $|\text{ran}(C)|$ on a log-scale, in which case they represent upper bounds on the number of leaked bits. Notice that the guarantees of Theorem 1 fundamentally rely on assumptions about the initial distribution of X : if X is easy to guess to begin with, Theorem 1 does not imply meaningful security guarantees.

For more discussion on the interpretation of the security guarantees, see Section 7.4. For a formal connection to traditional (entropy-based) presentations of quantitative information-flow analysis [43] and a proof of Theorem 1, see the extended version [19].

3.6 Adversarially Chosen Cache States

We sometimes assume that initial states are pairs consisting of *high* and *low* components, i.e. $I = I_{hi} \times I_{lo}$, where only the high component is meant to be kept secret and the low component may be provided by the adversary, a common setting in information-flow analysis [42]. In this case, a program and a view define a *family* of channels $C_{\sigma_{lo}} : I_{hi} \rightarrow O$, one for each low component $\sigma_{lo} \in I_{lo}$.

A particularly interesting instance is the decomposition into secret memory $I_{hi} = \mathcal{M}$ and adversarially chosen cache $I_{lo} = \mathcal{C}$. While bounds for the corresponding channel can be derived by considering all possible initial cache states, corresponding analyses suffer from poor precision. The following lemma enables us to derive bounds for the general case, based on the empty cache state.

Lemma 1. *For all initial cache states $c \in \mathcal{C}$, adversaries $adv \in \{acc, accd, time, tr\}$, and LRU, FIFO, or PLRU replacement: If no block in c is accessed during program execution, then*

$$|\text{ran}(C_{\emptyset}^{adv})| = |\text{ran}(C_c^{adv})|, \quad (1)$$

where \emptyset is a shorthand for the empty cache state. For $adv \in \{acc, accd\}$ and LRU, $|\text{ran}(C_{\emptyset}^{adv})| \geq |\text{ran}(C_c^{adv})|$ holds without any constraints on the initial cache state c .

This lemma was proved in [34] for *acc*, *accd* and the LRU case with the initial cache state not containing any block of the victim. The proof is based on the fact that memory blocks in the cache do not affect the position of memory blocks that are accessed during computation whenever the two sets of memory blocks are disjoint, which allows us to construct a bijective function from $\text{ran}(C_{\emptyset}^{adv})$ to $\text{ran}(C_c^{adv})$. The argument immediately extends to FIFO, PLRU, and all *adv*. For LRU and access-based adversaries, the function remains surjective even without the disjointness requirement.

4 Automatic Quantification of Cache Side Channels

Theorem 1 enables the quantification of side channels by determining their range. As channels are defined in terms of views on computations, their range can be determined by computing *Col* and applying *view*. However, this entails computing a fixpoint of the *next* operator and is practically infeasible in most cases. Abstract interpretation [16] overcomes this fundamental problem by computing a fixpoint with respect to an efficiently computable over-approximation of *next*. This new fixpoint represents a superset of all computations, which is sufficient for deriving an upper bound on the range of the channel and thus on the leaked information.

In this section, we describe the interplay of the abstractions used for over-approximating next in CacheAudit (namely, those for memory, cache, and events), and we explain how the global soundness of CacheAudit can be established from local soundness conditions. This modularity is key for the future extension of CacheAudit using more advanced abstractions. Our results hold for all adversaries introduced in Section 3.4 and we omit the superscript adv from channels and views for readability.

4.1 Sound Abstraction of Leakage

We frame a static analysis by defining a set of abstract elements Traces^\sharp together with an abstract transfer function $\text{next}^\sharp : \text{Traces}^\sharp \rightarrow \text{Traces}^\sharp$. Here, the elements $a \in \text{Traces}^\sharp$ represent subsets of Traces , which is formalized by a concretization function

$$\gamma : \text{Traces}^\sharp \rightarrow \mathcal{P}(\text{Traces}) .$$

The key requirements for next^\sharp are (1) that it be efficiently computable, and (2) that it over-approximates the effect of next on sets of computations, which is formalized as the following local soundness condition:

$$\forall a \in \text{Traces}^\sharp : \text{next}(\gamma(a)) \subseteq \gamma(\text{next}^\sharp(a)) . \quad (2)$$

Intuitively, if we maintain a superset of the set of computations during each step of the transfer function as in (2), then this inclusion must also hold for the corresponding fixpoints. More formally, any post-fixpoint of next^\sharp that is greater than an abstraction of the initial states I is a sound over-approximation of the collecting semantics. We use Col^\sharp to denote any such post-fixpoint.

Theorem 2 (Local soundness implies global soundness, from [16]). *If (2) holds then*

$$\text{Col} \subseteq \gamma(\text{Col}^\sharp) .$$

The following theorem is an immediate consequence of Theorem 2 and the fact that $\text{view}(\text{Col}) = \text{ran}(C)$. It states that a sound abstract analysis can be used for deriving bounds on the size of the range of a channel.

Theorem 3 (Upper bounds on leakage).

$$|\text{ran}(C)| \leq |\text{view}(\gamma(\text{Col}^\sharp))| .$$

With the help of Theorem 1, these bounds immediately translate into security guarantees. The relationship of all steps leading to these guarantees is depicted in Figure 2.

4.2 Abstraction Using a Control Flow Graph

In order to come up with a tractable and modular analysis, we design independent abstractions for cache states, memory, and sequences of events.

- \mathcal{M}^\sharp abstracts memory and $\gamma_{\mathcal{M}} : \mathcal{M}^\sharp \rightarrow \mathcal{P}(\mathcal{M})$ formalizes its meaning.
- \mathcal{C}^\sharp abstracts cache configurations and $\gamma_{\mathcal{C}} : \mathcal{C}^\sharp \rightarrow \mathcal{P}(\mathcal{C})$ formalizes its meaning.
- \mathcal{E}^\sharp abstracts sequences of events and $\gamma_{\mathcal{E}} : \mathcal{E}^\sharp \rightarrow \mathcal{P}(\mathcal{E}^*)$ formalizes its meaning.

But, since cache updates and events depend on memory state, independent analyses would be too imprecise. In order to maintain some of the relations, we link the three abstract domains for memory state, caches, and events through a finite set of labels L so that our abstract domain is

$$\text{Traces}^\sharp = L \rightarrow \mathcal{M}^\sharp \times \mathcal{C}^\sharp \times \mathcal{E}^\sharp ,$$

where we write $a^{\mathcal{M}}(l)$, $a^{\mathcal{C}}(l)$ and $a^{\mathcal{E}}(l)$ for the first, second, and third components of an abstract element $a(l)$.

Labels roughly correspond to nodes in a control flow graph in classical data-flow analyses. One could simply use program locations as labels. But in our setting, we use more general labels, allowing for a more fine-grained analysis in which we can distinguish values of flags or results of previous tests [36]. To capture that, we associate a meaning with each label via a function $\gamma_L : L \rightarrow \mathcal{P}(\text{Traces})$. If the labels are program locations, then $\gamma_L(l)$ is the set of traces ending in a state in location l . The analogy with control flow graphs can be extended to edges of that graph: using the next operator, we define the successors and predecessors of a location l as: $\text{succ}(l) = \{k \mid \text{next}(\gamma_L(l)) \cap \gamma_L(k) \neq \emptyset\}$, and $\text{pred}(l) = \{k \mid \text{next}(\gamma_L(k)) \cap \gamma_L(l) \neq \emptyset\}$.

Then we can describe the meaning of an element $a \in \text{Traces}^\sharp$ with:

$$\begin{aligned} \gamma(a) = & \{ \sigma_0 e_0 \sigma_1 \dots \sigma_n \in \text{Traces} \mid \forall i \leq n, \forall l \in L : \\ & \sigma_0 e_0 \sigma_1 \dots \sigma_i \in \gamma_L(l) \Rightarrow \\ & \sigma_i^{\mathcal{M}} \in \gamma_{\mathcal{M}}(a^{\mathcal{M}}(l)) \wedge \sigma_i^{\mathcal{C}} \in \gamma_{\mathcal{C}}(a^{\mathcal{C}}(l)) \\ & \wedge e_0 \dots e_{i-1} \in \gamma_{\mathcal{E}}(a^{\mathcal{E}}(l)) \} \end{aligned} \quad (3)$$

That is, the meaning of an $a \in \text{Traces}^\sharp$ is the set of traces, such that for every prefix of a trace, if it “ends” at program location l , then the memory state, cache state, and the event sequence satisfy the respective abstract elements for that location.

The abstract transfer function next^\sharp will be decomposed into:

$$\text{next}^\sharp(a) = \lambda l. (\text{next}_{\mathcal{M}^\sharp}(a, l), \text{next}_{\mathcal{C}^\sharp}(a, l), \text{next}_{\mathcal{E}^\sharp}(a, l)) , \quad (4)$$

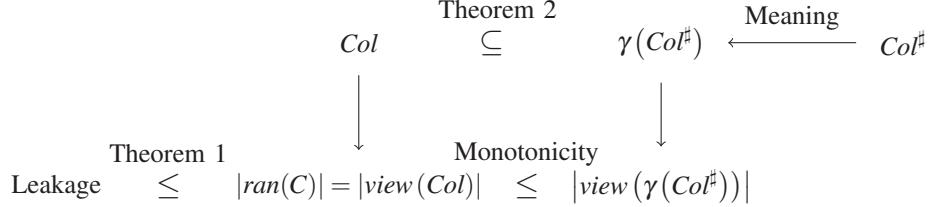


Figure 2: Relationship of collecting semantics Col , abstract fixpoint Col^\sharp , side channels C , and leakage bounds.

where each next function over-approximates the corresponding concrete update function defined in the previous section. The effects used for defining the concrete updates are reflected as information flow between otherwise independent abstract domains, which is formalized as a *partial reduction* in the abstract interpretation literature [18].

4.3 Local Soundness

The products and powers of sound abstract domains with partial reductions are again sound abstract domains [17]. The soundness of Traces^\sharp hence immediately follows from the local soundness of the memory, cache and event domains. Below we describe those soundness conditions for each domain.

The abstract next^\sharp operation is implemented using local update functions for the memory, cache, and event components. For the memory domain we have, for each label $k \in L$ and each $l \in \text{succ}(k)$:

- an abstract memory update $\text{upd}_{\mathcal{M}^\sharp, (k, l)} : \mathcal{M}^\sharp \rightarrow \mathcal{M}^\sharp$, and
- an abstract memory effect $\text{eff}_{\mathcal{M}^\sharp, (k, l)} : \mathcal{M}^\sharp \rightarrow \mathcal{P}(\mathcal{E}_\mathcal{M})$.

For the cache domain, there is no need for separate functions for each pair (k, l) , because the cache update only depends on the accessed block which is delivered by the abstract memory effect. Likewise, the update of the event domain only depends on the abstract cache effect. Thus, we further have:

- an abstract cache update $\text{upd}_{\mathcal{C}^\sharp} : \mathcal{C}^\sharp \times \mathcal{P}(\mathcal{E}_\mathcal{M}) \rightarrow \mathcal{C}^\sharp$,
- an abstract cache effect $\text{eff}_{\mathcal{C}^\sharp} : \mathcal{C}^\sharp \times \mathcal{P}(\mathcal{E}_\mathcal{M}) \rightarrow \mathcal{P}(\mathcal{E}_\mathcal{C})$, and
- an abstract event $\text{upd}_{\mathcal{E}^\sharp} : \mathcal{E}^\sharp \times \mathcal{P}(\mathcal{E}_\mathcal{C}) \rightarrow \mathcal{E}^\sharp$.

With these functions, we can approximate the effect of next on each label l , using the abstract values associated with the labels that can lead to l , $\text{pred}(l)$. For the example of the cache domain, this yields

$$\text{next}_{\mathcal{C}^\sharp}(a, l) = \bigsqcup_{k \in \text{pred}(l)}^{\mathcal{C}^\sharp} \text{upd}_{\mathcal{C}^\sharp}\left(a^{\mathcal{C}}(k), \text{eff}_{\mathcal{M}^\sharp, (k, l)}(a^{\mathcal{M}}(k))\right),$$

where $\bigsqcup^{\mathcal{C}^\sharp}$ refers to the join function and can be thought of as set union. That is, $\text{next}_{\mathcal{C}^\sharp}(a, l)$ collects all cache states that can reach l within one transition when updated with an over-approximation of the corresponding memory blocks. See the full version [19] for a description of the corresponding update functions for memory and effects.

Now from Equations 2, 3, and 4, we can derive conditions for each domain that are sufficient to guarantee local soundness for the whole analysis:

Definition 1 (Local soundness of abstract domains). *The abstract domains are locally sound if the abstract joins are over-approximations of unions, and if for any function $f^\sharp \in \{\text{upd}_{\mathcal{M}^\sharp, (k, l)}, \text{eff}_{\mathcal{M}^\sharp, (k, l)}, \text{upd}_{\mathcal{C}^\sharp}, \text{eff}_{\mathcal{C}^\sharp}, \text{upd}_{\mathcal{E}^\sharp}\}$ approximating concrete function $f \in \{\text{upd}_\mathcal{M}, \text{eff}_\mathcal{M}, \text{upd}_\mathcal{C}, \text{eff}_\mathcal{C}, \text{next}\}$ and corresponding meaning function γ_f , we have for any abstract value x :*

$$\gamma_f\left(f^\sharp(x)\right) \supseteq f\left(\gamma_f(x)\right).$$

For example, for the cache abstract domain, we have the following local soundness conditions:

$$\forall c^\sharp \in \mathcal{C}^\sharp, M \in \mathcal{P}(\mathcal{E}_\mathcal{M}) :$$

$$\begin{aligned} \gamma_{\mathcal{C}}(\text{upd}_{\mathcal{C}^\sharp}(c^\sharp, M)) &\supseteq \text{upd}_{\mathcal{C}}(\gamma_{\mathcal{C}}(c^\sharp), M), \\ \text{eff}_{\mathcal{C}^\sharp}(c^\sharp, M) &\supseteq \text{eff}_{\mathcal{C}}(\gamma_{\mathcal{C}}(c^\sharp), M), \end{aligned}$$

$$\forall G^\sharp \subseteq \mathcal{C}^\sharp : \gamma_{\mathcal{C}}\left(\bigsqcup^{c^\sharp} G^\sharp\right) \supseteq \bigcup_{G^\sharp \in \mathcal{G}^\sharp} \gamma_{\mathcal{C}}(G^\sharp).$$

Lemma 2 (Local Soundness Conditions). *If local soundness holds on the abstract memory, cache, and events domains, then the corresponding next^\sharp function satisfies local soundness.*

Due to the above lemma, abstract domains for the memory, cache, and events can be separately developed and proven correct. We exploit this fact in this paper, and we plan to develop further abstractions in the future, targeting different classes of adversaries or improving precision.

4.4 Soundness of Delivered Bounds

We implemented the framework described above in a tool named CacheAudit. Thanks to the previous results, CacheAudit provides the following guarantees.

Theorem 4. *The bounds derived by CacheAudit soundly over-approximate $|\text{ran}(C^{\text{adv}})|$, for $\text{adv} \in \{\text{acc}, \text{accd}, \text{tr}, \text{time}\}$, and hence correspond to upper bounds on the maximal amount of leaked information.*

The statement is an immediate consequence of combining Lemma 2 with Theorems 2 and 3, under the assumption that all involved abstract domains satisfy local soundness conditions, and that the corresponding counting procedures are correct. We formally prove the validity of these assumptions only for our novel relational and trace domains (see Section 6). For the other domains, corresponding proofs are either standard (e.g. the value domain) or out of scope of this submission.

5 Tool Design and Implementation

In this section we describe the architecture and implementation of CacheAudit.

We take advantage of the compositionality of the framework described in Section 4 and use a generic iterator module to compute fixpoints, where we rely on independent modules for the abstract domains that correspond to the components of the next^\sharp operation. Figure 3 depicts the overall architecture of CacheAudit, with the individual modules described below.

5.1 Control Flow Reconstruction

The first stage of the analysis is similar to a compiler front end. The main challenge is that we directly analyze x86 executables with no explicit control flow graph, which we need for guiding the fixpoint computation.

For the parsing phase, we rely on Chlipala’s parser for x86 executables [13], which we extend to a set of instructions that is sufficient for our case studies (but not yet complete). For the control-flow reconstruction, we consider only programs without dynamically computed jump and call targets, which is why it suffices to identify the basic blocks and link them according to the corresponding branching conditions and (static) branch targets. We plan to integrate more sophisticated techniques for control-flow reconstruction [30] in the future.

5.2 Iterator

The iterator module is responsible for the computation of the next^\sharp operator and of the approximation of its fixpoint using adequate iteration strategies [17]. Our analysis uses an *iterative* strategy, i.e., it stabilizes components

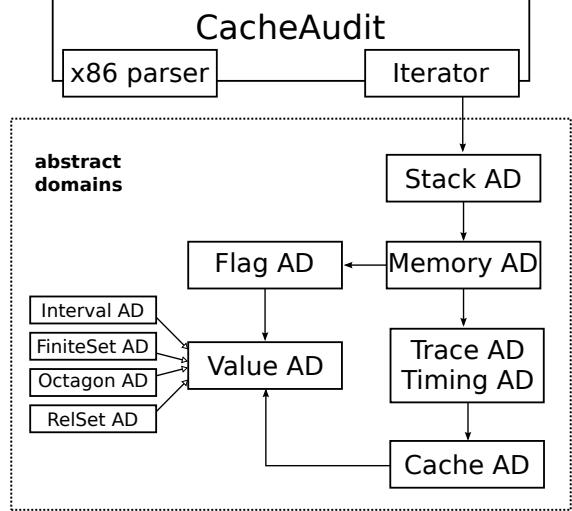


Figure 3: The architecture of CacheAudit. The solid boxes represent modules. Black-headed arrows mean that the module at the head is an argument of the module at the tail. White-headed arrows represent is-a relationships.

of the abstract control flow graph according to a weak topological ordering, which we compute using Bourdoncle’s algorithm [12].

The iterator also implements parts of the reduced cardinal power, based on the labels computed according to the control-flow graph: Each label is associated with an initial abstract state. The analysis computes the effect of the commands executed from that label to its successors on the initial abstract state, and propagates the resulting final states using the abstract domains described below. In order to increase precision, we expand locations using loop unfolding, so that we have a number of different initial and final abstract states for each label inside loops, depending on a parameter describing the number of loop unfoldings we want to perform. Most of our examples (such as the cryptographic algorithms) require only a small, constant number of loop iterations, so that we can choose unfolding parameters that avoid joining states stemming from different iterations.

5.3 Abstract Domains

As described in Section 4, we decompose the abstract domain used by the iterator into mostly independent abstract domains describing different aspects of the concrete semantics.

Value Abstract Domains A value abstract domain represents sets of mappings from variables to (integer)

values. Value abstract domains are used by the cache abstract domain to represent ages of blocks in the cache, and by the flag abstract domain to represent values stored at the addresses used in the program. We have implemented different value abstract domains, such as the interval domain, an exact finite sets domain (where the sets become intervals when they are growing too large) and a relational set domain (as described in Section 6.1).

Flag Abstract Domain In x86 binaries, there are no high level guards: instead, most operations modify flags which are then queried in conditional branches. In order to deal precisely with such branches, we need to record relational information between the values of variables and the values of these flags. To that end, for each operation that modifies the flags, we compute an over-approximation of the values of the arguments that may lead to a particular flag combination. The flag abstract domain represents an abstract state as a mapping from values of flags to elements of the value abstract domain. When the analysis reaches a conditional branch, it can identify which combination of flag values corresponds to the branch and propagate the appropriate abstract values.

Memory Abstract Domain The memory abstract domain associates memory addresses and registers with variables and translates machine instructions into the corresponding operations on those variables, which are represented using flag abstract domains as described above. One important aspect for efficiency is that variables corresponding to addresses are created dynamically during the analysis whenever they are needed. The memory abstract domain further records all accesses to main memory using a cache abstract domain, as described below.

Stack Abstract Domain Operations on the stack are handled by a dedicated stack abstract domain. In this way the memory abstract domain does not have to deal with stack operations such as procedure calls, for which special techniques can be implemented to achieve precise interprocedural analysis.

Cache Abstract Domain The cache abstract domain only tracks information about the cache state. We represent this state by sets of mappings from blocks to ages in the cache, which we implement using an instance of value abstract domains. Effects from the memory domain are passed to the cache domain through the trace domain. The cache abstract domain tracks which addresses are touched during computation and returns information about the presence or absence of cache hits and misses to the trace abstract domain, which we

present in Section 6.2. The timings are then obtained as an abstraction from the traces.

6 Abstract Domains for Cache Adversaries

6.1 Cache State Domains

Abstractions of cache states are at the heart of analyses for all three cache adversaries considered in this paper. Thus, precise abstraction of cache states is crucial to determine tight leakage bounds.

The current state-of-the-art abstraction for LRU replacement by Ferdinand et al. [21] maintains an upper and a lower bound on the age of every memory block. This abstraction was developed with the sole goal of classifying memory accesses as cache hits or cache misses. In contrast, our goal is to develop abstractions that yield tight bounds on the maximal leakage of a channel. For access-based adversaries the leakage is bounded by the size of the concretization of an abstract cache state, i.e. the size of the set of concrete cache states represented by the abstract state.

Intuition behind Relational Sets To derive tighter leakage bounds, we propose a new domain called *relational sets* that improves previous work along two dimensions:

1. Instead of *intervals* of ages of memory blocks, we maintain *sets* of ages of memory blocks.
2. Instead of maintaining *independent* information about the age of each memory blocks, we record the *relation* between ages of different memory blocks.

In addition to increasing precision, moving from intervals to sets allows us to analyze caches with FIFO and PLRU replacement. Interval-based analysis of FIFO and PLRU has been shown to be rather imprecise in the context of worst-case execution time analysis [24].

Motivating Example Consider the following method, which performs a table lookup based on a secret input, as it may occur in e.g. an AES implementation:

```
unsigned int A[size];

int getElement(int secret) {
    if (secret < size)
        return A[secret];
}
```

Assume we want to determine the possible cache states after one invocation of `getElement`. As the value of `secret` is unknown to the analysis, every memory location of the array might be accessed.

size	8	16	32	64	128	256
LRU/IV	1	2	4	8	16	32
LRU/Set	1	2	4	8	16	32
LRU/Rel	1	1.58	2.32	3.17	4.01	5.04

Figure 4: Bounds on the number of leaked bits about the parameter *secret* for varying array sizes. The cache parameters are fixed, with a block size of 32 bytes, associativity 4 and cache size 4 KB.

Assuming the array was not cached before the invocation of `getElement`, the interval-based domain by Ferdinand et al. [21] determines a lower bound of 0 and an upper bound of k on the age of each array element.

By tracking sets instead of intervals of ages for each memory block, we would get 0 and k as possible ages of each array element.

Both non-relational domains, however, are not powerful enough to infer or even express the fact, that *only one* of the array’s memory blocks has been accessed, and can thus be cached. Therefore, the number of possible cache states represented by non-relational abstractions grows exponentially in the size of the array, while the actual number of possible cache states only grows linearly.

A relational domain, tracking the possible ages of, e.g., pairs of memory blocks, would indeed yield a linear growth in the number of possible cache states. For each pair of array elements, it would be able to infer that only one of the two blocks may be cached. From this, it follows that only one of all of the array elements may be cached.

Figure 4 shows experimental results for the example program with three domains: the interval domain (*IV*), and two instances of the *relational sets* domain, tracking sets of ages of individual blocks (*Set*) and sets of ages of pairs of blocks (*Rel*), respectively.

We do not see an improvement of *sets* over *intervals* in this particular example, as the information that a block has either age 0 or age k can be inferred from the intervals in the counting procedure. This is because the considered arrays are small and thus no two array elements map to the same cache set. We have, however, observed in case studies that *sets* alone often improve over *intervals*.

A detailed formalization of relational sets and their implementation, including efficient counting, is provided in the extended version of this paper [19]. There, we also show that the domain is locally sound according to Definition 1:

Lemma 3. *The relational sets domain is locally sound.*

6.2 A Trace Domain

We devise an abstract domain for keeping track of the sets of event traces that may occur during the execution of a program. Following the way events are computed in the concrete, namely as a function from cache states and memory effects (see Section 3.3), the abstract cache domain provides abstract cache effects.

In our current implementation of CacheAudit, we use an exact representation for sets of event traces: we can represent any finite set of event traces, and assuming an incoming set of traces \mathcal{S} and a set of cache effects E , we compute the resulting event set precisely as follows:

$$upd_{\mathcal{E}^\sharp}(\mathcal{S}, E) = \{\sigma.e \mid \sigma \in \mathcal{S} \wedge e \in E\}$$

Then soundness is obvious, since the abstract operation is the same as its concrete counterpart. Due to loop unfolding, we do not require widenings, even though the domain contains infinite ascending chains (see Section 5.2).

Lemma 4. *The trace domain is locally sound.*

Representation for Sets of Event Traces We represent sets of finite event traces corresponding to a particular program location by a directed acyclic graph (DAG) with vertices V , a dedicated root $r \in V$, and a node labeling $\ell: V \rightarrow \mathcal{P}(\mathcal{E}) \cup \{\sqcup\}$. In this graph, every node $v \in V$ represents a set of traces $\gamma(v) \in \mathcal{P}(\mathcal{E}^*)$ in the following way:

1. For the root r , $\gamma(r) = \{\varepsilon\}$
2. For v with $L(v) = \sqcup$ and predecessors u_1, \dots, u_n , $\gamma(v) = \bigcup_{i=1}^n \gamma(u_i)$.
3. For v with $L(v) \neq \sqcup$ and predecessors u_1, \dots, u_n , $\gamma(v) = \{t.u \mid u \in L(v) \wedge t \in \bigcup_{i=1}^n \gamma(u_i)\}$

Intuitively, every $v \in V$ represents a set of event traces, namely the sequences of labels of paths from r to v .

In the context of CacheAudit, we need to implement two operations on this data structure, namely (1) the join $\sqcup^{\mathcal{E}^\sharp}$ of two sets of traces and the (2) addition $upd_{\mathcal{E}^\sharp}(\mathcal{S}, E)$ of a cache event to a particular set of traces.

For the join of two sets of traces represented by v and w , we add a new vertex u with label \sqcup and add edges from v and w to u .

For the extension of a set of traces represented by a vertex v by a set of cache events E , we first check whether v already has a child w labeled with E . If so, we use w as a representation of the extended set of traces. If not, we add a new vertex u with label E and add an edge (u, v) . In this way we make maximal use of sharing and obtain a prefix DAG. The correctness of the representation follows by construction. In CacheAudit, we use hash consing for efficiently building the prefix DAG.

Counting Sets of Traces The following algorithm $count_{tr}$ overapproximates the number of traces that are represented by a given graph.

1. For the root r , $count_{tr}(r) = 1$
2. For v with $L(v) = \square$ and predecessors u_1, \dots, u_n , $count_{tr}(v) = \sum_{i=1}^n count_{tr}(u_i)$
3. For v with $L(v) \neq \square$ and predecessors u_1, \dots, u_n , $count_{tr}(v) = |L(v)| \cdot \sum_{i=1}^n count_{tr}(u_i)$

The soundness of this counting, i.e. the fact that $|L(v)| \leq count_{tr}(v)$, follows by construction. Notice that the precision dramatically decreases with larger sets of labels. In our case, labels contain at most three events and the counting is sufficiently precise.

Counting Timing Variations We currently model execution time as a simple abstraction of traces, see Section 3. In particular, timing is computed from a trace over $\mathcal{E} = \{hit, miss, \perp\}$ by multiplying the number of occurrences of each event by the time they consume: t_{hit} , t_{miss} , and t_{\perp} , respectively. The following algorithm $count_{time}$ over-approximates the set of timing behaviors that are represented by a given graph.

1. For the root r , $count_{time}(r) = \{0\}$
2. For v with $L(v) = \square$ and predecessors u_1, \dots, u_n , $count_{time}(v) = \bigcup_{i=1}^n count_{time}(u_i)$
3. For v with $L(v) \neq \square$ and predecessors u_1, \dots, u_n ,

$$count_{time}(v) = \left\{ t_x + t \mid x \in L(v) \wedge t \in \bigcup_{i=1}^n count_{time}(u_i) \right\}$$

The soundness of $count_{time}$, i.e. the fact that it delivers a superset of the number of possible timing behaviors, follows by construction.

7 Case Studies

In this section we demonstrate the capabilities of CacheAudit in case studies where we use it to analyze the cache side channels of algorithms for sorting and symmetric encryption. All results are based on the automatic analysis of corresponding 32-bit x86 Linux executables that we compiled using `gcc`.

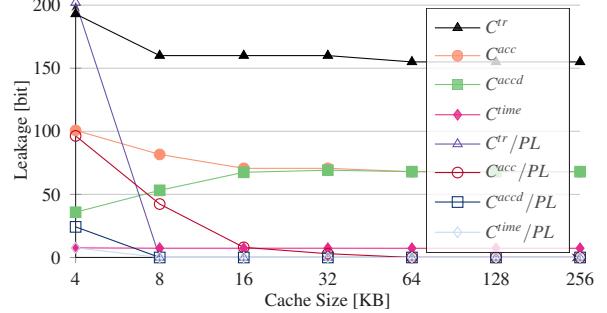


Figure 5: Effect of the attacker model and preloading (PL) on the security guarantee, for varying cache sizes. The results are given for a 4-way set associative cache with a line size of 64B and the LRU replacement strategy.

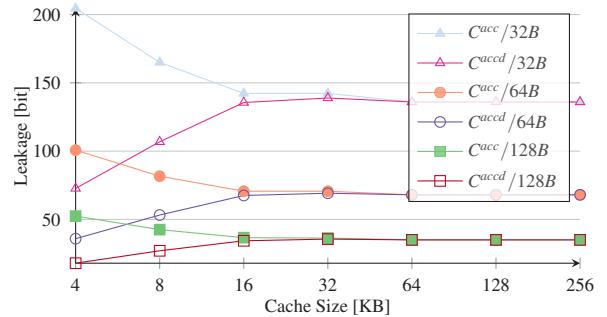


Figure 6: Effect of the cache line size on the security guarantee, for C^{acc} and C^{accd} , for varying cache sizes. The results are given for a 4-way set associative cache with the LRU replacement strategy.

7.1 AES 128

We analyze the AES implementation from the PolarSSL library [3] with keys of 128 bits, where we consider the implementation with and without preloading of tables, for all attacker models, different replacement strategies, associativities, and line sizes. All results are presented as upper bounds of the leakage in bits; for their interpretation see Theorem 1. In some cases, CacheAudit reports upper bounds that exceed the key size (128 bits), which corresponds to an imprecision of the static analysis. We opted against truncating to 128 bits to illustrate the degree of imprecision. The full data of our analysis are given in the extended version of this paper [19]. Here, we highlight some of our findings.

- Preloading almost consistently leads to better security guarantees in all scenarios (see e.g. Figure 5). However, the effect becomes clearly more apparent for cache sizes beyond 8KB, which is explained by the PolarSSL AES tables exceeding the size of the 4KB cache by 256B. For cache sizes that are larger than the preloaded tables, we can prove noninterference for C^{acc} and FIFO, C^{accd} and LRU, and for C^{tr} and C^{time} on LRU, FIFO, and PLRU. For C^{acc} with shared memory spaces and LRU,

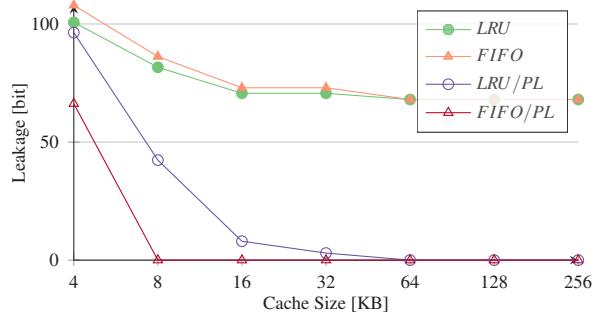


Figure 7: Effect of the replacement strategy on the security guarantee for C^{acc} , with and without preloading (PL), for varying cache sizes. The results are given for a 4-way set associative cache with a line size of 64B.

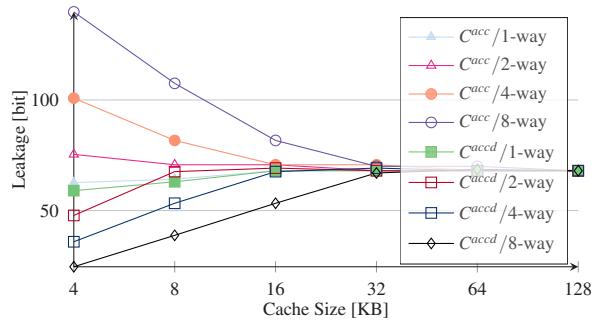


Figure 8: Effect of the associativity on the security guarantee, for C^{acc} and C^{accd} , without preloading, for varying cache sizes. The results are given for a cache with a line size of 64B and the LRU replacement strategy.

this result does *not* hold because the adversary can obtain information about the order of memory blocks in the cache.

- A larger *line size* consistently leads to better security guarantees for access-based adversaries (see e.g. Figure 6). This follows because more array indices map to a line which decreases the resolution of the attacker’s observations.

- In terms of *replacement strategies*, we consistently derive the lowest bounds for LRU, followed by PLRU and FIFO (see the extended version [19]), where the only exception is the case of C^{acc} and preloading (see Figure 7). In this case FIFO is more secure because with LRU the adversary can obtain information about the ordering of memory blocks in the cache.

- In terms of *cache size*, we consistently derive better bounds for larger caches, with the exception of C^{accd} . For this adversary model the bounds increase because larger caches correspond to distributing the table to more sets, which increases its possibilities to observe variations. The guarantees we obtain for C^{accd} and C^{acc} converge for caches of 4 ways and sizes beyond 16KB (see e.g. Figure 6). This is due to the fact that each cache

set can contain at most one unique block of the 4KB table. In that way, the ability to observe ordering of blocks within a set does not give C^{acc} any advantage.

- When increasing *associativity*, we observe opposing effects on the leakage of C^{acc} and C^{accd} (see Figure 8). This is explained by the fact that, for a fixed cache size, increasing associativity means decreasing the number of sets. For C^{accd} which can only observe the number of blocks that have been loaded into each set, this corresponds to a decrease in observational capability; for C^{acc} which can observe the ordering of blocks, this corresponds to an increase. This difference vanishes for larger cache sizes because then each set contains at most one unique block of the AES tables.

Comparison to [34]: In a recent study [34] we analyzed the PolarSSL AES implementation with respect to access-based adversaries and LRU replacement, using the cache component of a closed-source tool for worst-case execution time analysis [1]. The results we obtain using CacheAudit go beyond that analysis in that we derive bounds w.r.t. access-based, trace-based, and time-based adversaries, for LRU, FIFO, and PLRU strategies. For access-based adversaries and LRU, the bounds we derive are lower than those in [34]; in particular, for C^{accd} we derive bounds of zero for implementations with preloading for *all* cache sizes that are larger than the AES tables—which is obtained in [34] only for caches of 128KB. While these results are obtained for different platforms (x86 vs. ARM) and are hence not directly comparable, they do suggest a significant increase in precision. In contrast to [34], this is achieved without any code instrumentation.

7.2 Salsa20

Salsa20 is a stream cipher by Bernstein [11]. Internally, the cipher uses XOR, addition mod 2^{32} , and constant-distance rotation operations on an internal state of 16 32-bit words. The lack of key-dependent memory lookups intends to avoid cache side channels in software implementations of the cipher. With CacheAudit we could formally confirm this intuition by automated analysis of the reference implementation of Salsa20 encryption, which includes a function call to a hash function. Specifically, we analyze the leakage of the encryption operation on an arbitrary 512-byte message for C^{acc} , C^{tr} , and C^{time} , FIFO and LRU strategies, on 4KB caches with line size of 32B, where we consistently obtain upper bounds of 0 for the leakage. The time required for analyzing each of the cases was below 11s.

7.3 Sorting Algorithms

In this section we use CacheAudit to establish bounds on the cache side channels of different sorting algorithms. This case study is inspired by an early investigation of secure sorting algorithms [8]. While the authors of [8] consider only time-based adversaries and noninterference as a security property, CacheAudit allows us to give quantitative answers for a comprehensive set of side-channel adversaries, based on the binary executables and concrete cache models.

As examples, we use the implementations of BubbleSort, InsertionSort, and SelectionSort from [4], which are given in Section 2 and Appendix A, respectively, where we use integer arrays of lengths from 8 to 64.

The results of our analysis are summarized in Figure 9. In the following we highlight some of our findings.

- We obtain the same bounds for BubbleSort and SelectionSort, which is explained by the similar structure of their control flow. A detailed explanation of those bounds is given in Section 2. InsertionSort has a different control flow structure, which is reflected by our data. In particular InsertionSort has only $n!$ possible execution traces due to the possibility of leaving the inner loop, which leads to better bounds w.r.t. trace-based adversaries. However, InsertionSort leaks more information to timing-based adversaries, because the number of iterations in the inner loop varies and thus fewer executions have the same timing.

- For access-based adversaries we obtain zero bounds for all algorithms. For trace-based adversaries, the derived bounds do not imply meaningful security guarantees: the bounds reported for InsertionSort are in the order of $\log_2(n!)$, which corresponds to the maximum information contained in the ordering of the elements; the bounds reported for the other sorting algorithms exceed this maximum, which is caused by the imprecision of the static analysis.

- We performed an analysis of the sorting algorithms for smaller (256B) and larger (64KB) cache sizes and obtained the exact same bounds as in Figure 9, with the exception of the case of arrays of 64 entries and 256B caches: there the leakage increases because the arrays do not fit entirely into the cache due to their misalignment with the memory blocks.

7.4 Discussion and Outlook

A number of comments are in order when interpreting the bounds delivered by CacheAudit. First, we obtained all of the bounds for an *empty* initial cache. As described in Section 3.6, they immediately extend to bounds for *arbitrary* initial cache states, as long as the victim does not access any block that is contained in it. This is relevant,

e.g. for an adversary who can fill the initial cache state only with lines from its own disjoint memory space. For LRU and access-based adversaries, our bounds extend to arbitrary initial cache states without further restriction.

Second, while CacheAudit relies on more accurate models of cache and timing than any information-flow analysis we are aware of, there are several timing-relevant features of hardware it does not capture (and make assertions about) yet, including out-of-order execution, which may reorder memory accesses, TLBs, and multiple levels of caches.

Third, for the case of AES and Salsa20, the derived bounds hold for the leakage about the key in *one* execution, with respect to any payload. For the case of zero leakage (i.e., noninterference), the bounds trivially extend to bounds for multiple executions and imply strong security guarantees. For the case of non-zero leakage, the bounds can add up when repeatedly running the victim process with a fixed key and varying payload, leading to a decrease in security guarantees. One of our prime targets for future work is to derive security guarantees that hold for multiple executions of the victim process. One possibility to achieve this is to employ leakage-resilient cryptosystems [20, 47], where our work can be used to bound the range of the leakage functions.

Finally, note that the bounds delivered by CacheAudit can only be used for certifying that a system is secure; they cannot be used for proving that it is *not*. There are two reasons why the bounds may be overly pessimistic: First, CacheAudit may over-estimate the amount of leaked information due to imprecision of the static analysis. Second, the secret input may not be effectively recoverable from the leaked information by an adversary that is computationally bounded.

8 Related Work

The work most closely related to ours is [34]. There, the authors quantify cache side channels by connecting a commercial, closed-source tool for the static analysis of worst-case execution times [1] to an algorithm for counting concretizations of abstract cache states. The application of the tool to side-channel analysis is limited to access-based adversaries and requires heavy code instrumentation. In contrast, CacheAudit provides tailored abstract domains for all kinds of cache side-channel adversaries, different replacement strategies, and is modular and open for further extensions. Furthermore, the bounds delivered by CacheAudit are significantly tighter than those reported in [34]; see Section 7.

Zhang et al. [48] propose an approach for mitigating timing side channels that is based on contracts between software and hardware. The contract is enforced on the software side using a type system, and on the hardware

array length	8			16			32			64		
	C^{tr}	C^{time}	C^{acc}									
BubbleSort	28	4.86	0	120	6.92	0	496	8.96	0	2016	11	0
InsertionSort	15.23	6.91	0	44.3	10.15	0	117.7	13.3	0	296	15.8	0
SelectionSort	28	4.86	0	120	6.92	0	496	8.96	0	2016	11	0

Figure 9: The table illustrates the security guarantees derived by CacheAudit for the implementations of BubbleSort, SelectionSort, and InsertionSort, for trace-based, timing-based, and access-based adversaries, for LRU caches of 4KB and line sizes of 32B.

side, e.g., by using dedicated hardware such as partitioned caches. The analysis ensures that an adversary cannot obtain any information by observing public parts of the memory; any confidential information the adversary obtains must be via timing, which is controlled using dedicated mitigate commands. Tiwari et al. [45] sketch a novel microarchitecture that facilitates information-flow tracking by design, where they use noninterference as a baseline confidentiality property. Other mitigation techniques include coding guidelines [15] for thwarting cache attacks on x86 CPUs, or novel cache architectures that are resistant to cache side-channel attacks [46]. The goal of our approach is orthogonal to those approaches in that we focus on the *analysis* of microarchitectural side channels rather than on their mitigation. Our approach does not rely on a specific platform; rather it can be applied to any language and hardware architecture, for which abstractions are in place.

Kim et al. put forward StealthMem [29], a system-level defense against cache-timing attacks in virtualized environments. The core of StealthMem is a software-based mechanism that locks pages of a virtual machine into the cache and prevents their eviction by other VMs. StealthMem can be seen as a lightweight variant of flushing/preloading countermeasures. As future work, we plan to use our tool to derive formal, quantitative guarantees for programs using StealthMem.

For the case of AES, there are efficient software implementations that avoid the use of data caches by bit-slicing [28]. Furthermore, a model for statistical estimation of the effectiveness of AES cache attacks based on sizes of cache lines and lookup tables has been presented in [44]. In contrast, our analysis technique applies to arbitrary programs.

Technically, our work builds on methods from quantitative information-flow analysis (QIF) [14], where the automation by reduction to counting problems appears in [9, 38, 26, 37], the connection to abstract interpretation in [35], and the application to side channel analysis in [33]. Finally, our work goes beyond language-based approaches that consider caching [7, 25] in that we rely on more realistic models of caches and aim for more permissive, quantitative guarantees.

9 Conclusions

We presented CacheAudit, the first automatic tool for the static derivation of formal, quantitative security guarantees against cache side-channel attacks. We demonstrate the usefulness of CacheAudit by establishing the first formal proofs of security of software-based countermeasures for a comprehensive set of adversaries and based on executable code.

The open architecture of CacheAudit makes it an ideal platform for future research on microarchitectural side channels. In particular, we are currently investigating the derivation of security guarantees for concurrent adversaries. Progress along those lines will provide a handle for extending our security guarantees to the operating system level. We will further investigate abstractions for hardware features such as pipelines, out-of-order execution, and leakage-resilient cache designs, with the goal of providing broad tool support for reasoning about side-channels arising at the hardware/software interface.

Acknowledgments We thank Adam Chlipala and the anonymous reviewers for the constructive feedback, and Ignacio Echeverría and Guillermo Guridi for helping with the implementation.

This work was partially funded by European Projects FP7-256980 NESSoS and FP7-229599 AMAROUT, by the Spanish Project TIN2012-39391-C04-01 StrongSoft, by the Madrid Regional Project S2009TIC-1465 PROMETIDOS, and by the German Research Council (DFG) as part of the Transregional Collaborative Research Center AVACS.

References

- [1] AbsInt aiT Worst-Case Execution Time Analyzers. <http://www.absint.com/a3/>.
- [2] Intel Advanced Encryption Standard (AES) Instructions Set. <http://software.intel.com/file/24917>.
- [3] PolarSSL. <http://polarssl.org/>.
- [4] Sorting algorithms. <http://www.codebeach.com/2008/09/sorting-algorithms-in-c.html>.

- [5] O. Acı̄cmez and Ç. K. Koç. Trace-driven cache attacks on AES. In *ICICS*, pages 112–121. Springer, 2006.
- [6] O. Acı̄cmez, W. Schindler, and Ç. K. Koç. Cache based remote timing attack on the AES. In *CT-RSA*, pages 271–286. Springer, 2007.
- [7] J. Agat. Transforming out timing leaks. In *POPL 2000*, pages 40–53. ACM, 2000.
- [8] J. Agat and D. Sands. On confidentiality and algorithms. In *SSP*, pages 64–77. IEEE, 2001.
- [9] M. Backes, B. Köpf, and A. Rybalchenko. Automatic discovery and quantification of information leaks. In *SSP*, pages 141–153. IEEE, 2009.
- [10] D. Bernstein. Cache-timing attacks on AES. <http://cr.yp.to/antiforgery/cachetiming-20050414.pdf>.
- [11] D. Bernstein. Salsa20. <http://cr.yp.to/snuffle.html>.
- [12] F. Bourdoncle. Efficient chaotic iteration strategies with widenings. In *FMPA*, pages 128–141. Springer, 1993.
- [13] A. Chlipala. Modular development of certified program verifiers with a proof assistant. In *ICFP*, pages 160–171. ACM, 2006.
- [14] D. Clark, S. Hunt, and P. Malacaria. A static analysis for quantifying information flow in a simple imperative language. *JCS*, 15(3):321–371, 2007.
- [15] B. Coppens, I. Verbauwhede, K. D. Bosschere, and B. D. Suter. Practical mitigations for timing-based side-channel attacks on modern x86 processors. In *SSP*, pages 45–60. IEEE, 2009.
- [16] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction of approximation of fixpoints. In *POPL*, pages 238–252, 1977.
- [17] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *POPL*, pages 269–282, 1979.
- [18] P. Cousot, R. Cousot, and L. Mauborgne. Theories, solvers and static analysis by abstract interpretation. *Journal of the ACM*, 59(6):31, 2012.
- [19] G. Doychev, D. Feld, B. Köpf, L. Mauborgne, and J. Reineke. CacheAudit: A tool for the static analysis of cache side channels. <http://eprint.iacr.org/2013/253>.
- [20] S. Dziembowski and K. Pietrzak. Leakage-resilient cryptography. In *FOCS*, pages 293–302. IEEE, 2008.
- [21] C. Ferdinand, F. Martin, R. Wilhelm, and M. Alt. Cache behavior prediction by abstract interpretation. *Science of Computer Programming*, 35(2):163 – 189, 1999.
- [22] D. Grund. *Static Cache Analysis for Real-Time Systems – LRU, FIFO, PLRU*. PhD thesis, Saarland University, 2012.
- [23] D. Gullasch, E. Bangerter, and S. Krenn. Cache games - bringing access-based cache attacks on AES to practice. In *SSP*, pages 490–505. IEEE, 2011.
- [24] R. Heckmann, M. Langenbach, S. Thesing, and R. Wilhelm. The influence of processor architecture on the design and the results of WCET tools. *IEEE Proceedings on Real-Time Systems*, 91(7):1038–1054, 2003.
- [25] D. Hedin and D. Sands. Timing aware information flow security for a JavaCard-like bytecode. *ENTCS*, 141(1):163–182, 2005.
- [26] J. Heusser and P. Malacaria. Quantifying information leaks in software. In *ACSAC*, pages 261–269. ACM, 2010.
- [27] S. Jana and V. Shmatikov. Memento: Learning secrets from process footprints. In *SSP*, pages 143–157. IEEE, 2012.
- [28] E. Käasper and P. Schwabe. Faster and timing-attack resistant AES-GCM. In *CHES*, pages 1–17, 2009.
- [29] T. Kim, M. Peinado, and G. Mainar-Ruiz. Stealth-Mem: System-level protection against cache-based side channel attacks in the cloud. In *19th USENIX Security Symposium*. USENIX, 2012.
- [30] J. Kinder, F. Zuleger, and H. Veith. An abstract interpretation-based framework for control flow reconstruction from binaries. In *VMCAI*, pages 214–228. Springer, 2009.
- [31] P. Kocher. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In *CRYPTO*, pages 104–113. Springer, 1996.
- [32] P. Kocher, J. Jaffe, and B. Jun. Differential power analysis. In *CRYPTO*, pages 388–397. Springer, 1999.
- [33] B. Köpf and D. Basin. An Information-Theoretic Model for Adaptive Side-Channel Attacks. In *CCS*, pages 286–296. ACM, 2007.
- [34] B. Köpf, L. Mauborgne, and M. Ochoa. Automatic quantification of cache side-channels. In *CAV*, pages 564–580. Springer, 2012.
- [35] B. Köpf and A. Rybalchenko. Approximation and randomization for quantitative information-flow analysis. In *CSF*, pages 3–14. IEEE, 2010.
- [36] L. Mauborgne and X. Rival. Trace partitioning in abstract interpretation based static analyz-

- ers. In *ESOP*, volume 3444 of *LNCS*, pages 5–20. Springer, 2005.
- [37] Z. Meng and G. Smith. Calculating bounds on information leakage using two-bit patterns. In *PLAS*. ACM, 2011.
- [38] J. Newsome, S. McCamant, and D. Song. Measuring channel capacity to distinguish undue influence. In *PLAS*, pages 73–85. ACM, 2009.
- [39] D. A. Osvik, A. Shamir, and E. Tromer. Cache attacks and countermeasures: the case of AES. In *CT-RSA*, volume 3860 of *LNCS*, pages 1–20. Springer, 2006.
- [40] C. Percival. Cache missing for fun and profit. In *BSDCan*, 2005.
- [41] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage. Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds. In *CCS*, pages 199–212. ACM, 2009.
- [42] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, 2003.
- [43] G. Smith. On the foundations of quantitative information flow. In *FoSSaCS*, pages 288–302. Springer, 2009.
- [44] K. Tiri, O. Acıicmez, M. Neve, and F. Andersen. An analytical model for time-driven cache attacks. In *FSE*, volume 4593 of *LNCS*, pages 399–413. Springer, 2007.
- [45] M. Tiwari, J. Oberg, X. Li, J. Valamehr, T. E. Levin, B. Hardekopf, R. Kastner, F. T. Chong, and T. Sherwood. Crafting a usable microkernel, processor, and I/O system with strict and provable information flow security. In *ISCA*, pages 189–200. ACM, 2011.
- [46] Z. Wang and R. B. Lee. New cache designs for thwarting software cache-based side channel attacks. In *ISCA*, pages 494–505. ACM, 2007.
- [47] Y. Yu, F.-X. Standaert, O. Pereira, and M. Yung. Practical leakage-resilient pseudorandom generators. In *CCS*, pages 141–151. ACM, 2010.
- [48] D. Zhang, A. Askarov, and A. C. Myers. Language-based control and mitigation of timing channels. In *PLDI*, pages 99–110. ACM, 2012.
- [49] Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart. Cross-VM side channels and their use to extract private keys. In *CCS*. ACM, 2012.

A Example Code

Selection Sort

```
void SelectionSort(int a[], int array_size){
    int i;
    for (i = 0; i < array_size - 1; ++i){
        int j, min, temp;
        min = i;
        for (j = i+1; j < array_size; ++j){
            if (a[j] < a[min])
                min = j;
        }
        temp = a[i];
        a[i] = a[min];
        a[min] = temp;
    }
}
```

Insertion Sort

```
void InsertionSort(int a[], int array_size){
    int i, j, index;
    for (i = 1; i < array_size; ++i){
        index = a[i];
        for (j = i; j > 0 && a[j-1] > index; j--)
            a[j] = a[j-1];
        a[j] = index;
    }
}
```

Transparent ROP Exploit Mitigation using Indirect Branch Tracing

Vasilis Pappas, Michalis Polychronakis, Angelos D. Keromytis
Columbia University

Abstract

Return-oriented programming (ROP) has become the primary exploitation technique for system compromise in the presence of non-executable page protections. ROP exploits are facilitated mainly by the lack of complete address space randomization coverage or the presence of memory disclosure vulnerabilities, necessitating additional ROP-specific mitigations.

In this paper we present a practical runtime ROP exploit prevention technique for the protection of third-party applications. Our approach is based on the detection of abnormal control transfers that take place during ROP code execution. This is achieved using hardware features of commodity processors, which incur negligible runtime overhead and allow for completely transparent operation without requiring any modifications to the protected applications. Our implementation for Windows 7, named *kBouncer*, can be selectively enabled for installed programs in the same fashion as user-friendly mitigation toolkits like Microsoft’s EMET. The results of our evaluation demonstrate that kBouncer has low runtime overhead of up to 4%, when stressed with specially crafted workloads that continuously trigger its core detection component, while it has negligible overhead for actual user applications. In our experiments with in-the-wild ROP exploits, kBouncer successfully protected all tested applications, including Internet Explorer, Adobe Flash Player, and Adobe Reader.

1 Introduction

Despite considerable advances in system protection and exploit mitigation technologies, the exploitation of software vulnerabilities persists as one of the most common methods for system compromise and malware infection. Recent prominent examples include in-the-wild exploits against Internet Explorer [7], Adobe Flash Player [2], and Adobe Reader [19, 1], all capable of successfully

bypassing the data execution prevention (DEP) and address space layout randomization (ASLR) protections of Windows [49], even on the most recent and fully updated (at the time of public notice) systems.

Data execution prevention and similar non-executable page protections [55], which prevent the execution of injected binary code (shellcode), can be circumvented by reusing code that already exists in the vulnerable process to achieve the same purpose. Return-oriented programming (ROP) [62], the latest advancement in the more than a decade-long evolution of code reuse attacks [30, 51, 50, 43], has become the primary exploitation technique for achieving arbitrary code execution in the presence of non-executable page protections.

Although DEP is complemented by ASLR, which is meant to prevent code reuse attacks by randomizing the load addresses of executables and DLLs, its deployment is problematic. A few code segments left in static locations can be enough for mounting a robust ROP attack, and unfortunately this is quite often the case [35, 75, 40, 54]. More importantly, even if a process is fully randomized, it might be possible to calculate the base address of a DLL at runtime [19, 61, 44, 69, 37, 66], or infer it in a brute-force way [63].

This situation has prompted active research on additional defenses against return-oriented programming. Recent proposals can be broadly classified in static software hardening and runtime monitoring solutions. Schemes of the former type include compiler extensions for the protection of indirect control transfers [45, 52], which break the chaining of the “gadgets” that comprise a return-oriented program, and code diversification techniques based on static binary rewriting [70, 53], which randomize the locations or the outcome of the available gadgets. The lack of source code for proprietary software hinders the deployment of compiler-based approaches. Depending on the applied code transformations, static binary rewriting approaches may be applied on stripped binaries, but their outcome depends on the accuracy

of code disassembly and control flow graph extraction, while the rewriting phase is time-consuming. Depending on the vulnerable program, fine-grained code randomization may be circumvented by dynamically building the ROP payload at the time of exploitation [66, 16]. Runtime solutions monitor execution at the instruction level to apply various protection approaches, such as performing anomaly detection by checking for an unusually high frequency of `ret` instructions [24, 28], ensuring the integrity of the stack [29], or randomizing the locations of code fragments [36]. The use of dynamic binary instrumentation allows these systems to be transparent to the protected applications, but is also their main drawback, as it incurs a prohibitively high runtime overhead.

Transparency is a key factor for enabling the practical applicability of techniques that aim to protect proprietary software. The absence of any need for modifications to existing binaries ensures an easy deployment process, and can even enable the protection of applications that are already installed on end-user systems [47]. At the same time, to be practical, mitigation techniques should introduce minimal overhead, and should not affect the proper execution of the protected applications due to incompatibility issues or false positives.

Aiming to fulfill the above requirements, in this paper we present a fully transparent runtime ROP exploit mitigation technique for the protection of third-party applications. Our approach is based on monitoring the executed indirect branches at critical points during the lifetime of a process, and identifying abnormal control flow transfers that are inherently exhibited during the execution of ROP code. The technique is built around Last Branch Recording (LBR), a recent feature of Intel processors. Relying mainly on hardware for instruction-level monitoring allows for minimal runtime overhead and completely transparent operation, without requiring any modifications to the protected applications.

Inspired by application hardening toolkits like Microsoft’s EMET [47], our prototype implementation for Windows 7, named *kBouncer*, can be selectively enabled for the protection of already installed applications. Besides typical ROP code, kBouncer can also identify the execution of “jump-oriented” code that uses gadgets ending with indirect `jmp` or `call` instructions. To minimize context switching overhead, branch analysis is performed only before critical system operations that could cause any harm. To verify that kBouncer introduces minimal overhead, we stress-tested our implementation with workloads that trigger excessively the protected system functions. In the worst case, the average measured overhead was 1%, and it never exceeded 4%. As the protected operations occur several orders of magnitude less frequently in regular applications, the performance impact of kBouncer in practice is negligible. We evaluated the

effectiveness and practical applicability of our technique using publicly available ROP exploits against widely used software, including Internet Explorer, Adobe Flash Player, and Adobe Reader. In all cases, kBouncer blocks the exploit successfully, and notifies the user through a standard error message window.

The main contributions of our work are:

- We present a *practical* and *transparent* ROP exploit mitigation technique based on runtime monitoring of indirect branch instructions using the LBR feature of recent CPUs.
- We have implemented the proposed approach as a self-contained toolkit for Windows 7, and describe in detail its design and implementation.
- We provide a quantitative analysis of the robustness of the proposed ROP code execution prevention technique against potential evasion attempts.
- We have experimentally evaluated the performance and effectiveness of kBouncer, and demonstrate that it can prevent in-the-wild exploits against popular applications with negligible runtime overhead.

2 Practical Indirect Branch Tracing for ROP Prevention

The proposed approach uses runtime process monitoring to block the execution of code that exhibits return-oriented behavior. In contrast to typical program code, the code used in ROP exploits consists of several small instruction sequences, called *gadgets*, scattered through the executable segments of the vulnerable process. Gadgets end with an indirect branch instruction that transfers control to the following gadget according to a sequence of gadget addresses contained in the “payload” that is injected during the attack. As the name of the technique implies, gadgets typically end with a `ret` instruction, although any combination of indirect control transfer instructions can be used [23].

The key observation behind our approach is that the execution behavior of ROP code has some inherent attributes that differentiate it from the execution of legitimate code. By monitoring the execution of a process while focusing on those properties, kBouncer can identify and block a ROP exploit before its code accomplishes any critical operation.

In this section, we discuss in detail how kBouncer leverages the Last Branch Recording feature of recent processors to retrieve the sequence of the most recent indirect branch instructions that took place right before the invocation of a system function. In the following section, we discuss how kBouncer uses this information to identify the execution of ROP code. As the vast majority of in-the-wild ROP exploits target Windows software,

Technique	Overhead	Requir.	Compat.	Deployment
Compiler-level	med	source	some	hard
Binary rewrit.	med	pdb	no	med
Dynamic Instr.	high	-	yes	med
LBR monitoring	low	-	yes	easy

Table 1: Qualitative comparison of alternative techniques for runtime branch monitoring.

our design focuses on achieving transparent operation for existing Windows applications without raising any compatibility issues or false alerts.

2.1 Branch Tracing vs. Other Approaches

Execution monitoring at the instruction level usually comes with an increased runtime overhead. Even when tracking only a particular subset of instructions, e.g., in our case only indirect control transfer instructions, the overhead of interrupting the normal flow of control and updating the necessary accounting information is prohibitive for production systems. There are several different approaches that can be followed for monitoring the execution of indirect branch instructions, each of them having different requirements, performance overhead, transparency level, and deployment effort.

Extending the compiler to generate and embed runtime checks in the executable binary at compile time is one of the simplest techniques [52]. However, the high frequency of control transfer instructions in typical code means that a lot of additional instrumentation code must be added. Also, deployment requires a huge effort as all programs have to be recompiled. Another option is static binary rewriting. Its main advantage over compiler-level techniques is that no source code is required, but only debug symbols (e.g., PDB files) [17]. Still, all control transfers need to be checked. Even worse, it breaks self-checksumming or signed code and cannot be applied to self-modifying programs. Dynamic binary instrumentation is another alternative that can handle even stripped binaries (no need for source code or debug symbols), but the runtime performance overhead of existing binary instrumentation frameworks slows down the normal execution of an application by a factor of a few times [29].

In contrast to the above approaches, our system monitors the executed indirect branch instructions using Last Branch Recording (LBR) [39, Sec. 17.4], a recent feature of Intel processors introduced in the Nehalem architecture. When LBR is enabled, the CPU tracks the last N (16 for the CPU model we used) most recent branches in a set of 64-bit model-specific registers (MSR). Each branch record consists of two MSR registers, which hold the linear addresses of the branch instruction and its target instruction, respectively. Records from the LBR

stack can be retrieved using a special instruction (`rdmsr`) from privileged mode. The processor can be configured to track only a subset of branches based on their type: relative/indirect calls/jumps, returns, and so on.

Table 1 shows a summarized comparison of the alternative strategies discussed above. For our particular case, the use of LBR has several advantages: it incurs zero overhead for storing the branches; it is fully transparent to the running applications; it does not cause any incompatibility issues as it is completely decoupled from the actual execution; it does not require source code or debug symbols; and it can be dynamically enabled for already installed applications—there is no need for recompilation or instruction-level instrumentation.

2.2 Using Last Branch Recording for ROP Prevention

Although the CPU continuously records the most recent branches in the LBR stack with zero overhead, accessing the LBR registers and retrieving the recorded information unavoidably adds some overhead. Considering the limited size (16 entries) of the LBR stack, and that it can be accessed only from kernel-level code, checking the targets of all indirect control transfer instructions would incur a prohibitively high performance overhead. Indirect branches occur very frequently in typical programs, and a monitored process should be interrupted once every 16 branches with a context switch. In fact, the implementation of such a scheme is not facilitated by the current design of the LBR feature, as it does not provide any means of interrupting execution whenever the stack gets full after retrieving its previous 16 records.

Fortunately, when considering the actual operations of a ROP exploit, it is possible to dramatically reduce the number of control transfer instructions that need to be inspected. The typical end goal of malicious code is to give the attacker full control of the victim system. This usually involves just a few simple operations, such as dropping and executing a malicious executable on the victim system, which unavoidably require interaction with the OS through the system call interface. Based on this observation, we can refine the set of indirect branches that need to be inspected to only those along the final part of the execution path that lead to a system call invocation. (Depending on the vulnerable program, exploitation might be possible without invoking any system call, e.g., by modifying a user authentication variable [25], but such attacks are rarely found in the client-side applications that are typically targeted by current ROP exploits, and are outside the scope of this work.)

Figure 1 illustrates this approach. Vertical bars correspond to snapshots of the address space of a process, and arrows correspond to indirect control transfers. The ver-

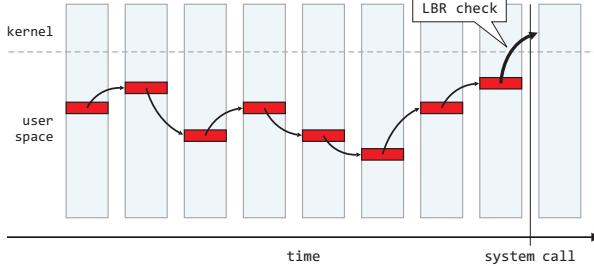


Figure 1: Illustration of a basic scheme for ROP code detection. Whenever control is transferred from user to kernel space (vertical line), the system inspects the most recent indirect branches to decide whether the system call was invoked by ROP code or by the actual program.

tical line denotes the point at which the flow of control is transferred from user space to kernel space through a system call. At this point, by interposing at the OS’s system call handler, the system can access the LBR stack and retrieve the targets of the indirect branches that led to the system call. It can then check the control flow path for abnormal control transfers and distinctive properties of ROP-like behavior using the techniques that will be described in Sec. 3, and decide whether the system call is part of malicious ROP code, or it is being invoked legitimately by the actual program.

2.2.1 System Calls vs. API Calls

User-level programs interact with the underlying system mainly through system calls. Unix-like systems provide to applications wrapper functions for the available system calls (often using the same name as the system call they invoke) as part of the standard library. In contrast, Windows does not expose the system call interface directly to user-level programs. Instead, programs interact with the OS through the Windows API [13], which is organized into several DLLs according to different kinds of functionality. In turn, those DLLs call functions from the undocumented Native API [59], implemented in `ntdll.dll`, to invoke kernel-level services.

Exploit code rarely relies on the Native API for several reasons. One problem is that system call numbers change between Windows versions and service pack levels [18, 14], reducing the reliability of the exploit across different targets (or increasing attack complexity by having to adjust the exploit according to the victim’s OS version). Most importantly, the desired functionality is often not conveniently exposed at all through the Native API, as for example is the case with the socket API [65]. Typically, the purpose of ROP code is to give execute permission to a second-stage shellcode using `VirtualProtect` or a similar API function [31, 27, 1, 6, 7, 2]. The

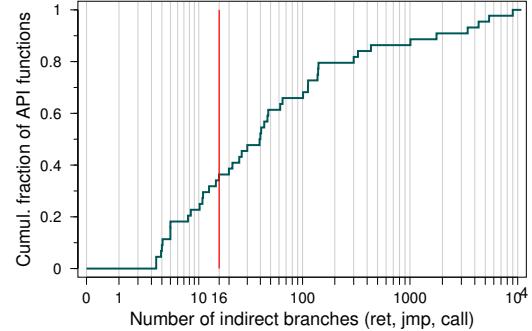


Figure 2: LBR overwriting due to indirect branches that take place within Windows API functions, prior to the execution of a system call.

second-stage shellcode can be avoided altogether by implementing all the necessary functionality solely using ROP code, as is the case with a recent exploit against Adobe Reader XI, in which the ROP code calls directly the `fsopen`, `write`, `fclose`, and `LoadLibraryW` functions to drop and execute a malicious DLL [19].

The implementation of many of the functions exported by the Windows API is quite complex, and often involves several internal functions that are executed before the invocation of the intended system call. Due to the limited size of the LBR stack, this means that by the time execution reaches the actual system call, the LBR stack might be filled with indirect branches that took place *after* the Windows API function was called. To assess the extent of this effect, we measured the average number of indirect branch instructions (`ret`, `jmp`, and `call`) that are executed between the first instruction of a Windows API function and the system call it invokes, for a set of 52 “sensitive” functions that are commonly used in Windows shellcode and ROP code implementations (a complete list of the tested functions is provided in the appendix). As shown in Fig. 2, about 34% of the API functions execute less than 16 indirect branches, while the rest of them completely overwrite the LBR stack.

As these branches are made as part of legitimate execution paths, calling a function that completely overwrites the LBR stack would allow ROP code to evade detection. However, this scheme can be improved to provide robust detection of ROP code that calls *any* sensitive API function, irrespectively of the extent of overwriting in the LBR stack due to code in the function body.

2.2.2 LBR Stack Inspection on API Function Entry

Given that i) exploit code usually calls Windows API functions instead of directly invoking system calls, and ii) most API functions overwrite the LBR stack with legitimate indirect branches before invoking a system call,

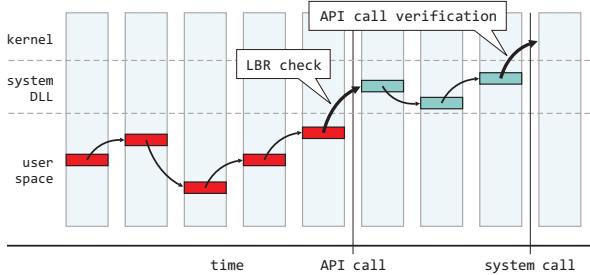


Figure 3: Overview of the detection scheme of kBouncer. Before the invocation of protected Windows API functions, the system inspects the LBR stack to identify whether the execution path that led to the call was part of ROP code, and writes a checkpoint. To account for ROP code that would bypass the check by jumping over kBouncer’s function hook, the system then verifies the entry point of the API function at the time of the corresponding system call invocation.

kBouncer inspects the LBR stack at the time an API function is called, instead upon system call invocation. This allows the detection of ROP code that uses any sensitive API function, irrespectively of the number of legitimate indirect branches executed within its body. In case an API function is called by ROP code, all entries in the LBR stack at the time of function entry will correspond to the indirect branches of the gadgets that lead to the function call, as depicted in Fig. 3.

Still, without any additional precautions, this scheme would allow an attacker to bypass the LBR check at the entry point of a function. An implementation of the LBR check in the system call handler—within the kernel—safeguards it from user-level code and any bypass attempt. In contrast, implementing the LBR check as a hook to a user-level function’s entry point does not provide the same level of protection. An attacker could avoid the check by jumping over the hook at the function’s prologue, instead of jumping at its main entry point, and then normally executing the function body. Alternatively, by trading off some of its reliability, the ROP code could avoid calling the API function altogether by invoking directly the relevant Native API call.

Fortunately, as the Native API is not exposed to user-level programs, i.e., applications never invoke Native API calls directly; we can solve this issue by ensuring that system calls are *always* invoked solely through their respective Windows API functions. After a clear LBR check at an API function’s entry point, kBouncer writes a checkpoint that denotes a legitimate invocation of that particular function. When the respective system call is later invoked, the system call handler verifies that a proper checkpoint was previously set by the expected API function, and clears it. If the checkpoint was not

set, then this means that the flow of control did not pass through the proper API function preamble, and kBouncer reports a violation.

We should note that user-level ROP code cannot bypass kBouncer’s checks by faking a checkpoint. The code for setting a checkpoint can only run with kernel privileges, and the checkpoint itself is stored in kernel space so that i) the system call handler can later access it, and ii) any user-level code (and consequently the ROP code itself) cannot tamper with it. The checkpoint code is tied with and comes right after the code that inspects the LBR stack, and both run in an atomic way at kernel level, i.e., the checkpoint cannot be set without previously analyzing the LBR for the presence of ROP code. This prevents any ROP code from faking a checkpoint without being detected—the part of the ROP code with the task of setting the checkpoint would be detected by the LBR check before the checkpoint is actually set.

3 Identifying the Execution Behavior of ROP Code

Before allowing a Windows API function call to proceed, kBouncer analyzes the most recent indirect branches that were recorded in the LBR cache prior to the function call. LBR is configured to record only `ret`, indirect `jmp`, and indirect `call` instructions. The execution of ROP code is identified by looking for two prominent attributes of its runtime behavior: i) illegal `ret` instructions that target locations not preceded by call sites, and ii) sequences of relatively short code fragments “chained” through any kind of indirect branches.

Returns that do not transfer control right after call sites is an illegitimate behavior exhibited by all publicly available ROP exploits against Windows software, which rely mainly on gadgets ending with `ret` instructions (`ret` conveniently manipulates both the program counter and the stack pointer). The second, more generic attribute captures an inherent property of not only purely return-oriented code, but also of advanced (and admittedly harder to construct) jump-oriented code (or even “hybrid” ROP/JOP code that might use any combination of gadgets ending with `jmp`, `call`, and `ret` instructions).

3.1 Illegal Returns

When focusing on the control flow behavior of ROP code at the instruction level, we expect to observe the successive execution of several `ret` instructions, which correspond to the transfer of control from each gadget to the next one. Although this control flow pattern is quite distinctive, the same pattern can also be observed in legitimate code, e.g., when a series of functions consecutively

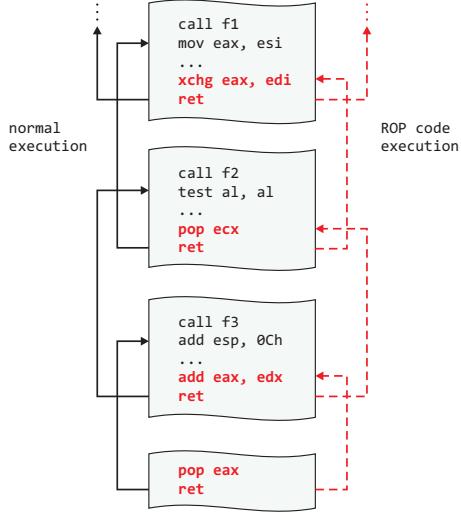


Figure 4: In normal code, `ret` instructions target valid call sites (left), while in ROP code, they target gadgets found in arbitrary locations (right).

return to their callers. However, when considering the *targets* of `ret` instructions, there is a crucial difference.

In a typical program, `ret` instructions are paired with `call` instructions, and thus the target of a legitimate `ret` corresponds to the location right after the call site of the respective caller function, i.e., an instruction that follows a `call` instruction, as illustrated in the left part of Fig. 4. In contrast, a `ret` instruction at the end of a gadget transfers control to the first instruction of the following gadget, which is unlikely to be preceded by a `call` instruction. This is because gadgets are found in arbitrary locations across the code image of a process, and often may correspond to non-intended instruction sequences that happen to exist due to overlapping instructions [62].

At runtime, the `ret` instructions of ROP code can be easily distinguished from the legitimate return instructions of a benign program by checking their targets. A `ret` instruction that transfers control to an instruction not preceded by a `call` is considered illegal, and the observation of an illegal `ret` is flagged by kBouncer as an indication of ROP code execution.

Ensuring `call-ret` pairing by verifying caller-callee semantics, e.g., using a shadow stack [29], constrains the control flow of a process in a much stricter way than the proposed scheme. In practice, though, enforcing such a strict policy is problematic, due to the use of `setjmp/longjmp` constructs, `call/pop` “getPC” code commonly found in position-independent executables, tail call optimizations, and lightweight user-level threads such as Windows fibers, in which the context switch function called by the current thread returns to the thread that is scheduled next.

Instead of enforcing a strict control flow, kBouncer simply makes sure that `ret` instructions always target *any* among all valid call sites (even those that correspond to non-intended `call` instructions). This is a more relaxed constraint that is not expected to be violated (and which did not, for the set of applications tested as part of our experimental evaluation) even in programs that use constructs like the above. Its implementation is also much simpler, as there is no need to track the execution of `call` instructions—checking that the target of each `ret` falls right after a `call` is enough.

Call-preceded Gadgets Although the above scheme prohibits the execution of illegal returns, which are prominently exhibited by typical ROP exploits, an attacker might still be able to construct functional ROP code using gadgets that begin right after a `call` instruction, to which we refer as *call-preceded gadgets*. Note that *call-preceded* gadgets may begin after either intended or unintended `call` instructions. As kBouncer cannot know which `call` instructions were actually emitted by the compiler, if any of the possible valid instructions immediately preceding the instruction at a target address is a `call` instruction, then that address may correspond to the beginning of a *call-preceded* gadget.

The observation of a `ret` that targets an instruction located right after a `call` is considered by kBouncer as normal, and thus ROP code comprising only *call-preceded* gadgets would not be identified based on the first ROP code attribute kBouncer looks for during branch analysis. Although such code would still be identified due to its “chained gadgets” behavior, which we will discuss below, we first briefly explore the feasibility of such an attempt.

For our analysis we use a set of typical Windows applications, detailed in Table 2. The data is collected using a purpose-built execution analysis framework, described in Sec. 4.2. We consider as a gadget any (intended or unintended) instruction sequence that ends with an indirect branch, and which does not contain any privileged or invalid instruction. In contrast to the gadget constraints typically considered in relevant studies [62, 23, 60, 24, 73, 53, 36, 70] and the actual gadgets used in real exploits [27, 19, 1, 6, 7, 2], i.e., contiguous instruction sequences no longer than five instructions, we follow a more conservative approach and consider gadgets that i) may be *split* into several fragments due to internal conditional or unconditional relative jumps, and ii) have a maximum length of 20 *instructions*.

Figure 5 shows the fraction of *call-preceded* gadgets among all gadgets that end with a `ret` instruction, for different Windows applications. In the worst case, only 6.4% of the gadgets begin right after call sites, a percentage much smaller compared to all available `ret` gad-

Application	Workload	Indirect jump	Branches call	ret	System Calls	Protected API Calls
Windows Media Player	Music playback for ~30 secs	7.3M	7.5M	30.0M	196K	5150
Internet Explorer 9	Browse to google.com	3.4M	4.8M	17.7M	87K	7092
Adobe Flash Player	Watch a youtube video	9.6M	21.7M	94.1M	317K	46658
Microsoft Word	Scroll through a document	3.3M	11.5M	38.8M	178K	5425
Microsoft Excel	Open a rich spreadsheet	6.3M	18.1M	54.5M	212K	3957
Microsoft PowerPoint	View a presentation	10.2M	19.3M	68.8M	275K	6577
Adobe Reader XI	Scroll through a few pages	9.7M	19.5M	100.6M	101K	5026

Table 2: Details about the dataset used for gadget analysis.

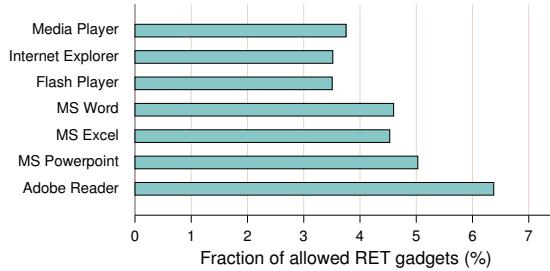


Figure 5: Among all gadgets that end with a `ret` instruction, only a small fraction (6.4% in the worst case for Adobe Reader) begin right after call sites.

gets. Given that many of them are longer than the typical gadget size, and are thus harder to use in ROP code due to the many different operations and register or memory state changes they incur, an attacker would be left with a severely limited set of gadgets to work with. For comparison, the ROP payloads of the exploits we used in our evaluation, listed in Table 4, collectively use 44 unique gadgets with an average length of just 2.25 instructions, and only *three* of them happen to be `call`-preceded—the rest of them would all result in illegal returns.

3.2 Gadget Chaining

It is clear from the previous section that even a “lighter” version of kBouncer that would just prohibit the execution of illegal returns would still significantly raise the bar, as i) it would prevent the execution of the ROP code typically found in publicly available Windows exploits, and more importantly, ii) it would force attackers to either use only a limited set of `ret` gadgets, or resort to jump-oriented code—options of increased complexity.

To account for potential future exploits of these sorts, the second attribute that kBouncer uses to identify the execution of ROP code is an inherent characteristic of its construction: the observation of several short instruction sequences *chained* through indirect branches. This is a generic constraint that holds for both return-oriented and jump-oriented code (or potential combinations—in the rest of this section we refer to both techniques as ROP).

LBR Stack		
Branch	Target	
G05 7C3411C0	pop ecx	
G05 7C3411C1	ret	
G11 7C3415A2	jmp [eax]	
G02 7C34252C	pop ebp	G01
G02 7C34252D	ret	
G04 7C345249	pop edx	G02
G04 7C34524A	ret	
G10 7C346C0B	ret	G03
G01 7C34A028	pop edi	
G01 7C34A029	pop esi	G04
G01 7C34A02A	ret	
G06 7C34B8D7	pop edi	G05
G06 7C34B8D8	ret	
G07 7C366FA6	pop esi	G06
G07 7C366FA7	ret	
G03 7C36C55A	pop ebx	G07
G03 7C36C55B	ret	
G08 7C3762FB	pop eax	G08
G08 7C3762FC	ret	
G09 7C378C81	pusha	G09
G09 7C378C82	add al, 0EFh	
G09 7C378C84	ret	G10
		G11

Figure 6: The state of the LBR stack at the time kBouncer blocks an exploit against Adobe Flash [2]. Diagonal pairs of addresses with the same shade correspond to the first and last instruction of each gadget.

Although legitimate programs also contain an abundance of code fragments linked with indirect branches, these fragments are typically much larger than gadgets, and more importantly, they do not tend to form long uninterrupted sequences (as we show below).

The CPU records in-sequence all executed indirect branches, enabling kBouncer to reconstruct the chain of gadgets used by any ROP code. Each LBR record $R[b,t]$ contains the address of the branch (b) and the address of its target (t), or from the viewpoint of ROP code, the *end* of a gadget and the *beginning* of the following one.

Figure 6 illustrates the contents of the LBR stack at the time kBouncer blocks the ROP code of an exploit against Adobe Flash [2] (although kBouncer blocks this exploit due to illegal returns, we use it for illustrative purposes, as we are not aware of any publicly available JOP exploit). Starting with the most recent (bottom-most) record, the detection algorithm checks whether the tar-

get (located at address $R_{n-1}[t]$) of the previous branch, is an instruction that precedes the branch (located at address $R_n[b]$) of the current record. If starting from address $R_{n-1}[t]$, there exists an uninterrupted sequence of at most 20 instructions that ends with the indirect branch at address $R_n[b]$, then the sequence is considered as a gadget. Recall that kBouncer treats as gadgets even fragmented instruction sequences linked through conditional or unconditional relative jumps. The same process repeats with the previous records, moving upwards, as long as chained gadgets are found.

The ROP code in this example consists of 11 gadgets, all ending with a `ret` instruction except the final one (G11), which is a single-instruction gadget with an indirect `jmp` that transfers control to `VirtualProtect` in `kernel32.dll` (note the difference in the high bytes of the target address in record 13). The two bottom-most records in the LBR stack correspond to kBouncer’s function hook (from `VirtualProtect` to `DeviceIoControl`, which signals the kernel component), and a `ret` from `_SEH_prolog4` which is called by `DeviceIoControl`.

A crucial question for the effectiveness of the above algorithm is whether legitimate code could be misclassified as ROP code due to excessively long chains of gadget-like instruction sequences. To assess this possibility, we measured the length of the gadget chains observed across all inspected LBR stack instances for the applications and workloads listed in Table 2. As described in Sec. 2.2.2, kBouncer inspects the LBR stack right before the execution of a sensitive Windows API function. In total, kBouncer inspected 79,885 LBR stack instances, i.e., the tested applications legitimately invoked a sensitive API function 79,885 times.

Figure 7 (solid line) shows the percentage of instances with a given gadget chain length. In the worst case, there is just one instance with a chain of five gadgets, and there are no instances with six or more gadgets. On the other hand, complex ROP code that would rely on `call`-preceded or non-`ret` gadgets would result in excessively long gadget chains, filling the LBR stack. Indicatively, a jump-oriented Turing-complete JOP implementation for Linux uses 34 gadgets [23]. Furthermore, current JOP code implementations rely on a special dispatcher gadget that always executes between useful gadgets, at least doubling the amount of executed gadgets.

Although we can never rule out the possibility that benign code in some other application might result in a false positive, to ascertain that this possibility is unlikely, we also analyzed 97,554,189 LBR stack instances taken at the entry points of all executed functions during the lifetime of the same tested applications. In this orders-of-magnitude larger data set, the maximum gadget chain length observed is nine (dashed line), which is still far

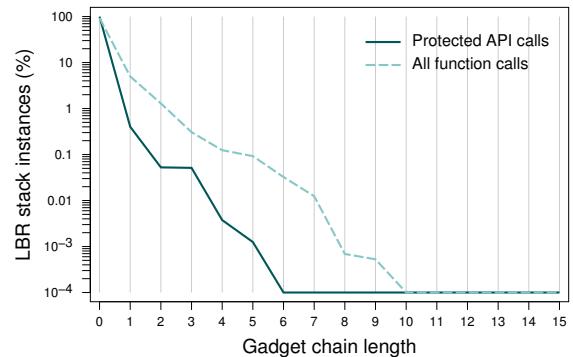


Figure 7: Percentage of LBR stack instances with a given gadget chain length for i) the instances inspected by kBouncer at the entry points of protected API function calls, and ii) the instances taken at the entry points of all function calls.

from filling up the LBR stack. This means that even if there is a need in the future to protect more API functions, or perform LBR checks in other parts of a program, we will more than likely still be able to set a robust detection threshold that will not result in false positives. For the current set of protected functions we use a threshold of eight gadgets, which allows for increased resilience to false positives.

Finally, note that in the above benign executions, the vast majority of the gadget-like chains stem from our conservative choice of considering *fragmented* gadgets of up to 20 *instructions* long—significantly more complex and longer than the gadgets used in actual exploits. Although we could choose more reasonable constraints about what is considered as a gadget, we preferred to stress the limits of the proposed approach.

4 Implementation

4.1 kBouncer

To demonstrate the effectiveness of our proposed approach, we developed a prototype implementation for the x86 64-bit version of Windows 7 Professional SP1. Our prototype, kBouncer, consists of three components: i) an offline gadget extraction and analysis toolkit, ii) a user-space thin interposition layer between the applications and Windows API functions, and iii) a kernel module.

For the executable segments of a protected application, the gadget extraction toolkit identifies any instruction sequence ending in an indirect branch, starting from each and every byte of a segment. In the current version of our prototype we assume that the complete set of an application’s modules is available in advance. However, it is possible to trivially relax this assumption by process-

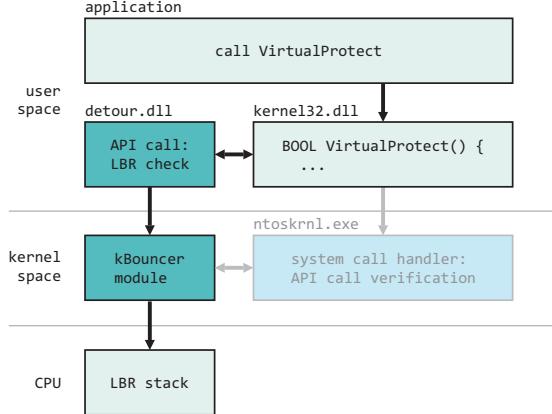


Figure 8: Overview of kBouncer’s implementation. At the entry point of Windows API functions, kBouncer detours the execution, inspects the LBR stack in kernel mode, and then returns control back to the application.

ing new modules on-the-fly at the time they are loaded by a protected application. The maximum gadget length is given as a parameter—in our experiments we conservatively used a length of 20 instructions. As discussed in Sec. 3.1, our extraction algorithm differs from previous approaches as it considers even instruction sequences that contain conditional or unconditional relative jumps. For this reason, code analysis explores all possible paths from every offset within a code segment, and follows recursively any conditional branches. The output of the analysis phase is two hash tables: one containing the offsets of `call`-preceded gadgets, and another containing the rest of the found gadgets. In the future, we will consider switching to Bloom filters to save space.

The overall operation of the runtime system is depicted in Fig. 8. The interposition component is implemented on top of the Detours framework [38], which provides a library call interception mechanism for the Windows platform. During initialization, it requests by kBouncer’s kernel module to enable the LBR feature on the CPU. The two components communicate through control messages over a pseudo-device that is exported by the kernel module (using the `DeviceIoControl` API function). Then, it selectively hooks the set of the protected Windows API functions. Each time a protected function is called, the detour code sends a control message to the kernel component, instructing it to inspect the contents of the LBR stack for abnormal control transfers.

The kernel module is responsible for three main tasks: i) enabling or disabling the LBR facility, ii) analyzing the recorded indirect branches, and iii) writing and verifying the appropriate checkpoint before allowing a system call to proceed. The first task involves reading and writing a few Model Specific Registers (MSR) using the `rdmsr`

and `wrmsr` instructions. For the second task, whenever a control request is received from the user-space component, kBouncer analyzes the contents of the LBR stack, looking for the attributes described in Sec. 3. The MSR registers that hold the recorded information and configuration parameters are considered part of the running process context, and are preserved during context switches.

To identify illegal return instructions, the kernel module fetches a few bytes before each return target and attempts to decode any `call` instruction located right before the target instruction (call site check). Gadget chaining patterns are identified as follows: starting from the most recent branch in the LBR stack, the number of consecutive targets that point to gadgets are counted. Any `ret` targets are looked up in the `call`-preceded gadgets hash table, whereas `call` or `jmp` targets are looked up in both hash tables, `call`-preceded or not. The most recent branch target is not considered, as it does not point to a gadget, but to the protected API function. To protect the kernel-level component from potential crashes when accessing invalid user-level locations, we use the `ProbeForRead` function of the Windows kernel API.

Unfortunately, the final task for API call verification has been only partly implemented, as it is not possible to perform system-call interposition in the current version of Windows 7. A recently added kernel feature in the 64-bit version of Windows, called PatchGuard [32], protects against kernel-level rootkits by preventing any changes to critical data structures, such as the System Service Descriptor Table (SSDT). Although this is effective against rootkits, PatchGuard removed the ability of legitimate applications, such as antivirus software, to intercept system calls. In response, Microsoft added a set of kernel-level APIs for filtering network and file-system operations (Windows Filtering Platform [48]). Hopefully, future OS versions will provide system call filtering capabilities as well.

Still, we did verify the correct operation of checkpoint verification by simulating it using the dataset of Table 2. We should note that this is not a design limitation, but only an implementation issue stemming from our choice of the target platform. For example, this would not have been an issue had we decided to implement kBouncer for Linux, or any other open platform. For now, we plan to implement the checkpointing functionality for 32-bit applications by hooking system calls at user level through the WOW64 layer [4] (which, however, will not provide the same protection guarantees as an actual kernel-level implementation).

In case an attack attempt is detected after the analysis of the recorded branches, the process is terminated and the user is informed with an alert message, as shown in Fig. 9. In this example, kBouncer blocks a malicious PDF sample that exploits an (at the time of writing)

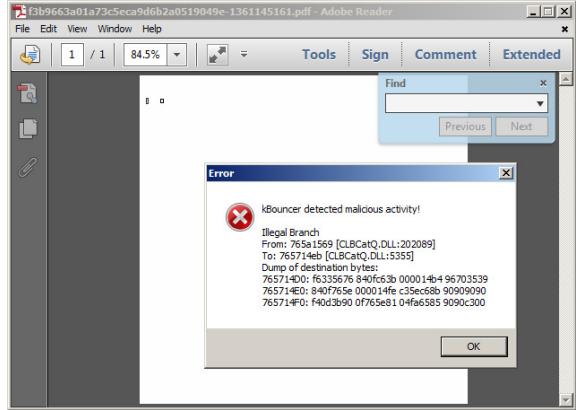


Figure 9: A screen capture of kBouncer in action, blocking a zero-day exploit against Adobe Reader XI [19].

unpatched vulnerability in the latest version of Adobe Reader XI [19]. The displayed information, such as branch locations and targets, is supplied from the kernel-level module.

4.2 Analysis Framework

Moving from the basic concept to a functional prototype required a number of decisions that were mostly based on analyzing the behavior of large applications. To ease the effort required to perform this type of analysis, we developed an LBR analysis framework. Its goal is to provide a way to iterate over the LBR instances during the lifetime of an application, while at the same time providing useful information, such as translating addresses to function or image names. The framework is split in two parts: data gathering and analysis.

The data-gathering component is based on dynamic binary instrumentation. Although the runtime overhead of dynamic instrumentation is quite high (as discussed in Sec. 2.1), we use it here only for data gathering, which is an off-line and one-time operation. The tool we developed is built on top of Pin [64, 46], and records the following information during process execution: i) the file path and starting and ending address of any loaded executable image, ii) the location and name of any recognized function (e.g., exported functions), iii) the thread ID, location, and target of executed indirect branches (`ret`, `call` or `jmp`), iv) the thread ID, location, and number of system calls, and v) the thread ID, location, and return address of any identified function that was called.

The analysis part is a set of Python scripts that process the gathered data for each application. It provides a configurable LBR iterator which simulates different scenarios, such as returning LBR stack instances before system calls or certain function calls, or even after each branch is

Type	# iter.	Avg. ms	Total Time (stddev)	Single ns
HashLookup	1B	8231.6	(9.8)	8.2
IllegalRet	1B	10889.9	(312.9)	10.8
SysNull	10M	5145.0	(66.0)	514.5
SysLBR	10M	19981.8	(504.5)	1998.1
SysRead	10M	47267.7	(30925.6)	4726.7

Table 3: Microbenchmarks.

executed. To avoid mixing branches from different system threads in the same LBR instance, it internally keeps a list of separate LBRs per thread id. Finally, it provides convenient methods to translate addresses to function or image names when available.

5 Evaluation

In this section we present the results of our experimental evaluation of kBouncer in terms of runtime overhead and effectiveness against real-world ROP exploits. All experiments were performed on a computer with the following specifications: Intel i7 2600S CPU, 8GB RAM, 128GB SSD, 64-bit Windows 7 Professional SP1.

5.1 Performance Overhead

5.1.1 Microbenchmarks

We started with some micro-benchmarks of different parts of kBouncer’s functionality. Specifically, we measure the average time needed for the following operations, also listed in Table 3: hash table lookups (“HashLookup”), checks for illegal returns (“IllegalRet”), performing a system call (“SysNull”), reading the contents of the LBR stack (“SysLBR”), and reading parts of a process’ address space (“SysRead”).

In each case, we isolated the measured operation and tried to make the experiment as realistic as possible. For example, we extracted the hash table characteristics (domain size, hash table size, hit ratio) based on the dataset shown in Table 2. The data we used for the illegal return checks come from `kernel32.dll`, and use a worst-case workload by treating each location in its code segment as a possible return target. The next three experiments were measured in kernel level, as opposed to the first two. We measured the time needed to perform a no-op system call, a system call that only reads the LBR stack contents, and finally, a system call that in addition to reading the LBR stack, also fetches data from the sources and targets of each branch.

Table 3 shows the results of these benchmarks. Each benchmark runs the number of operations shown in the second column ten times, and calculates the average and

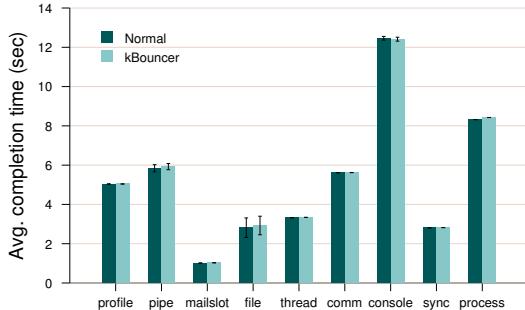


Figure 10: Execution time with and without kBouncer for Wine’s `kernel32.dll` test suite, which resulted in the invocation of about 100K monitored Windows API functions. The average runtime overhead is 1%.

standard deviation (next two columns). The last column shows the average time for a single operation. As we can see, looking up the hash table and checking for an illegal return are both very fast operations, in the order of a few nanoseconds. Performing a system call and reading the LBR stack are relatively more expensive, but still, in the order of a few microseconds. When attempting to access the instructions located at the source and target addresses of each branch record, the measured duration starts to fluctuate. We are not sure whether this behavior is normal, or it is a result of non-optimal use of the kernel API for accessing user-level memory. Overall, these microbenchmarks show that kBouncer’s LBR stack analysis on each protected API function call takes on average no more than 5 microseconds.

5.1.2 Runtime Overhead

Measuring the performance overhead impact on interactive applications, such as web browsers and document viewers, is a challenging task. Instead, we decided to measure the performance overhead on programs that stress the core functionality of kBouncer, by making heavy use of the monitored Windows API functions. For this purpose, we used a subset of the tests provided in the test suite of Wine [15], which repeatedly call Windows API functions with different arguments. To get more confident timing results, we kept only tests that do not interfere with external factors, such as network communication. The final set we used performs about 100,000 calls to Windows API functions that are protected by kBouncer, which is 20 times more than the protected calls made by the actual applications we previously tested (listed in Table 2).

Figure 10 shows the completion time for each of the different tests, with and without kBouncer. The average runtime overhead is 1%, with the maximum being 4%

Application	Vulnerability
Adobe Reader v11.0.1	Function pointer overwrite [19]
Adobe Reader v9.3.4	Stack-based overflow [1]
MPlayer Lite r33064	SEH pointer overwrite [6]
Internet Explorer 8	Use-after-free vulnerability [8]
Internet Explorer 9	Use-after-free vulnerability [7]
Adobe Flash 11.3.300	Integer overflow [2]

Table 4: Tested ROP exploits.

in the worst case. The total extra time spent across all tests when enabling kBouncer was 0.3 sec, a result consistent with the average cost of $5 \mu\text{s}$ per check based on our microbenchmarks ($100,000 \text{ calls} \times 5 \mu\text{s} = 0.5 \text{ sec}$). Based on these results, which show that the performance overhead is negligible even for workloads that continuously trigger the core detection component, we believe that kBouncer is not likely to cause any noticeable impact on user experience.

5.2 Effectiveness

In the final part of our evaluation, we tested whether our prototype can effectively protect applications that are typically targeted by in-the-wild attacks, using the ROP exploits shown in Table 4. All exploits except the ones against Internet Explorer work on the latest and up-to-date version of Windows 7 Professional SP1 64-bit. For the IE exploits to work, we had to uninstall the updates that fixed the relevant vulnerabilities (KB2744842 and KB2799329). We also had to tweak the ROP payload of the MPlayer exploit to correctly calculate the offset of `VirtualProtect` for the latest version of `kernel32.dll`, as the public version of the exploit was based on a previous version of that DLL.

The ROP code in the exploit against Adobe Reader v9.3.4 creates a file (`CreateFileA`), memory-maps the file in RWX mode (`CreateFileMappingA`, `MapViewOfFile`), copies the shellcode in the newly mapped area, and executes it. Similarly, the MPlayer and IE 8 exploits change the permissions of the memory region where the shellcode resides to RWX (`VirtualProtect`) and execute it. What is interesting about the IE 8 ROP code, is that it is constructed from the statically loaded Skype protocol handler DLL (`skype4com.dll`). The last two exploits in Table 4 were generated using the Metasploit Framework [5]. For vulnerable applications that include widely used non-ASLR modules (like Java’s `msvcrt71.dll`, which is loaded in Internet Explorer), Metasploit uses the same ROP payload based on `msvcrt71.dll`, which has been pre-generated by Mona [27]. This payload is similar to the one used in the MPlayer exploit, as it also uses `VirtualProtect` to bypass Data Execution Prevention (DEP). Finally, the Adobe Reader XI (v11.0.1) exploit is more complex,

as it is the first in-the-wild exploit that uses ROP-only code, i.e., it does not carry any shellcode [19]. The malicious sample we tested (“Visaform Turkey.pdf”) exploits a first vulnerability to escape from Reader’s sandboxed process, and a second one to hijack the execution of its privileged process by loading a malicious DLL using `LoadLibraryW`.

In the first five exploits, the embedded shellcode simply invokes `calc.exe` using `WinExec`. The Reader XI exploit drops a malicious DLL. In all cases, we verified that the exploits worked properly on our testbed, by confirming that the calculator was successfully launched, or, for the Reader XI exploit, that the malicious DLL was loaded successfully. When kBouncer was enabled, it successfully blocked all exploits due to the identification of illegal returns at the time one of the `CreateFileA`, `VirtualProtect` or `LoadLibraryW` functions was invoked by the ROP code in each case.

6 Limitations

The Last Branch Recording feature of recent Intel processors is what enables kBouncer to achieve its transparent and low-overhead operation. Many of our design decisions are corollaries of the very limited size of the LBR stack, which in the most recent processors holds only 16 records. Given that previous processor generations had even more size-constrained LBR implementations, this is definitely a significant improvement, and hopefully future processors will support even larger LBR stacks. This would allow kBouncer to achieve even higher accuracy by inspecting longer execution paths, making potential evasion attempts even harder.

Currently, an attacker could evade kBouncer by ensuring that the final 16 executed gadgets before the invocation of an API function are considered legitimate. Specifically, given that kBouncer looks for both illegal returns and gadget chaining in parallel, this would require i) all 16 gadgets to be either `call`-preceded or `non-ret` gadgets, and ii) at least one out of every eight of them (eight is our current gadget chaining detection threshold) to be longer than 20 instructions.

A more thorough analysis on the feasibility of constructing such a payload for typical applications is part of our future work. Our preliminary evidence (Section 3.1), however, shows that only 6.4% of all gadgets ending with `ret` are `call`-preceded, and this is when considering even fragmented gadgets up to 20 instructions long (this percentage drops to 3% when considering gadgets with at most five instructions). On the other hand, ROP compilers like Q [60] typically take into account non-fragmented gadgets up to five instructions long. Longer gadgets incur more CPU state changes, which complicate the (either manual or automated) gadget arrange-

ment process. Indicatively, for a similar set of applications, even when 20% of all gadgets are available, Q could not generate a functional payload [53]. Note that the selection of a maximum gadget length of 20 instructions was arbitrary—four times the typically used standard seemed enough. If evasion becomes an issue, longer gadgets could be considered during the gadget chaining analysis of an LBR snapshot.

Alternatively, an attacker could look for a long-enough execution path that leads to the desired API call as part of the application’s logic. Such a path should satisfy the following constraints: i) contain at least 16 indirect branches, the targets of which happen to lead to the execution of the desired API function, and ii) the executed code along the path should not alter the state or the function arguments set by the previously executed ROP code. Finding such a path seems quite challenging, as in many cases the desired function might not be imported at all, and the path should end up with the appropriate register values and arguments to properly invoke the function. This is even more difficult in 64-bit systems, where the first four parameters are passed through registers, as opposed to the 32-bit standard calling conventions in which parameters are passed through the stack.

Our selection of sensitive Windows API functions was made empirically based on a large set of different shellcode and ROP payload implementations [5, 3, 56, 12, 27, 60]. A list of the 52 currently protected functions is provided in the appendix. Although current ROP exploits rely mainly on only a handful of API functions (see Sec. 5.2), we have included many others that have been used in the past in legacy shellcode, as some exploits might implement their whole functionality using purely ROP code (as demonstrated recently by an exploit against the latest version of Adobe Reader XI [19]). The set of protected functions can be easily extended with any additional potentially sensitive functions that we might have left out. Although it would be possible to protect all Windows API calls, we believe that this would not offer any additional protection benefits, and would just introduce unnecessary overhead.

7 Related Work

Address Space Randomization and Code Diversification As code-reuse attacks require precise knowledge of the structure and location of the code to be reused, diversifying the execution environment or even the program code itself is a core concept in preventing code-reuse exploits [26, 33]. Address space layout randomization [55, 49] is probably one of the most widely deployed countermeasures against code-reuse attacks. However, its effectiveness is hindered by code segments left in static locations [35, 75, 40], while, depending on the ran-

domization entropy, it might be possible to circumvent it using brute-force guessing [63]. Even if all the code segments of a process are fully randomized, vulnerabilities that allow the leakage of memory contents can enable the calculation of the base address of a DLL at runtime [19, 61, 44, 69, 37, 66].

Intra-DLL randomization at the function [20, 21, 42, 9], basic block [11, 10], or instruction level [53, 36, 70] can provide protection for executables that do not support ASLR, or against de-randomization attacks through memory leaks. The practical deployment of these techniques for the protection of third-party applications depends on the availability of source code [20, 21, 42, 9], debug symbols [11, 10], or the accuracy of disassembly and control flow graph extraction [53, 36, 70, 74].

As kBouncer is completely transparent to user applications, it can complement all above randomization techniques as an additional mitigation layer against ROP exploits, while it does not depend on source code, debug symbols, or code disassembly.

Control Flow Integrity and Indirect Branch Protection The execution of ROP code disrupts the normal call path of typical programs, resulting to an unanticipated flow of control. Control flow integrity [17] can confine program execution within the bounds of a pre-computed profile of allowed control flow paths, and thus can prevent most of the irregular control flow transfers that connect the gadgets of a ROP exploit. Depending on program complexity, however, deriving an accurate view of the control flow graph is often challenging. Alternative approaches against return-oriented programming enforce a more relaxed policy for the integrity of indirect control transfers [52, 45, 22]. Using code transformations, these techniques eliminate the occurrence of unintended indirect branch instructions in the generated code, and safeguard all legitimate indirect branches using cookies or additional levels of indirection.

The main factor that limits the practical applicability of the above techniques is that they require the re-compilation of the target application, which is usually not possible for the popular proprietary applications that are commonly targeted by ROP exploits. In contrast, kBouncer is completely transparent to applications and does not require any modification to their code.

Runtime Execution Monitoring Many defenses against return-oriented programming are based on monitoring program execution at the instruction level. A widely used mechanism for this purpose is dynamic binary instrumentation (DBI), using frameworks such as Pin [46]. DROP [24] and DynIMA [28] follow this approach to monitor the frequency of `ret` instructions, and raise an alert in case irregularly many of them are

observed within a small window of executed instructions. ROPdefender [29] also uses DBI to keep a shadow stack that is updated by instrumenting `call` and `ret` instructions. A disruption of the expected `call-ret` pairs due to ROP code is detected by comparing the shadow stack with the system’s stack on every function exit. A limitation of the above techniques is that they cannot prevent exploits that use gadgets ending with indirect `jmp` or `call` instructions. More importantly, though, the significant runtime overhead imposed by the additional instrumentation instructions and the DBI framework itself limit their practical applicability.

Similarly to kBouncer, ROPGuard [34] is based on the observation that a ROP exploit will eventually invoke critical API functions, and performs various checks before such a function is called. These include checking whether `esp` is within the proper stack boundaries, whether a proper return address is present at the top of the stack, the consistency of stack frames, and other function-specific attributes. Although ROPGuard focuses only on non-JOP code, and some of its checks can result in false positives or can be easily evaded [58, 57], they are effective against current in-the-wild exploits, and some have been integrated in EMET [47].

Last branch recording is only one of the available instruction tracing facilities available in modern CPUs. Branch Trace Storage (BTS) is a debugging mechanism that enables the recording of all branch instructions in a user-defined memory area. However, the overhead due to the significant number of memory accesses, combined with the overall slower operation of the processor due to the special debug mode in which it enters when BTS is enabled, result to slowdowns typically in the range of 20–40x [67]. Consequently, systems that use BTS and similar mechanisms for control flow integrity [72, 73] or execution recording [68] suffer from significant runtime overheads. In contrast, LBR uses on-chip registers to store the traced branches with no additional overhead.

A recent technique against kernel-level ROP uses the processor’s performance counters to raise an interrupt after a number of mispredicted `ret` instructions, an indication of possible ROP code execution [71]. To rule out mispredictions caused by legitimate code, upon an interrupt, the LBR stack is used to check whether the targets of the previously executed `ret` instructions are preceded by a `call` instruction. The use of JOP or call-preceded gadgets, however, can circumvent this protection.

Branch regulation [41] is a proposal for extending current processor architectures with a protection mechanism against ROP attacks. Besides maintaining a secondary call stack, the technique restricts the allowed targets of indirect `jmp` instructions to locations within the same function, or to the entry point of any other function, and only the latter for `call` instructions. Besides being quite

restrictive for many legitimate programs, this approach requires protected binaries to go through a static binary instrumentation phase for annotating function boundaries, a process that requires precise code disassembly.

8 Conclusion

Exploit mitigation add-ons that can be readily enabled for the protection of already installed applications are among the most practical ways for deploying additional layers of defenses on existing systems. To be usable in practice, any such solution should be completely transparent and should not impact in any way the normal operation of the protected applications.

Starting on this basis, we have presented the design and implementation of kBouncer, a transparent ROP exploit mitigation based on the identification of distinctive attributes of return-oriented or jump-oriented code that are inherently exhibited during execution. Built on top of the Last Branch Recording (LBR) feature of recent processors for tracking the execution of indirect branches at critical points during the lifetime of a process, kBouncer introduces negligible runtime overhead, and does not require any modifications to the protected applications. We believe that the most important advantage of the proposed approach is its practical applicability. We demonstrate that our prototype implementation for Windows 7 can effectively protect complex, widely used applications, including Internet Explorer, Adobe Flash Player, and Adobe Reader, against in-the-wild ROP exploits, without any false positives.

As part of our future work, we plan to perform a more extensive evaluation with real applications to ensure the compatibility of the detection checks with existing code, assess the feasibility of constructing ROP payloads that could evade the currently implemented checks, and port our prototype implementation to Linux.

Acknowledgements

This work was supported by DARPA, the US Air Force, and ONR through Contracts DARPA-FA8750-10-2-0253, AFRL-FA8650-10-C-7024 and N00014-12-1-0166, respectively, with additional support from Intel. 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, ONR, or Intel.

References

- [1] Adobe CoolType SING Table “uniqueName” Stack Buffer Overflow. http://www.metasploit.com/modules/exploit/windows/fileformat/adobe_cooltype_sing.
- [2] Adobe Flash Player 11.3 Kern Table Parsing Integer Overflow. http://www.metasploit.com/modules/exploit/windows/browser/adobe_flash_otf_font.
- [3] Common Shellcode Naming Initiative. <http://nepenthes.carnivore.it/csni>.
- [4] Intercepting System Calls on x86_64 Windows. http://jbremer.org/intercepting-system-calls-on-x86_64-windows/.
- [5] Metasploit framework. <http://www.metasploit.com>.
- [6] Mplayer (r33064 lite) buffer overflow + rop exploit. <http://www.exploit-db.com/exploits/17124/>.
- [7] MS12-063 Microsoft Internet Explorer execCommand Use-After-Free Vulnerability. http://www.metasploit.com/modules/exploit/windows/browser/ie_execcommand_uaf.
- [8] MS13-008 Microsoft Internet Explorer CButton Use-After-Free Vulnerability. <http://www.greyhathacker.net/?p=641>.
- [9] /ORDER (put functions in order). <http://msdn.microsoft.com/en-us/library/00kh39zz.aspx>.
- [10] Profile-guided optimizations. <http://msdn.microsoft.com/en-us/library/e7k32f4k.aspx>.
- [11] Syzygy - profile guided, post-link executable reordering. <http://code.google.com/p/sawbuck/wiki/SyzygyDesign>.
- [12] White Phosphorus Exploit Pack. <http://www.whitephosphorus.org/>.
- [13] Windows api list. [http://msdn.microsoft.com/en-us/library/windows/desktop/ff818516\(v=vs.85\).aspx](http://msdn.microsoft.com/en-us/library/windows/desktop/ff818516(v=vs.85).aspx).
- [14] Windows X86 System Call Table. <http://j00ru.vexillium.org/ntapi/>.
- [15] Wine. <http://www.winehq.org>.
- [16] MWR Labs Pwn2Own 2013 Write-up - Webkit Exploit, 2013. <http://labs.mwrinfosecurity.com/blog/2013/04/19/mwr-labs-pwn2own-2013-write-up---webkit-exploit/>.
- [17] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow integrity. In *Proceedings of the 12th ACM conference on Computer and Communications Security (CCS)*, 2005.
- [18] Piotr Bania. Windows Syscall Shellcode, 2005. <http://www.securityfocus.com/infocus/1844>.
- [19] James Bennett, Yichong Lin, and Thoufique Haq. The Number of the Beast, 2013. <http://blog.fireeye.com/research/2013/02/the-number-of-the-beast.html>.
- [20] Eep Bhatkar, Daniel C. DuVarney, and R. Sekar. Address obfuscation: an efficient approach to combat a broad range of memory error exploits. In *In Proceedings of the 12th USENIX Security Symposium*, 2003.
- [21] Sandeep Bhatkar, R. Sekar, and Daniel C. DuVarney. Efficient techniques for comprehensive protection from memory error exploits. In *Proceedings of the 14th USENIX Security Symposium*, August 2005.
- [22] Tyler Bleisch, Xuxian Jiang, and Vince Freeh. Mitigating code-reuse attacks with control-flow locking. In *Proceedings of the 27th Annual Computer Security Applications Conference (ACSAC)*, 2011.
- [23] Stephen Checkoway, Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, Hovav Shacham, and Marcel Winandy. Return-oriented programming without returns. In *Proceedings of the 17th ACM conference on Computer and Communications Security (CCS)*, 2010.
- [24] Ping Chen, Hai Xiao, Xiaobin Shen, Xinchun Yin, Bing Mao, and Li Xie. DROP: Detecting return-oriented programming malicious code. In *Proceedings of the 5th International Conference on Information Systems Security (ICISS)*, 2009.

- [25] Shuo Chen, Jun Xu, Emre C. Sezer, Prachi Gauriar, and Ravishankar K. Iyer. Non-control-data attacks are realistic threats. In *Proceedings of the 14th USENIX Security Symposium*, August 2005.
- [26] Frederick B. Cohen. Operating system protection through program evolution. *Computers and Security*, 12:565–584, October 1993.
- [27] Corelan Team. Mona. <http://redmine.corelan.be/projects/mona>.
- [28] Lucas Davi, Ahmad-Reza Sadeghi, and Marcel Winandy. Dynamic integrity measurement and attestation: towards defense against return-oriented programming attacks. In *Proceedings of the 2009 ACM workshop on Scalable Trusted Computing (STC)*, 2009.
- [29] Lucas Davi, Ahmad-Reza Sadeghi, and Marcel Winandy. ROPdefender: A practical protection tool to protect against return-oriented programming. In *Proceedings of the 6th Symposium on Information, Computer and Communications Security (ASIACCS)*, 2011.
- [30] Solar Designer. Getting around non-executable stack (and fix). <http://seclists.org/bugtraq/1997/Aug/63>.
- [31] Úlfar Erlingsson. Low-level software security: Attack and defenses. Technical Report MSR-TR-07-153, Microsoft Research, 2007. <http://research.microsoft.com/pubs/64363/tr-2007-153.pdf>.
- [32] Scott Field. An introduction to kernel patch protection. <http://blogs.msdn.com/b/windowsvistasecurity/archive/2006/08/11/695993.aspx>.
- [33] S. Forrest, A. Somayaji, and D. Ackley. Building diverse computer systems. In *Proceedings of the 6th Workshop on Hot Topics in Operating Systems (HotOS-VI)*, 1997.
- [34] Ivan Fratric. Runtime prevention of return-oriented programming attacks, 2012. <https://code.google.com/p/ropguard/>.
- [35] Giampaolo Fresi Roglia, Lorenzo Martignoni, Roberto Paleari, and Danilo Bruschi. Surgically returning to randomized lib(c). In *Proceedings of the 25th Annual Computer Security Applications Conference (ACSAC)*, 2009.
- [36] Jason Hiser, Anh Nguyen-Tuong, Michele Co, Matthew Hall, and Jack W. Davidson. ILR: Where'd my gadgets go? In *Proceedings of the 33rd IEEE Symposium on Security & Privacy (S&P)*, 2012.
- [37] Ralf Hund, Carsten Willem, and Thorsten Holz. Practical timing side channel attacks against kernel space ASLR. In *Proceedings of the 34th IEEE Symposium on Security & Privacy (S&P)*, 2013.
- [38] Galen Hunt and Doug Brubacher. Detours: Binary Interception of Win32 Functions. In *Proceedings of the 3rd USENIX Windows NT Symposium*, 1999.
- [39] Intel. Intel 64 and IA-32 architectures software developer's manual, volume 3B: System programming guide, part 2. <http://www.intel.com>.
- [40] Richard Johnson. A castle made of sand: Adobe Reader X sandbox. CanSecWest, 2011.
- [41] M. Kayaalp, M. Ozsoy, N. Abu-Ghazaleh, and D. Ponomarev. Branch regulation: Low-overhead protection from code reuse attacks. In *Proceedings of the 39th Annual International Symposium on Computer Architecture (ISCA)*, pages 94–105, 2012.
- [42] Chongkyung Kil, Jinsuk Jun, Christopher Bookholt, Jun Xu, and Peng Ning. Address space layout permutation (ASLP): Towards fine-grained randomization of commodity software. In *Proceedings of the 22nd Annual Computer Security Applications Conference (ACSAC)*, 2006.
- [43] Sebastian Krahmer. x86-64 buffer overflow exploits and the borrowed code chunks exploitation technique. <http://www.suse.de/~krahmer/no-nx.pdf>.
- [44] Haifei Li. Understanding and exploiting Flash ActionScript vulnerabilities. CanSecWest, 2011.
- [45] Jinku Li, Zhi Wang, Xuxian Jiang, Michael Grace, and Sina Bahram. Defeating return-oriented rootkits with “return-less” kernels. In *Proceedings of the 5th European conference on Computer Systems (EuroSys)*, 2010.
- [46] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Arthur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, 2005.
- [47] Microsoft. The Enhanced Mitigation Experience Toolkit. <http://www.microsoft.com/emet>.
- [48] Microsoft. Windows filtering platform. [http://msdn.microsoft.com/en-us/library/windows/desktop/aa366510\(v=vs.85\).aspx](http://msdn.microsoft.com/en-us/library/windows/desktop/aa366510(v=vs.85).aspx).
- [49] Matt Miller, Tim Burrell, and Michael Howard. Mitigating software vulnerabilities, July 2011. <http://www.microsoft.com/download/en/details.aspx?displaylang=en&id=26788>.
- [50] Nergal. The advanced return-into-lib(c) exploits: PaX case study. *Phrack*, 11(58), December 2001.
- [51] Tim Newsham. Non-exec stack, 2000. <http://seclists.org/bugtraq/2000/May/90>.
- [52] Kaan Onarlioglu, Leyla Bilge, Andrea Lanzi, Davide Balzarotti, and Engin Kirda. G-Free: defeating return-oriented programming through gadget-less binaries. In *Proceedings of the 26th Annual Computer Security Applications Conference (ACSAC)*, 2010.
- [53] Vasilis Pappas, Michalis Polychronakis, and Angelos D. Keromytis. Smashing the gadgets: Hinder return-oriented programming using in-place code randomization. In *Proceedings of the 33rd IEEE Symposium on Security & Privacy (S&P)*, 2012.
- [54] Parvez. Bypassing Microsoft Windows ASLR with a little help by MS-Help, August 2012. <http://www.greyhathacker.net/?p=585>.
- [55] PaX Team. Address space layout randomization. <http://pax.grsecurity.net/docs/aslr.txt>.
- [56] Michalis Polychronakis, Kostas G. Anagnostakis, and Evangelos P. Markatos. An empirical study of real-world polymorphic code injection attacks. In *Proceedings of the 2nd USENIX Workshop on Large-scale Exploits and Emergent Threats (LEET)*, April 2009.
- [57] Aaron Portnoy. Bypassing all of the things. SummerCon, 2013.
- [58] Dan Rosenberg. Defeating Windows 8 ROP Mitigation, 2011. <http://vulnfactory.org/blog/2011/09/21/defeating-windows-8-rop-mitigation/>.
- [59] Mark Russinovich. Inside native applications, November 2006. <http://technet.microsoft.com/en-us/sysinternals/bb897447.aspx>.
- [60] Edward J. Schwartz, Thanassis Avgerinos, and David Brumley. Q: Exploit hardening made easy. In *Proceedings of the 20th USENIX Security Symposium*, 2011.
- [61] Fermin J. Serna. CVE-2012-0769, the case of the perfect info leak, February 2012. http://zhodiac.hispahack.com/my-stuff/security/Flash_ASLR_bypass.pdf.
- [62] Hovav 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)*, 2007.

- [63] Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagedra Modadugu, and Dan Boneh. On the effectiveness of address-space randomization. In *Proceedings of the 11th ACM conference on Computer and Communications Security (CCS)*, 2004.
- [64] Alex Skaletsky, Tevi Devor, Nadav Chachmon, Robert Cohn, Kim Hazelwood, Vladimir Vladimirov, and Moshe Bach. Dynamic program analysis of microsoft windows applications. In *International Symposium on Performance Analysis of Software and Systems*, 2010.
- [65] Skape. Understanding windows shellcode, 2003. <http://www.hick.org/code/skape/papers/win32-shellcode.pdf>.
- [66] Kevin Z. Snow, Lucas Davi, Alexandra Dmitrienko, Christopher Liebchen, Fabian Monroe, and Ahmad-Reza Sadeghi. Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization. In *Proceedings of the 34th IEEE Symposium on Security & Privacy (S&P)*, 2013.
- [67] Mary Lou Soffa, Kristen R. Walcott, and Jason Mars. Exploiting hardware advances for software testing and debugging (nier track). In *Proceedings of the 33rd International Conference on Software Engineering (ICSE)*, 2011.
- [68] A. Vasudevan, Ning Qu, and A. Perrig. Xtrec: Secure real-time execution trace recording on commodity platforms. In *Proceedings of the 44th Hawaii International Conference on System Sciences (HICSS)*, 2011.
- [69] Peter Vreugdenhil. Pwn2Own 2010 Windows 7 Internet Explorer 8 exploit. <http://vreugdenhilresearch.nl/Pwn2Own-2010-Windows7-InternetExplorer8.pdf>.
- [70] Richard Wartell, Vishwath Mohan, Kevin W. Hamlen, and Zhiqiang Lin. Binary stirring: Self-randomizing instruction addresses of legacy x86 binary code. In *Proceedings of the 19th ACM Conference on Computer and Communications Security (CCS)*, pages 157–168, October 2012.
- [71] Georg Wicherski. Taming ROP on Sandy Bridge. SyScan, 2013.
- [72] Yubin Xia, Yutao Liu, Haibo Chen, and Binyu Zang. CFIMon: Detecting violation of control flow integrity using performance counters. In *Proceedings of the 42nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2012.
- [73] Liwei Yuan, Weichao Xing, Haibo Chen, and Binyu Zang. Security breaches as PMU deviation: detecting and identifying security attacks using performance counters. In *Proceedings of the Second Asia-Pacific Workshop on Systems (APSys)*, 2011.
- [74] Chao Zhang, Tao Wei, Zhao Feng Chen, Lei Duan, László Szekeres, Stephen McCamant, Dawn Song, and Wei Zou. Practical control flow integrity & randomization for binary executables. In *Proceedings of the 34th IEEE Symposium on Security & Privacy (S&P)*, 2013.
- [75] Dino A. Dai Zovi. Practical return-oriented programming. SOURCE Boston, 2010.

Appendix

In our current prototype implementation, kBouncer protects the following 52 Windows API functions:

kernel32.dll

```
CloseHandle
CreateFileA
CreateFileMappingA
CreateFileMappingW
CreateFileW
```

```
CreateProcessA
CreateProcessW
DeleteFileA
DeleteFileW
DuplicateHandle
ExitProcess
ExitThread
GetCurrentProcess
GetProcAddress
GetSystemDirectoryA
GetSystemDirectoryW
GetTempPathA
GetTempPathW
LoadLibraryA
LoadLibraryW
PeekNamedPipe
ReadFile
SetUnhandledExceptionFilter
Sleep
VirtualAlloc
VirtualProtect
WaitForSingleObject
WinExec
WriteFile
```

ws2_32.dll

```
accept
bind
closesocket
connect
ioctlsocket
listen
recv
send
socket
WSASocketA
WSASocketW
WSAStartup
```

wininet.dll

```
InternetOpenA
InternetOpenUrlA
InternetOpenUrlW
InternetOpenW
InternetReadFile
```

msvcrt.dll

```
_execv
fclose
fopen
fwrite
```

urlmon.dll

```
URLDownloadToFileA
URLDownloadToFileW
```

FIE on Firmware: Finding Vulnerabilities in Embedded Systems using Symbolic Execution

Drew Davidson

Benjamin Moench

Somesh Jha

Thomas Ristenpart

University of Wisconsin–Madison, {davidson, bsmoench, jha, rist}@cs.wisc.edu

Abstract

Embedded systems increasingly use software-driven low-power microprocessors for security-critical settings, surfacing a need for tools that can audit the security of the software (often called firmware) running on such devices. Despite the fact that firmware programs are often written in C, existing source-code analysis tools do not work well for this setting because of the specific architectural features of low-power platforms. We therefore design and implement a new tool, called FIE, that builds off the KLEE symbolic execution engine in order to provide an extensible platform for detecting bugs in firmware programs for the popular MSP430 family of microcontrollers. FIE incorporates new techniques for symbolic execution that enable it to verify security properties of the simple firmwares often found in practice. We demonstrate FIE’s utility by applying it to a corpus of 99 open-source firmware programs that altogether use 13 different models of the MSP430. We are able to verify memory safety for the majority of programs in this corpus and elsewhere discover 21 bugs.

1 Introduction

Embedded microprocessors are already ubiquitous, providing programmatic control over critical, increasingly Internet-connected physical infrastructure in consumer devices, automobiles, payment systems, and more. Typical low-power embedded systems combine a software-driven microprocessor, together with peripherals such as sensors, controllers, etc. The software on such devices is referred to as *firmware*, and it is most often written in C.

The use of firmware exposes embedded systems to the threat of software vulnerabilities, and researchers have recently discovered exploitable vulnerabilities in a wide variety of deployed embedded firmware programs [12, 18, 19, 21, 22, 24, 27]. These bugs were found using a combination of customized fuzz testing and manual reverse engineering, requiring large time investments by those with rare expertise.

To improve firmware security, one possible approach would be to use the kinds of source-code analysis tools that have been successful in more traditional desktop and server settings (e.g., [2, 4, 8, 9, 11, 13, 17, 26, 28, 31, 36]). These tools, however, prove insufficient for analyzing firmware: the microcontrollers used in practice have a wide range of architectures, the nuances of which frustrate tools designed with other architectures in mind (most often x86). Firmware also exhibits characteristics dissimilar to more traditional desktop and server programs, such as frequent interrupt-driven control flow and continuous interaction with peripherals. All this suggests the need to develop new analysis tools for this setting.

We initiate work in this space by building a system, called FIE, that uses symbolic execution to audit the security of firmware programs for the popular MSP430 family of 16-bit microcontrollers. We have used FIE to analyze 99 open-source firmware programs written in C and of varying code complexity. To do so, FIE had to support 13 different versions of the MSP430 family of 16-bit RISC processors. Our analyses ultimately found 20 distinct memory-safety bugs and one peripheral-misuse bug.

We designed FIE to support analysis of all potential execution paths of a firmware. This means that, modulo standard but important caveats (see Section 6), FIE can verify security properties hold for the relatively simple firmware programs often seen in practice. For example, we verify memory safety for 53 of the 99 firmware programs in our corpus.

Overview of approach: FIE is based on the KLEE symbolic execution framework [10]. In addition to the engineering efforts required to make KLEE work at all for MSP430 firmware programs, we architected FIE to include various features that render it effective for this new domain. First, we develop a modular way to specify the memory layout of the targeted MSP430 variant, the way in which special memory locations related to peripherals should be handled, and when interrupt handlers should

be invoked. This allows analysts to flexibly detail peripheral behavior. We provide a default specification that models worst-case behavior of all peripherals and interrupts. This default enables analysis without any knowledge or access to (simulators of) individual microcontrollers or peripheral components, while ensuring consideration of any possible deployment environment.

Small firmware programs appear to arise frequently (our corpus has many that have less than 100 lines of code) and for these we might hope to achieve complete analyses, meaning all possible paths are checked. Even with very small firmware programs, however, deep or infinite loops arise often and force the analysis to visit already-analyzed states of the symbolic execution. We therefore use a technique called *state pruning* [6], which detects when a program state has been previously analyzed, and if so, removes it from further consideration. Our realization of pruning keeps a history of all changes made to memory at each program point, and while simpler than prior approaches (see Section 7) it proves effective. We also introduce a new technique called *memory smudging*, which heuristically identifies loop counters and replaces them with unconstrained symbolic variables. While smudging can introduce false positives, our experiments show them to be rare. Together, pruning and smudging significantly improve code coverage and support the ability to analyze all possible paths of simpler firmware programs.

Summary: This paper has the following contributions:

- We provide (to the best of our knowledge) the first open-source tool designed for automated security analysis of firmware for the widely used MSP430 microcontrollers.
- We explore use of state pruning and memory smudging to enhance coverage of symbolic execution and to attempt to verify the absence of classes of bugs. Ultimately, FIE is able to verify memory safety on 53 open-source firmware programs.
- FIE found 21 distinct bugs in the firmware corpus, many of which appear to be exploitable memory-safety violations.

To do these analyses at scale, we developed a system for managing FIE-powered analyses on Amazon EC2 [1]. The source code for FIE, the firmware corpus, and the EC2 virtual machine images and associated management scripts will all be made publicly available from the first author’s website.¹

Outline: The remainder of this paper is structured as follows: In Section 2, we give background on embedded systems and the MSP430 family, describe a corpus of open-source firmware that we gathered, and explain

some of the key challenges that must be overcome for use of symbolic execution in our context. We then give a high-level overview of how FIE works in Section 3, and explain its mechanisms in greater detail in Section 4. We evaluate FIE on the corpus of firmware examples and discuss the vulnerabilities found in Section 5. Finally we discuss limitations of FIE in Section 6, related work further in Section 7 and conclude in Section 8.

2 Background and Analysis Targets

Our system, FIE, analyzes embedded firmware programs for the MSP430 family of microcontrollers using *symbolic execution* [2, 8–11, 13, 17, 28, 31, 36]. In this section, we describe details of the MSP430 family, discuss a representative corpus of firmware programs that we gathered, review symbolic analysis, and explore the challenges faced in attempting to use existing tools for analysis of firmware programs.

2.1 MSP430 Microcontrollers

We chose Texas Instruments’ (TI’s) MSP430 family of microcontrollers as our analysis target because of its popularity. MSP430s already find use in security critical applications such as credit-card point of sale systems, smoke detectors, motion detectors, seismic sensors, and more [34]. We believe porting our approach to other, similar low-power microprocessor families would be straightforward.

Architecture and memory layouts: MSP430s use a custom, but simple, RISC instruction set, and have a von Neumann architecture (instructions and data share the same address space) with at least 16-bit addressing. MSP430s have a set of CPU registers, which are accessed via special memory locations. There are over 500 different MSP430 microcontroller products. One example is the MSP430G2x53 series, which consists of 5 different chips. These have from 1 kB to 16 kB of non-volatile flash memory and from 256 to 512 bytes of volatile random access memory. The memory layouts for the different models are distinct, meaning some physical addresses are invalid on one variant while valid on another.

Hardware peripherals: MSP430 microcontrollers are used in conjunction with both built-in and external hardware peripherals. Built-in peripherals include flash memory, timers, power management interfaces and the like, whereas external peripherals (USB hardware, modems, sensors, etc.) must be connected to the microcontroller via I/O pins. MSP430s have a limited number of I/O pins, and so they are multiplexed amongst various functions. Usually, one function is general purpose I/O and the other is an internal function. For applications that need to use many different functions of the device, a given pin may be switched between its multiplexed duties several times during execution. Accessing periph-

¹<http://pages.cs.wisc.edu/davidson/fie>

```

1   FCTL3 = FWKEY;
2   FCTL1 = FWKEY + ERASE;
3   *F_ptr = 0;
4   while (FCTL3 & BUSY);
5   FCTL1 = FWKEY;
6   FCTL3 = FWKEY + LOCK;

```

Figure 1: Code excerpt that clears a flash segment

erals works via memory-mapped I/O or special registers (which are, in turn, accessed via special memory locations). We refer to all memory that serves internal or external peripherals as *special memory*.

Peripherals often have intricate semantics. For example, consider accessing flash memory, which is a built-in peripheral for nearly all MSP430 models. Figure 1 gives a code snippet taken from the USB drivers in our corpus (see next section). This code clears a segment of flash memory using various special registers, which is required before any writes to that segment can occur: flash control register 3 (FCTL3) must be set to the special flash write key (FWKEY) to unlock the memory and allow writes to it, and flash control register 1 (FCTL1) must contain the FWKEY value masked with the particular value to indicate the type of write. Finally, after the memory is erased, the flash memory is re-locked by assigning the special value FWKEY + LOCK.

Firmware programming: Most MSP430 programs are written in C, using one of three compilers recommended by TI: IAR, CCS, and msp430-gcc. The first two compilers are commercial products packaged in IDEs, while the third is a port of the gcc toolchain to the MSP430. Each of these tools provides a number of extensions to C. Unfortunately, the extensions do not agree on a single syntax. As a result, many programs conditionally include code based on the compiler that is being used. we chose to base our tool on the msp430-gcc syntax as it is popular, open-source, and has straightforward extensions.

Embedded firmware usually operates by setting up configuration for the program and then spinning in an infinite loop while waiting for input from the environment. These event-driven programs use interrupt handlers, busy waiting (e.g., line 4 in Figure 1), and the like to drive computation in response to I/O from peripherals, and so interrupt handlers often contain the bulk of the program logic. A typical firmware will initialize several registers specifying which interrupts to activate and then go to sleep either by setting the chip to a low-power sleep mode or by entering an explicit infinite loop.

2.2 Firmware Corpus

As mentioned, MSP430s are used in a wide variety of security-critical applications. The diversity of applications is reflected in the firmware programs found in

practice, ranging from simple programs for controlling some external hardware peripheral on up to feature-rich lightweight operating systems such as Contiki [35]. To have a concrete set of analysis targets and as well educate our design of FIE, we have gathered a corpus of 99 open-source MSP430 firmware programs, which we now discuss further.

Cardreader: The first firmware in our corpus is cardreader, a secure credit card reader designed for the MSP430g2553 and written by one of the authors independently of the development of FIE. This was motivated by recent attacks against smartphone-based point-of-sale MSP430 devices [20]. Our firmware assumes the presence of a magnetic credit card stripe reader attached to port 1 (configured for general purpose I/O), and a UART connection on port 2 (to transmit gathered credit card data). The cardreader gets as input card data from the stripe reader, loads a stored cryptographic key from flash memory, and applies AES encryption to the card data before writing the result to the UART. cardreader is fully functional, with 1,883 lines of C code as computed by the cloc utility, and incorporates many of the MSP430 programming constructs that, as we will see, can thwart traditional symbolic-execution-based analysis. We made no efforts to tailor the code to be amenable to analysis by FIE. We also performed extensive manual audit of the code to verify the absence of memory or peripheral-misuse errors.

USB drivers: We additionally use two USB driver firmware programs, CDC Driver and HID Driver, taken from the TI-supplied USB developers package. These programs include a full USB code stack, and include 7,453 and 7,448 lines of C code, respectively. The particular programs we chose exercise the CDC (Communications Device Class) and HID (Human Interface Device) USB classes, which represent different device types in the USB specification. The CDC-using firmware, for example, takes string commands from an attached terminal program on a host PC, uses these commands to toggle the LED in various ways, and sends back an acknowledgement string to the host device. The code that we tested was written for the IAR compiler, but we manually wrote Makefiles to compile the source code for our analysis.

Community projects and GitHub: In order to increase the size of our corpus, we searched for open source projects both on the TI MSP430 Community Projects website [33] and GitHub. For the former, we manually crawled the website, and downloaded all projects with a Makefile, of which we found 12 that compiled properly. For the latter, we used the GitHub API to automatically download all projects that matched the keyword “msp430”. There were 360 such projects. Of these, we culled out those that either: did not include makefiles,

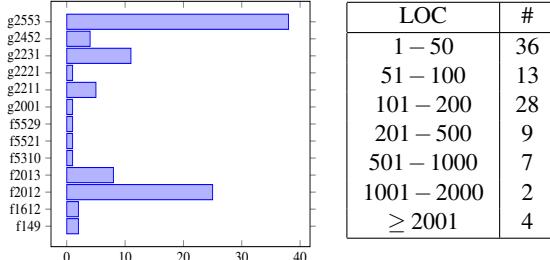


Table 2: Number of firmware program in the corpus (left) targeting the indicated MSP430 models and (right) having the number of lines of C code in the given range.

were not written in C, or did not compile properly for the MSP430 using their given makefiles (this includes projects such as desktop utilities for connection to an MSP430, and thus matched the keyword without being applicable to FIE). After this culling we had 83 firmware programs.

Contiki: Finally, we add to the corpus Contiki [35], which provides an operating system for microcontrollers. To use Contiki, one writes an application against it as a library, which is then statically linked for a complete firmware. Since we need an entrypoint to the library for testing, we use a “hello world” example program included with the Contiki distribution. The resulting C file for the firmware has only 10 lines of code, but this links against other, larger modules. There are over 200,000 lines of C code in the full Contiki source tree. We note that Contiki supports many architectures, including (amongst others) motes that support MSP430x, an extension of the MSP430 that supports 20-bit addresses. FIE only supports basic 16-bit MSP430, and thus cannot run on these motes. Fortunately, Contiki has support for a basic MSP430 backend: the esb, based on the `msp430f1612`. We use this backend in our analysis.

The table in Figure 2 shows a break down of the number of firmware programs whose number of lines of code (computed using `cloc`, including C and C/C++ header files) falls in the indicated range. As can be seen, the range of sizes of these firmware programs is large, but most are 2,000 lines of C code or less. This is not surprising given that MSP430s are often used to drive relatively simple controllers or sensors: our firmware set includes the large number of small hobbyist projects found on GitHub and the TI community projects webpage. A breakdown of the architectures targeted by firmware programs in the corpus is shown in the graph in Figure 2. (When a single firmware supports multiple target architectures, we restrict attention to one, picked arbitrarily.)

2.3 Symbolic Execution and Challenges

To date, finding vulnerabilities in embedded firmware programs has relied upon specialized fuzzing and reverse engineering [12, 18, 20–22, 24], which requires signifi-

cant manual effort and knowledge of the firmware under analysis. Almost all previous research on more general software analysis tools (see Section 7) has not focused on the setting of embedded microcontrollers, and so the relatively unique features of our context (relatively small firmware sizes, large diversity of architectures, and complex environmental interactions) mean that traditional approaches need to be revisited. We initiate such work, focusing in particular on symbolic execution. We feel it to be well-suited to firmware analysis, allowing fine-grained modeling of architectural nuances, flexibility in analysis approach, and typical limitations of symbolic analysis (i.e., scalability) may not prove to be as much of an issue for the small firmware programs seen in practice. We leave exploration of other approaches (e.g., static analysis, concolic execution, etc.) to future work.

Symbolic execution: In symbolic execution, variables corresponding to inputs to a program are treated as symbolic. This means one stores a representation of all of the possible values that each symbolic variable may take on. The program is then executed symbolically using an execution engine. A *symbolic state* (just state from now on) is a current program counter, other register contents, stack frames, and memory contents. The latter three may contain a mix of concrete values or symbolic variables and the constraints over those variables. From an initial state, the engine executes the program one instruction at a time and updates the state appropriately, changing concrete values or possibly adding constraints upon symbolic variables.

Should execution reach a control flow decision such as a branch, the executor uses a SAT solver to determine what are the possible next instructions. A new state is generated for each possible next instruction, with appropriate constraints for the outcome. For example, if a variable x is assigned symbolic variable α (that is unconstrained), and a branch `if ($x < 5$)` is encountered, two child states will be spawned: the first executes inside the if condition with the constraint $\alpha < 5$, and the second executes after the if condition with the constraint $\alpha \geq 5$. Once multiple states are active, the engine decides during each iteration which state to progress, based on some state selection heuristic.

(Complete) analyses: Analysis is performed by investigating each state for violations of some specific properties. A common choice is memory-safety violations, which can be checked by ensuring that all reads and writes are to properly allocated memory ranges. Should a state violate such a property, the execution halts and outputs one of the paths that could lead to this state as well as concrete values that drive the program’s execution along that path. The latter is facilitated by using the SAT solver to provide a solution for the formulas describing constraints on the symbolic variables.

It is well known that symbolic execution can, in theory, provide both sound and complete analyses of some programs. A sound analysis does not emit any false positives — bug reports that are spurious. We refer to a symbolic-execution-based analysis as being *complete* if it covers all of the finitely many possible symbolic states.

Obviously complete analyses are intractable for many programs. Past work on symbolic execution has therefore focused on achieving high *code coverage*, meaning the number of executable lines of code in a program that have been symbolically executed along any path. High-coverage symbolic execution enables finding bugs along the paths that are explored. One can also use the explored paths to generate inputs for use in testing. In our setting of resource-constrained, small firmware programs, there is hope that in addition to high code coverage, we may be able to sometimes achieve complete analyses as well.

Symbolic analysis (even when sound and complete) has inherent limitations, stemming from the possibility of bugs in the analysis engine or compilers used, source code that depends on memory address values, use of inline assembly, etc. We discuss these limitations more in Section 6.

Challenges: We use the symbolic execution system KLEE [10] as the foundation for FIE. Our problem domain, however, necessitates rethinking several aspects of KLEE’s design and use. In particular, we face the following three key challenges:

Challenge 1 (architecture ambiguity): Firmware programs make a number of assumptions about the hardware, including the overall layout of memory and location of memory-mapped hardware controls. These assumptions are not made explicit in a firmware’s source code. For example, it is common for a program to store persistent configuration data at a hard-coded memory address in flash. An architecture-agnostic analysis, or one tailored to x86 environments (as most prior tools are), would view code using this feature as having read from uninitialized memory. Making matters worse, the wide diversity of architectures mean that we will need a way to configure an analysis to the architectures of interest.

Challenge 2 (intensive I/O): Firmware programs are highly interactive with the environment throughout the lifecycle of the program and are designed to interact with a huge diversity of peripherals. Handling external inputs to a program is a well-studied issue in prior symbolic execution contexts, for example KLEE implements functions to determine for the symbolic executor the outcome of (a subset of) common Linux operating system calls. In our setting, the peripheral interface is via special registers and memory-mapped I/O and there exists a huge diversity of potential peripheral behaviors. This makes our setting closer to the one targeted by SymDrive, which uses the S2E [14] symbolic execution system to analyze

x86 Linux kernel drivers without hardware [26]. Like SymDrive, we need to support analysis without (simulators of) peripherals and often without even knowing the intended peripheral. When a peripheral and its behavior are known, we should support the detection of (what we call) peripheral misuse bugs, in which a firmware incorrectly implements the (sometimes complex) operations involving some peripheral.

Challenge 3 (event-driven programming): The event-driven model of programming used for firmware is problematic for symbolic execution because deep or infinite loops are frequent, and most program logic happens in, or as a direct result of, interrupt handlers. S2E and SymDrive both dealt with the frequent use of loops in code, via path selection heuristics or loop elision. These approaches do not allow complete analyses, which we hope to sometimes achieve in our setting. We note that since interrupts are so crucial to the operation of the program, failure to follow possible control flow paths through interrupt handlers will result in very low coverage results. Furthermore, disregarding the circumstances under which interrupts can occur may cause infeasible paths to be explored in the analysis. At the same time, the number of possible paths that can occur in the program due to interrupts causes state space explosion as, in the worst case, we must consider every instruction as a potential branch.

3 Overview of FIE

Our main contribution is FIE, an extensible tool for symbolic-execution-based analysis of MSP430 firmware. It is based on KLEE [10], but with significant modifications and embellishments to the frontend and core engine as we will explain. In this initial work we focus on analyzing memory safety of firmware programs, the lack of which has been exploited in many of the recent security exploits against embedded systems. We also report on detection of some peripheral misuse bugs. Our analyses, by default, use a conservative threat model in which all inputs from peripherals are untrusted.

In a departure from previous symbolic analysis systems, we target the ability to achieve complete analyses for simple firmware programs. These appear to be common and complete analysis in this context is particularly compelling since it means that (subject to various caveats discussed in Section 6) one can verify security or correctness properties of a firmware before deploying it. As far as we are aware, symbolic execution has not before been used for verification, since in most settings it is not feasible under current techniques.

This goal of completeness will guide our design in several ways, as previous optimizations (such as path selection heuristics) do not support this goal. That said, we will not always be able to provide verification, and when

not, FIE will be useful in a more traditional role for bug-finding and test case generation. Here its efficacy will be measured by its ability to provide high-percentage code coverage.

In the remainder of this section, we walk through the workflow of FIE, shown in Figure 3.

FIE frontend: The first step to analyzing firmware is compiling it to a form that can be symbolically executed. The core of this process uses the CLANG compiler to LLVM bitcode. However, there are three necessary features of this process that CLANG alone does not provide: (1) definitions for compiler intrinsics that are not expanded by CLANG; (2) definitions of standard library (stdlib) functions that would normally be included at link time; and (3) definitions of hardware-defined behavior. Handling (1) and (2) is straightforward: we provide a wrapper around CLANG which links pre-compiled bitcode for functions in stdlib and for compiler intrinsics. We took the definitions for the stdlib functions from the msp430-gcc source code, and we manually wrote stubs for the compiler intrinsics. As a convenience to the user, the wrapper also embeds a token into the firmware bytecode to specify which MSP430 variant the firmware is compiled for, which simplifies the FIE command line. Providing definitions for hardware-defined behavior is more involved, since it is often unknown at compile time — for instance, the firmware may interact with a variety of peripheral devices to the chip with varying behaviors. We address this at runtime using an analysis specification, which is described in depth in Section 4.2.

The most predominant compiler in use for our corpus is msp430-gcc, a version of gcc targeting the MSP430. Fortunately the arguments to CLANG are largely compatible with msp430-gcc, so we can drop in our CLANG wrapper in place of msp430-gcc, and use many of the original, unmodified Makefiles included in our corpus.

Core execution engine: Once firmware bitcode has been generated, FIE itself can be run. To analyze the LLVM firmware bytecode file `input.bc`, the user issues the command

```
fie -mmodel=<mem> -imodel=<intr> input.bc
```

The *memory spec* is specified by `mem`, which supplies the semantics of special memory such as attached devices, flash memory, etc. FIE comes with a set of default specifications which conservatively returns unconstrained symbolic values to any read from special memory and ignores writes. However, the user may wish to choose a different specification or write their own. We discuss this process in depth in Section 4.2.

The *interrupt spec* `intr` informs the analysis of when (and which) interrupts should be simulated to have fired at any given point in symbolic execution. Should an interrupt be deemed to fire, the state's execution is pro-

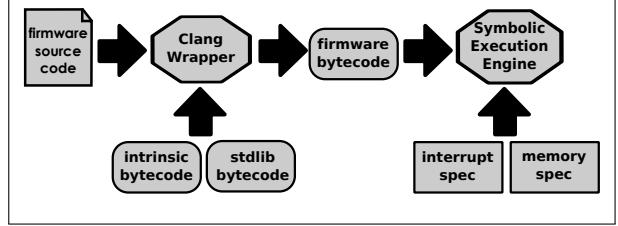


Figure 3: The FIE workflow and system components.

gressed to the appropriate interrupt handler function within the firmware. The interrupt spec allows us to flexibly model different interrupt firing behaviors. Our default is to allow any enabled interrupt to fire at every program point. See Section 4.2.

We inherit as well from KLEE various possible command-line options, so the user can optionally specify the wall-clock time to spend on the analysis, the search heuristic to use, etc.

FIE runs a modified version of the KLEE symbolic execution engine (the executor) to perform the analysis over the firmware bitcode. In particular, we use directly from KLEE their existing state selection heuristics, their underlying SAT solver framework, and much of their state management code. Our major changes include porting the entire execution engine to a 16-bit architecture, which includes a new memory manager to ensure that all memory objects are allocated within a 16-bit value, and the use of the memory spec and interrupt library to model execution when the engine interacts with special memory or fires an interrupt. We also implement two enhancements to the symbolic execution engine, state pruning (Section 4.3), first introduced by RWset [6] and adapted to our domain, and memory smudging (Section 4.4), which is novel to this work. These can improve code coverage and, for some programs, enable complete analyses.

FIE finishes when it completes an analysis by visiting every possible state, hits the requested time limit, or finds a memory-safety (or other) violation. In the latter case it outputs a description of a path leading to the bug. We call this a trace. The trace includes concrete examples of inputs (i.e., from peripherals) that cause the firmware to trigger the bug, and includes at what points in the execution interrupts fired to cause a jump to a specific interrupt handler. Currently, a trace is useful as a debugging log, but eventually it could be used to directly drive an MSP430 simulator to validate the potential bug.

We have additionally prepared an Amazon EC2 [1] virtual machine image and control scripts to run analyses on EC2. This made it easy to automate running FIE on our large corpus of programs. We will publicly release an open-source version of FIE, associated scripts, and the EC2 virtual machine image.

4 Details of FIE’s Architecture

4.1 Main Execution Loop

For the purposes of describing FIE, we define an execution *state* to be an immutable snapshot of the symbolic execution at a given point in time. That means it includes all values used to emulate LLVM bitcode, including a program counter, stack frames, and global memory (used for global variables, the heap, etc.). Any memory location may have either a concrete value or a symbolic one, the latter represented by a set of constraints.

In our abstraction, the main execution loop of FIE operates by generating successor states from the current immutable state. This allows us a history of past states, which, looking ahead, will be useful for describing our state pruning feature. This treatment differs from [10], which instead described states as mutable objects transformed by the symbolic execution.

Figure 4 gives high-level pseudocode for the main execution loop. A set \mathcal{AS} contains the active states to be run; at the start it holds just one initial state. The loop chooses a state from \mathcal{AS} according to a state selection heuristic R . For this we use the KLEE heuristic that seeks to maximize coverage. Once a state has been selected, new successor states may be immediately spawned according to `SpawnnInterrupts`. This function also outputs a boolean $shouldExec$ that can be set to false to force an interrupt to fire, otherwise the instruction at the current state’s program counter is symbolically executed.

Should $shouldExec$ be true, FIE symbolically executes the next instruction of the current state. Here FIE interposes on memory loads and stores that target memory addresses corresponding to special memory (e.g., peripherals). The addresses of special memory are provided by the memory spec as described in the next section. Other operations are handled by `Eval`, which works like KLEE’s evaluation mechanism, except with a new special-memory-aware memory manager, support for emulation of 16-bit firmware, and compiler intrinsics used by `msp430-gcc`.

Each of `SpecLoadEval`, `SpecStoreEval`, and `Eval` must check that security properties are satisfied. Should one fail, a warning will be generated and the set of successors \mathcal{S} output by the evaluation function will be empty. This allows execution to continue, along other paths, even after one path leads to an error.

The set of possible successor states \mathcal{SS} is then taken to be the union of those output by `SpawnnInterrupts` and one of the eval functions. In a normal symbolic execution engine, the full set \mathcal{SS} would be added to \mathcal{AS} . FIE works a bit differently due to state pruning and memory smudging as we explain in Sections 4.3 and 4.4.

```

1:  $\mathcal{AS} = \{S_{init}\}$ 
2: while  $\mathcal{AS} \neq \emptyset$  do
3:   Dequeue  $S$  from  $\mathcal{AS}$  according to  $R$ 
4:    $(shouldExec, \mathcal{S}_{int}) \leftarrow \text{SpawnnInterrupts}(S)$ 
5:   if  $shouldExec$  then
6:     Let  $p$  be the program counter of  $S$ 
7:     Let  $I$  be the instruction pointed to by  $p$ 
8:     if  $I$  is a load to special memory then
9:        $\mathcal{S} \leftarrow \text{SpecLoadEval}(I, S)$ 
10:      else if  $I$  is a write to special memory then
11:         $\mathcal{S} \leftarrow \text{SpecStoreEval}(I, S)$ 
12:      else
13:         $\mathcal{S} \leftarrow \text{Eval}(I, S)$ 
14:    $\mathcal{PS}_p \leftarrow \mathcal{PS}_p \cup \{S\}$ 
15:    $\mathcal{SS} \leftarrow \mathcal{S}_{int} \cup \mathcal{S}$ 
16:   for all  $S' \in \mathcal{SS}$  do
17:     Let  $p'$  be the program counter of  $S'$ 
18:     if  $\text{Prune}(S', \mathcal{PS}_{p'}) = \text{false}$  then
19:        $\mathcal{S}'' \leftarrow \text{MemorySmudge}(S', \mathcal{PS}_{p'})$ 
20:    $\mathcal{AS} \leftarrow \mathcal{AS} \cup \mathcal{S}''$ 

```

Figure 4: Pseudocode of FIE’s main execution loop.

4.2 Modeling Chips and Peripherals

FIE must be aware of various aspects of the target architecture, including what are valid memory addresses, whether they correspond to special memory locations, and how interrupt firing should be simulated. With over 400 chips in the MSP430 family, hard-coding this information would be cumbersome. Instead, FIE is configured at runtime to work for particular models of chips, external peripherals, and interrupt firing. In this section, we discuss the details of writing an analysis specification file, together with a memory spec and interrupt spec. Combined these serve as a layer of abstraction between the symbolic execution engine and the actual target chip’s hardware details.

Analysis specification: When FIE is run, the analyst indicates the target architecture on the command line. In turn FIE loads an associated *analysis specification* file, which is a plaintext file adhering to a simple format and specifying how the analysis should be configured. An example is shown in Figure 5.

Recall that each MSP430 chip has memory locations that correspond to on-chip peripheral addresses. As well there are other hardware specifics, e.g., the location and length of non-volatile flash memory segments. These memory locations differ amongst chips. For example, PORT 0 input resides at memory location 0x0020 on the MSP430G2221, but resolves to 0x200 on the MSP430F5521. This information is generally not included in firmware source-code. The specification file therefore includes information on the layout of memory and what addresses correspond to special memory. In the example, the file fixes the total size of memory on line 1,

```

layout 0x10000
range 0x1080 0x10bf flash
range 0x10c0 0x10ff flash
addr P1IN 0x0020 1
addr P1OUT 0x0021 1
addr P1DIR 0x0022 1
addr P1IFG 0x0023 1
interrupt PORT2_ISR check_PORT2

```

Figure 5: MSP430g2553 analysis specification excerpt

specifies flash regions on lines 2 and 3, and sets the locations and sizes of several special memory addresses on lines 4–7. The final line indicates that the function `check_PORT2` is used to determine when interrupts handled by `PORT2_ISR` fire.

For any MSP430 chip that is supported by `msp430-gcc`, this layout file can be synthesized automatically from firmware source code (for ISRs) and files included in the compiler. While we could therefore have made specifications completely internal, we expose the layout file explicitly to allow an analyst to modify the hardware model if desired.

The chip layout specification explicitly fixes architecture details that are implicit in firmware, but it does not specify the actual behavior of these special features, such as when to fire interrupts and the behavior of special memory. These are handled by the memory and interrupt specifications.

Memory spec: The expected functionality of special memory locations is not available in a firmware, and often not really fixed until the device is deployed with attached hardware peripherals. Thus, FIE uses a library of functions that, together, form a model of special memory behavior. For each special memory location, the memory spec contains a function `n_read` and `n_write`, where n is the name of the special memory location (e.g., `P1IN_read` and `P1IN_write`). The `SpecLoadEval` and `SpecStoreEval` functions determine which of the `n_read` and `n_write` functions to invoke, based on the target address. (Note that the target address may be symbolic, in which case FIE resolves the set of possible addresses, and generates new successors for each possible resulting behavior.)

Read and write functions are passed the entire symbolic execution state, and output a (possibly empty) set of states. This allows special memory reads and writes to define behavior as an arbitrary computation over the state. Security and domain experts can therefore modify a memory spec to refine models of peripheral behavior.

Although this modeling approach is flexible and expressive, previous work has noted that such models can be quite onerous to develop [14]. To eliminate this drawback, we provide a default memory spec which is auto-

matically generated from the analysis spec. For memory reads, the default memory spec returns a fresh, unconstrained symbolic value. For example, reading from `P1IN` always returns a new, unconstrained, symbolic, 8-bit variable, while writing to `P1OUT` is a no-op. This default conservatively assumes that an attacker has full control over all peripherals and uninitialized memory. This means that our analysis often overapproximates special memory behavior, and in particular might lead to finding vulnerabilities that cannot always be exploited when specific peripherals are used. This approach is in-line with similar work on modeling symbolic hardware [26], and as we will see in Section 5, empirically results in few false positives.

Interrupt spec: Deciding which interrupt is enabled at a given program point is nontrivial: the MSP430 design documents specify a partial order of priorities over interrupts, i.e., a higher priority interrupt cannot be preempted by a lower priority one. Furthermore, some (but not all) interrupts are only enabled when appropriate status register flags are set. Thus, determining the enabled set of interrupts requires knowledge not only of the architecture but also the current firmware state.

FIE handles this using an interrupt spec. It contains a number of *gate functions*, one for each possible interrupt that can occur on an MSP430. The `SpawnInterrupts` function executes each gate function, passing each a pointer to the entire execution state. The gate functions return a flag indicating that the interrupt: (1) *cannot* fire at the current instruction (usually indicating that the interrupt is disabled at that program point); or (2) *may* occur at the current program point. For case (2), the gateway function additionally returns a successor state S' that is the same as the current state S except advanced to the first instruction of the associated interrupt handler.

`SpawnInterrupts` collects the returned values produced into a set of successor states \mathcal{S}_{int} that includes one successor for each gateway that returned *may*. As well, `SpawnInterrupts` determines if it's valid for execution to proceed without an interrupt. This reflects the fact that when the firmware is in a sleep state the only valid successor states are in \mathcal{S}_{int} (i.e., the path must traverse an interrupt handler). In this case, `SpawnInterrupts` returns `shouldExec` set to false, correctly forgoing evaluation of S . Otherwise it is set to true, and S is evaluated.

FIE uses, by default, an interrupt spec that explores an over-approximation of all feasible paths: any interrupt that is enabled at a particular program point may fire. Thus, an instruction for which n interrupts may fire will have at least $|\mathcal{S}_{\text{int}}| = n$ successor states, and possibly multiple more in the case that the current state is evaluated. In practice, even an attacker with physical access to the chip is unlikely to be able to exercise all possible firing sequences. This means that FIE using the default

interrupt spec may yield false positives, but without further information about possible adversaries treating all possible firing sequences is necessary for verification.

The default interrupt spec can be used for all the MSP430 variants: if the firmware does not handle a certain type of interrupt, that gate function is simply ignored. However, swapping out interrupt libraries can still be useful as a way to tune the analysis. For example, in Section 5 we evaluate an interrupt spec that, instead of firing at every instruction, allows interrupts only to fire once per basic block. While this relaxed interrupt model misses feasible paths, it improves performance.

4.3 State Pruning

In the course of analysis, the main execution loop will often generate a set \mathcal{SS} including one or more states S' that will execute equivalently to another, already seen state \hat{S} . We call such an S' *redundant*. We denote states that lead to equivalent execution by $S' \approx \hat{S}$ and say S' and \hat{S} are *equivalent*.

Most prior symbolic execution frameworks, including KLEE, simply add redundant states to the set of active states, meaning they will potentially be scheduled for execution later. Consider Figure 4, but modified so that lines 14–18 are replaced by a single line $\mathcal{AS} \leftarrow \mathcal{AS} \cup \mathcal{S}_{\text{int}} \cup \mathcal{S}$. That is, all successors generated via interrupt spawning or evaluation are simply added to the set of active states. We refer to this variant as the PLAIN operating mode of FIE.

Redundant states arise frequently in our setting, and as we will show experimentally in Section 5, PLAIN is slowed down considerably by them. One reason is that interrupt firings can lead to two different paths leading to the same state. Figure 6(a) shows an example interrupt handler and code. At line 1, interrupts are enabled. By the beginning of line 4, when running PLAIN there would be 4 states resulting from the paths $P_1 = \langle s_2, s_3, s_4 \rangle$, $P_2 = \langle s_2, s_7, s_3, s_4 \rangle$, $P_3 = \langle s_2, s_3, s_7, s_4 \rangle$, $P_4 = \langle s_2, s_7, s_3, s_7, s_4 \rangle$, where s_i represents the statement at line i . The states S_4 and S'_4 resulting from execution along paths P_2 and P_3 are equivalent, since both increment a via the interrupt handler once — even though they explore distinct program paths all variables have the same value.

A second source of redundant states arises when symbolic execution of loops generates redundant states. This situation also causes the PLAIN mode of FIE to loop infinitely. Consider when running PLAIN from a state S_3 on the looping line 3 in the code snippet shown in Figure 6(b). The main loop will call `SpecLoadEval` and in turn invoke the memory spec function associated to `P1IN`. An unconstrained symbolic variable will be generated and two successor states will be returned: S_4 set to line 4 (the branch condition assumed to fail) and S'_3 remaining on line 3 (the branch condition succeeded).

When S'_3 runs, it will again generate two new states, S'_4 and S''_3 . Yet, $S'_4 \approx_A S_4$ and $S'_3 \approx_A S''_3$. This will continue endlessly, generating a large number of states and ultimately ensuring that the analysis will never complete.

In KLEE and most prior systems redundant states were dealt with indirectly, by way of state selection heuristics R that favored new lines of code. We would like to support complete analyses, however, and so we go a different route and instead build into FIE the ability to detect and prune redundant states.

State pruning was used previously by RWset [6], which detects if two states S', \hat{S} are equivalent by checking if the set of values taken by all live variables (plus appropriate context such as the call path) of S' match those seen in \hat{S} , giving rise to a narrower notion of equivalence that we denote by $\hat{S} \approx_L S'$. Deciding $\hat{S} \approx_L S'$ uses a live variable analysis at each program point. We do not have a live variable analysis that is sound in the presence of interrupt paths, which are prevalent in our domain. We expect that such an analysis would be costly and less accurate when accounting for interrupts, and so we go a different route. FIE checks equivalence by investigating if every variable, symbolic expression², program counter, and all other parts of the state are equal between \hat{S} and S' , denoted $\hat{S} \approx_A S'$. This embodies a trade-off between simplicity of equivalence checking (i.e., we forgo static analysis) and the ability to prune as aggressively as is theoretically possible.

Lines 14–20 of Figure 4 realize state pruning. There a function `Prune` checks each potential successor $S' \in \mathcal{SS}$ to see if it is equivalent to any of the previously generated states in $\mathcal{PS}_{p'}$, namely those that have the same program counter p' as S' . To use the \approx_A equivalence relation efficiently, we modify the way KLEE maintains states in memory, storing for each visited program counter a set of diffs of the memory contents of all states that have been seen at that program counter. This also allows fast comparisons to detect redundant states.

4.4 An Optimization: Memory Smudging

As we will see in the next section, FIE as described thus far can already be used to perform complete analyses for some simple firmware and achieves good code-coverage for some more complex firmware. However, it is clear that even small programs can force FIE to attempt to explore an intractable number of states. For example, consider the code snippet in Figure 6(c). The empty for loop on line 4 will force FIE to proceed down at least one path of length at least `MAX_LONG` instructions. Unlike the loop example in Figure 6(b), state pruning cannot short-circuit evaluation of this long path because the value of

²We only consider syntactic equality of constraints, and do not attempt to decide if two different sets of constraints define the same set of possible values.

```

1 int main(){
2     eint();
3     BCSCTL1 = CALBC1_1MHZ;
4     DCOCTL = CALDCO_1MHZ;
5 }
6 ISR(PORT1){
7     a += 1;
8 }

```

(a) Code with equivalent paths

```

1 uint8_t getByte(){
2     ...
3     while(P1IN & BIT2);
4     if (P1IN & BIT2){
5         goto WaitForStart;
6     }
7     ...
8 }

```

(b) Code with an infinite fork

```

1 int main(){
2     ...
3     long i = 0;
4     while(i < MAX_LONG) {
5         i++;
6     }
7     ...
8 }

```

(c) Code with a long loop

Figure 6: State pruning can detect and remove the redundant states produced in code samples (a) and (b). Memory smudging replaces i in code sample (c) with a symbolic variable after t iterations (e.g., $t = 100$), enabling analysis to move beyond the loop more quickly.

i is monotonically increasing and so states never repeat along the path.

To speed analysis for such settings, while retaining the ability to be complete, we use *memory smudging*. It is represented by the function `MemorySmudge` on line 19 of Figure 4. At analysis time, the analyst supplies a modification threshold t to FIE. Before adding a (non-redundant) successor state S' to \mathcal{AS} , the function `MemorySmudge` checks if any memory locations in S' have been modified t times. If so the location’s value is replaced by a special value \star . This wildcard value may take any value allowed by the type and cannot be constrained. To implement this, FIE keeps a count of every distinct value that an instance of a variable takes on at a program point. The count is associated with the activation record of the variable. Thus, if a local variable is smudged, it will again be concrete on the next call to that function while global variables remain smudged.

Smudging allows the analysis to explore more of a firmware at the cost of precision. To see this, consider again Figure 6(c), and let the smudging threshold be $t = 100$. On iteration 100, i takes on the value \star . Then, as the loop continues to iteration 101, the condition $i < \text{MAX_LONG}$ will cause the execution state to be split into two states: a state S_F that fails the loop condition and proceeds to line 7, and a state S_T that executes the body of the loop at line 5 again. By executing S_T , code that would not be executed until MAX_LONG iterations of the loop can instead be executed after 100 iterations. This approach does lead to the addition of new states (compared to execution without smudging), but we have found that pruning typically eliminates states added due to smudging. When FIE executes S_F in the example above, it will explore the (empty) body of the loop, and i will be incremented. However, since $i = \star$, the update to i will be discarded. Now, S_F is again at the head of the loop, and execution state is identical to the previous iteration: no variable besides i has been touched, and $i = \star$ as it did on the previous iteration. Thus, S_F ends up pruned.

Memory smudging over-approximates a state and so can be a source of false positives, i.e., \star contains values that may never be realized along any path. For example,

a pointer modified t times and then dereferenced can result in a false positive. FIE reports in output warnings if any involved values were smudged, making it easier for analyst to detect such a false positive. As we see in the next section, false positives due to smudging seem rare in practice for reasonable values of t .

4.5 Implementation Details

The pseudocode presented in Figure 4 gives the high-level logic of FIE but abstracts away many details for simplicity. Our implementation includes a number of important embellishments, which we can only briefly describe here.

Memory sharing: Since FIE creates at least one new state at nearly every instruction, it is important that the creation and storage of states be as efficient as possible. Thus, we only store one complete state per calling context for each \mathcal{PS} . Additional states with the same calling context are then compared to the existing state, and only the incremental difference in that state are stored. We also inherit memory optimizations from KLEE, the most important of which is copy-on-write memory for states.

Pruning frequency: The PRUNE operation shown on line 18 of Figure 4 can become expensive as the number of states at \mathcal{PS} becomes large. Rather than performing this operation at each instruction, the default mode of FIE prunes only at basic block boundaries. We preserve the ability to configure FIE to prune at each instruction, but have found that basic-block-level pruning improves performance in all our tests.

5 Evaluation

To evaluate FIE, we used it to analyze the 99 firmware programs in our corpus. We will investigate the overall efficacy in terms of code coverage, the ability to complete analyses, the utility of pruning and smudging, and the bugs FIE helped us find.

Firmware size and coverage: We first fix some conventions regarding how we measure the size of firmware programs and analysis coverage. For our evaluations, we measure firmware size by the *number of executable LLVM instructions*, denoted by the acronym NEXI. We

compute a firmware’s NEXI by: (1) compiling the firmware into LLVM bitcode using CLANG; (2) running the resulting bitcode through LLVM optimization passes for global and local dead code elimination; and (3) taking the number of LLVM instructions in the resulting bitcode as the NEXI. This count includes intrinsic functions and library functions called by the firmware. We note that some programs used external modules whose source code was not included in their source tree; we did not attempt to track down these libraries and FIE emits an error should it execute an instruction calling an omitted function. Likewise for inline assembly instructions not supported by FIE. This did not significantly affect our evaluation, e.g., only two programs ever reached missing functions or inline assembly in the 50 minute runs reported on below. Note that usually FIE continues running in such cases along other paths.

Code coverage is the fraction of LLVM instructions executed in the course of the analysis divided by the NEXI of the target firmware. Using NEXI as opposed to C lines of code better aligns our complexity and coverage metrics with the work done by FIE, and avoids any ambiguity in terms of overcounting coverage of partially executed blocks or lines of C code. The NEXI sizes are, on average, 1.5 times larger than the number of lines of C code computed in Section 2. NEXI was smaller than cloc for 23 of the programs due to dead code elimination.

Experimental setup: All the analyses reported on below used Amazon EC2 high-memory, double-extra-large (m2.2xlarge) instances which have 36 GB of RAM and 13 virtual CPUs (each advertised to be the equivalent of an 1.0–1.2 GHz 2007 Opteron or 2007 Xeon processor). Unless specified otherwise, FIE was given 50 minutes of runtime³, and each analysis was performed on a separate EC2 instance. To facilitate this effort, we wrote a set of scripts for launching, monitoring, and retrieving the results of FIE run via a custom EC2 VM image.

Coverage under different FIE modes: We started by analyzing each firmware for 50 minutes for each of five different modes supported by FIE, for a total of 495 executions. The resulting NEXI coverages are shown in Table 7. We now explain the modes and discuss their performance.

Baseline: The BASELINE mode reflects a bare minimum port of KLEE to the MSP430 environment, in particular it has support for: 16-bit addressing; a custom memory allocator that ensures that memory objects do not collide with special memory locations and have addresses within the chip’s address range; and implementation of intrinsics supported by msp430-gcc. It does not, however, have any knowledge of the semantics of, spe-

³Setting the time to a bit less than one hour halves the cost of running on EC2.

% NEXI	BASELINE	FUZZ	PLAIN	PRUNE	SMUDGE
Complete	n/a	n/a	7	35	52
100%	1	43	40	34	46
90–100%	1	10	9	15	15
80–90%	0	7	5	10	6
70–80%	0	5	5	6	4
60–70%	0	4	5	5	5
50–60%	0	4	6	5	3
40–50%	0	8	8	9	5
30–40%	0	0	0	2	3
20–30%	1	8	11	4	5
10–20%	10	5	5	3	3
0–10%	86	5	5	6	4
Total %	1.1	26.1	23.7	29.5	32.3
Avg. %	5.9	74.5	71.1	74.4	79.4
Median %	1.7	96.9	89.5	88.7	98.1

Table 7: Number of firmware programs for which FIE achieves coverage in the indicated range, for 50 minute runs of FIE in each of five operating modes. “Complete” gives the number of programs for which the mode was able to analyze all possible symbolic states.

cial memory or interrupts, etc. For most firmware, the BASELINE analysis performs very poorly, with a median of 1.7% coverage. This is because BASELINE almost always ends prematurely with a false positive since the firmware appears (to the analysis) as if it were reading from an uninitialized memory location. Manual inspection of the code of the two outliers (from GitHub) revealed that they are not using any features of the MSP430 architecture. The poor coverage of BASELINE for the other firmware programs attests to the importance of providing an architecture-aware analysis.

Fuzz: We next use FIE to realize a general-purpose fuzzing tool for MSP430 firmware. This mode, unlike BASELINE, takes advantage of knowledge of the memory layout, special registers, and interrupt handling semantics. We implemented a special memory spec in which any read to a peripheral results in a returned value chosen uniformly in the appropriate range. (Twice reading the same peripheral location leads to two independent values.) Writes to peripherals are ignored. We use the conservative interrupt spec, meaning that in the FUZZ mode the analysis branches off new states to execute interrupt handlers as appropriate. In this mode, then, FIE never generates symbolic variables, and so is able to quickly evaluate on concrete values along many paths. Fuzzing provides surprisingly good coverage for many of the firmware programs, in fact beating symbolic execution modes in many cases. This is because fuzzing can evaluate states more quickly, and for simple programs this can lead to good coverage in a 50 minute test.

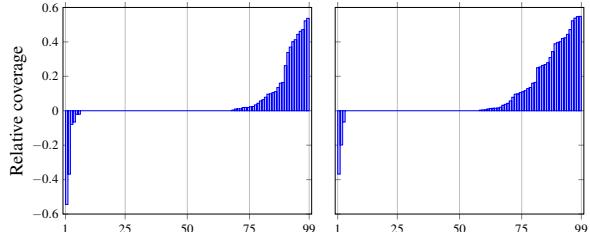


Figure 8: Coverage of SMUDGE relative to FUZZ (top) and SMUDGE relative to PLAIN (bottom) for the 50-minute tests. Here SMUDGE outperforms FUZZ and PLAIN for 32% and 42% of the programs, respectively.

Plain, Prune, and Smudge: We now turn to modes that use FIE as a symbolic executor with the architecture-aware analysis. To compare the efficacy of the state pruning and memory smudging techniques, we use three different modes: PLAIN (no pruning or smudging), PRUNE (with pruning but not smudging), and SMUDGE (pruning and smudging, with smudging threshold $t = 100$). All three modes used the most conservative interrupt model. Overall SMUDGE provides better coverage than all others, including FUZZ. A comparison of relative performance for each firmware appears in the charts of Figure 8. The x-axes is the firmware (ordered by y-values) and the y-value is $(N_s - N_f)/N_{tot}$ where N_s is, for the left chart, the number of instructions covered by SMUDGE for this firmware, N_f is the number covered by FUZZ, and N_{tot} is the NEXI for the firmware. The right chart is the same except comparing SMUDGE with PLAIN. These graphs surface two facts. First, there exists a large number of firmware programs for which the analyses do equally well (where relative coverage is 0, most often because both analyses had 100% coverage), which is due to the large number of very simple firmware from GitHub and the TI website. Second, SMUDGE can do worse than others on a few firmware programs, but improves performance over FUZZ for 32% of the programs and over PLAIN for 42% of the programs.

50-minute analysis outcomes: In Table 9 we give a breakdown of the emitted termination status for the analyses. FIE can either stop because it runs out of memory (No mem), the requested amount of execution time has been reached (Timeout), or because there exist no more active states (Finished). Additionally, FIE will output bug reports. As can be seen, pruning and smudging help reduce memory usage and increase the number of analyses that finish. Potential bugs were reported for 92 firmware programs by the BASELINE, all false positives. Smudging introduced a false positive in one firmware, since a pointer was smudged. (Smudging a pointer frequently leads to a memory safety violation, because any dereference of it will be viewed as an error.) FIE makes it easy to determine if a warning is related to smudging

Mode	Termination status			FPs
	No mem	Timeout	Finished	
BASELINE	9	2	88	93
FUZZ	10	79	10	0
PLAIN	7	85	7	0
PRUNE	0	64	35	0
SMUDGE	0	46	53	1

Table 9: Counts of each termination code seen in the 50-minute runs. “No mem”: the analysis exhausted memory; “Timeout”: analysis ran for the full 50 minutes; “Finished”: analysis completed early. The final column is the number of firmware programs with erroneous bug reports.

by marking variables that were smudged as such in the bug report. No true positives were found in these short runs.

Recall that an explicit design goal for FIE was the ability to support complete analyses (all possible symbolic states are checked). The PLAIN, PRUNE, and SMUDGE modes do support this: an analysis is *complete* if the termination status was Finished and no bugs were reported. Modulo the limitations discussed in the next section, this verifies the absence of bugs. The first row of Table 7 shows the number of firmware programs for which PLAIN, PRUNE, or SMUDGE were able to verify the absence of memory safety and (some kinds of) peripheral misuse bugs, in these short 50 minute runs. As can be seen, our pruning and smudging mechanisms enable a huge increase in the number of analyses that FIE can complete: a 6x increase when we use pruning and an additional factor of 1.48x improvement when we add in smudging. In the end, the total number of firmware programs for which one of the analysis modes completed is 53. (One firmware was completed by PRUNE but had a false positive under SMUDGE, and so it was not counted in the SMUDGE column of Table 7.)

We note that these complete analyses revealed that 13 firmware programs have dead code missed by the static optimization passes, which means for these our measured coverage is lower than it should be (e.g., for one firmware we achieved a complete analysis but only 45% coverage). For consistency, we do not correct the NEXI values for these firmware programs.

Firmware complexity measures: The above shows that FIE enables complete analyses for a majority of the firmware programs in our corpus, yet the total amount of code covered across the full corpus indicates that FIE was not able to explore most of the larger firmware programs given only 50 minutes. As can be seen in Figure 10, the coverage is uniformly poor for firmware programs with more than 4,000 executable instructions. More subtly, there exist many much smaller programs for which coverage is also poor (the vertical trend closer to the y-axis), which could be due to complicated but short code con-

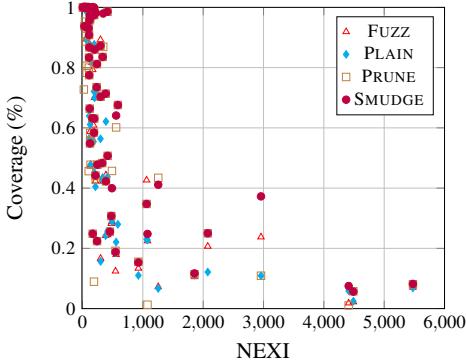


Figure 10: Coverage as a function of firmware size in the 50 minute tests.

Complexity	Criteria	# FWs	SMUDGE coverage
low	≤ 100 NEXI or < 2 loops	49	Avg: 93.6% Med: 100%
medium	≤ 500 NEXI and ≥ 2 loops	37	Avg: 79.5% Med: 93.1%
high	> 500 NEXI and ≥ 2 loops	13	Avg: 27.8% Med: 24.8%

Table 11: Criteria for firmware complexity groups, the number of firmware programs (# FWs) in each group, and SMUDGE’s average and median coverage for each group.

structs or, perhaps, undiscovered dead code.

To focus subsequent experiments on the more challenging firmware programs, we partition the programs into three complexity groups based on a simple static analysis. (Using static analysis avoids biasing the set unnecessarily by the nature of FIE’s analysis.) The criteria for partitioning our programs into low-, medium-, and high-complexity groups is described in Table 11. To determine the number of loops in a program, we use LLVM’s built-in loop detection. We chose this particular partitioning for its simplicity, but admit there are many other possibilities. We give the average and median performance of the 50-minute SMUDGE runs as broken down by each group. Of the 53 programs that FIE is able to complete analysis for in 50 minutes, 38 are low complexity and 14 are medium complexity. The average NEXI for these completed programs is 84.4 and the average number of loops is 2.2; the most complex completed program has a NEXI of 414 and 17 loops.

Effect of smudging threshold: We now measure the effect of the smudging threshold t on coverage and false positive rates for the high complexity firmware programs. By decreasing t one might hope to achieve a trade-off between coverage improvements (by breaking out of loops even more quickly) and increased risk in false positives. We run SMUDGE for 50-minutes for each of $t = 1, 10, 1000$ for the 13 firmware programs and

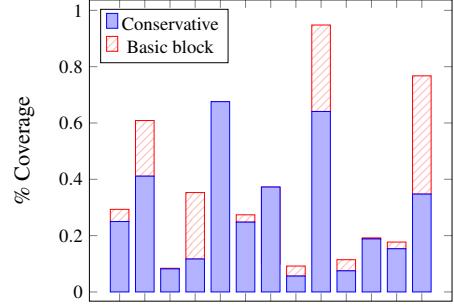


Figure 12: Coverage when spawning interrupts every instruction (Conservative) versus once per basic block for the 13 high-complexity firmware programs.

use as well the $t = 100$ results from the runs discussed above. The average coverages were 23.3%, 25.2%, 25.5%, 25.6% for $t = 1, 10, 100, 1000$, making the differences too small to be significant. The number of false positives increased for small t ; with $t = 1$ there were two false positives, and none for the larger values of t . We conclude that $t = 100$ strikes a reasonable balance, but further performance improvements may not be easily obtained by tweaking t .

Relaxing the interrupt model: Recall that we have so far been using FIE with a very conservative interrupt model in which all enabled interrupts fire at every program point. This can mean that most instructions, as opposed to just branches, end up forking off multiple new states. We therefore implement a relaxed interrupt model in which every enabled interrupt fires at the first instruction of each basic block, but not during subsequent instructions. This means analysis will miss possible paths (barring complete analyses) but could speed up performance and thereby increase code coverage. In Figure 12 we compare, for the high-complexity firmware programs, the coverage obtained by SMUDGE with $t = 100$ using both the conservative interrupt model (Conservative) and the new model that only fires at each basic block (Basic block). The results are both from 50 minute runs. Several of the firmware programs see drastic coverage improvements, the last bar on the right represents the largest improvement at 232%. No false positives arose in these basic block runs, however one program hit a code construct⁴ currently not supported by FIE.

Finding vulnerabilities: FIE currently supports finding two types of bugs: memory safety violations, such as buffer overruns and out-of-bounds accesses to memory objects like arrays, as well as peripheral-misuse errors in which a firmware writes to a read-only memory location or to locked flash. It will be easy to increase scope to further security properties in the future.

⁴A firmware used a custom variable argument function. We plan to add support in the public release version.

Firmware	NEXI	Types	# bugs
CDC Driver	4,489	Memory safety	10
HID Driver	2,958	Memory safety	11
controleasy	1,255	Flash misuse	1

Table 13: Summary of vulnerabilities discovered by FIE. The final column is the number of distinct vulnerabilities in the firmware.

We supplemented the above analyses with runs in which we allowed SMUDGE to run up to 24 hours, with $t = 100$ and using the conservative interrupt model, on each of the 13 high-complexity programs. Table 13 gives a breakdown of the 22 bugs found across all of the runs. The bugs were spread across three firmware programs, the two USB drivers supplied by TI and one community firmware called Controleasy. Of the bugs, 21 were memory safety violations while one was a flash misuse bug. CDC Driver and HID Driver share some common source files, one of the bugs spans both, while the others are from disjoint source files. The memory safety bugs in the two USB drivers include 18 vulnerabilities in which a USB protocol value (received from off-chip) ends up controlling an index into an array, allowing adversarial reads or the ability to crash the firmware. One of the vulnerabilities, in HID Driver, allows an adversary over the network to inject arbitrarily long strings due to an unprotected `strcat`. This allows crashing the firmware, but may also lead to a complete compromise by way of control flow hijacking. The final two memory safety bugs are present in both programs but arise from the same source file. The bug dereferences a value read from flash, which in our model is untrusted but unlikely to be exploitable in most settings.

The TI community code project controleasy has a peripheral misuse bug in which a read-only I/O port can be written to based on the value of another peripheral. Like the attacker-controlled reads in the USB code, this bug can be used by an attacker that can send data to PORT 1 to cause the firmware to crash.

6 Limitations

The evaluation in the last section evidenced FIE’s effectiveness at both finding bugs as well as verifying their absence. Of course, FIE does have some limitations.

The design of FIE arises from a philosophy that sound and complete analysis are valuable and can be feasible for the embedded firmware often found in practice. However, it is simple to show that there exist firmware for which complete analyses are intractable, and likewise soundness is only with respect to the symbolic execution framework (it is possible that reported bugs may not arise in the firmware when run natively, as discussed below). Indeed some of the firmware in our corpus (e.g., Contiki) appear to have, in particular, an intractably large

number of reachable states. Here FIE attempts to provide as high as possible code coverage, but improving on the results reported in the last section might require different techniques than currently used. For example, a combination of loop elision [14, 26] and improved state selection heuristics might be more effective than state pruning and memory smudging. Future work might therefore explore incorporation of other techniques into FIE.

Both when achieving complete analyses and when not, there exist various sources of imprecision in analysis that may lead to false positives or false negatives. In developing FIE we often encountered analysis errors due to bugs in the analysis software or misconfiguration (e.g., using the incorrect target architecture almost always yields false positives). These problems were subsequently fixed, and while we are unaware of any outstanding bugs in FIE and have manually verified all the bugs reported in Section 5, it could be that some analysis errors remain.

Imprecision can also arise due to discrepancies between the firmware as symbolically executed in FIE and natively in deployment. In building FIE, we had to implement extensions to C that are (sometimes implicitly) defined by msp430-gcc. We encountered inconsistencies between msp430-gcc and FIE, which were subsequently fixed, but some may remain. These C extensions also differ among the three MSP430 compilers, and so analyzing firmware written to work for the IAR or CCS compilers (e.g., the USB drivers in our corpus) may give rise to analysis errors when using FIE. Even so FIE can still be useful for finding vulnerabilities in such firmware programs, as the bugs found in the USB drivers shows.

As a final source of imprecision, our most conservative analysis models peripherals and interrupt firing as adversarially controlled. This means that FIE may explore states that do not arise in real executions, and errors flagged due to such states would constitute false positives. We feel that fixing even such bugs should be encouraged, since it reduces the potential for latent vulnerabilities. Moreover, it is unclear where to draw the line in terms of adversarial access to a chip. That said, FIE is easily customizable should such false positives prove burdensome, or to receive the speed-ups of other environmental models.

Finally, we note that currently FIE fails execution paths that include inline assembly. While we added some explicit handlers for several inline assembly instructions (e.g., `nop`), this approach would struggle with complex assembly code constructs. Future work might investigate performing symbolic analysis starting with MSP430 assembly, similarly to [7].

7 Related Work

FIE is based off of KLEE and, in turn, builds off the work of KLEE’s predecessors such as EXE [11]. These prior systems target generation of high-coverage test suites for non-embedded programs (e.g., Linux CoreUtils). As we saw in previous sections, using KLEE with a minimal amount of porting provides poor coverage. The many systems that extend KLEE [2,9,13,28,31,36] do not target embedded systems, with the exception of KleeNet [28]. It targets wireless sensor nodes running Contiki [35], but only on x86 platforms, and so does not work for our setting of MSP430 firmware programs.

Concolic execution systems extend symbolic execution by concretizing values that cannot be handled by the constraint solver efficiently (or cannot be handled by the constraint solver at all) [14, 30]. Whole-system concolic execution tools like S2E [14] can execute external functions natively by concretizing symbolic arguments, and then providing the concrete value in the call. Their model of concretization makes less sense in our setting, where we have a firmware that specifies all software on the system and interacts only with hardware peripherals. For the latter, we can support concretization in the sense that a memory specification can return concrete values, change symbolic values to concrete, etc.

SymDrive [26] builds off S2E to test Linux and FreeBSD kernel drivers without the need for the actual hardware, and treats many of the same problems as FIE, including modeling hardware, dealing with polling loops, etc. SymDrive uses static analysis to help guide execution along states that reach deep paths and to avoid loops. This improves code coverage, but does not enable complete analyses. We leave incorporating such static analysis techniques into FIE, in order to increase code coverage in conjunction with state pruning and memory smudging, for future work.

Pruning redundant states during an analysis has been considered before in a variety of program analysis contexts [3, 5, 32]. Closest to our work is RWset [6], which extended the EXE [11] symbolic execution engine to track live variables and to discard a state should the values of all live variables have already been executed upon. Our state pruning approach is simpler and does not require an auxiliary live variable analysis (which can be challenging in the face of interrupt-driven code). The trade-off for this simplicity is that FIE may prune less aggressively than possible. On the other hand, FIE goes further than RWset in limiting path explosion via memory smudging, which is effective even when, for example, variables written within a loop are live.

There is a body of work on improving the performance of symbolic execution by merging similar states [23, 25]. State merging seeks to combine identical (or similar) active states, whereas state pruning compares active states

to both active and prior states. Only the latter enables complete analysis. Whether the two techniques are useful in conjunction is an interesting open question.

Much effort has gone into improving the scalability of symbolic execution [7, 15, 29]. One such example is Cloud9, which speeds symbolic execution by parallelizing the execution of multiple memory states across a cluster of commodity hardware. We note that such techniques are applicable to FIE, and future work may involve adopting such techniques to improve the performance of FIE for large firmware programs.

Finally, we are aware of two commercial tools of potential relevance to FIE. The first, Codenomicon [16], offers a network protocol fuzzing tool for embedded medical devices. It therefore targets protocol parsing logic, which is a frequent source of vulnerabilities. FIE already supports rudimentary fuzzing, and could perform network protocol fuzzing (or a mixture of fuzzing and symbolic execution) by implementing more detailed memory specs. Second is Coverity [4], a static analysis tool that targets a number of platforms, including the MSP430. While we have access to Coverity, their software license unfortunately prevents head-to-head comparisons in published research.

8 Conclusion

In this paper, we presented the design and implementation of FIE, a tool for performing symbolic-execution-based analysis of MSP430 firmware programs. It provides an extensible platform for finding security vulnerabilities and other kinds of bugs, and has proven effective in analyzing a large corpus of open-source MSP430 firmware programs. To increase code coverage in a way that supports verification of security properties, we incorporate into FIE the techniques of state pruning and memory smudging. We used FIE to verify memory safety for 53 firmware programs and elsewhere found 21 distinct vulnerabilities, some of which appear to be remotely exploitable. All this shows that FIE is particularly well-suited to the small, simple firmware programs often used for microcontrollers and proves useful for analysis of more complex firmware programs as well.

Acknowledgements

We would like to thank Kevin Fu, Matt Renzelmann and the anonymous reviewers for their extensive feedback on earlier drafts of this paper. This work was supported, in part, by DARPA and AFRL under contracts FA8650-10-C-7088 and CNS-1064944. The views, opinions, and/or findings contained herein are those of the authors and should not be interpreted as representing the official views or policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the Department of Defense.

References

- [1] Amazon. Amazon elastic compute cloud. <http://aws.amazon.com/ec2>, 2013. Last accessed Jun 2013.
- [2] T. Avgerinos, S. K. Cha, B. L. T. Hao, and D. Brumley. AEG: Automatic exploit generation. In *Network and Distributed System Security Symposium (NDSS)*, 2011.
- [3] T. Ball, V. Levin, and S. K. Rajamani. A decade of software model checking with SLAM. *Commun. ACM*, 54(7):68–76, July 2011.
- [4] A. Bessey, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C. Henri-Gros, A. Kamsky, S. McPeak, and D. Engler. A few billion lines of code later: using static analysis to find bugs in the real world. *Commun. ACM*, 53(2):66–75, Feb. 2010.
- [5] D. Beyer, T. A. Henzinger, R. Jhala, and R. Majumdar. The software model checker Blast: Applications to software engineering. *Int. J. Softw. Tools Technol. Transf.*, 9(5):505–525, Oct. 2007.
- [6] P. Boonstoppel, C. Cedar, and D. Engler. RWset: Attacking path explosion in constraint-based test generation. In C. Ramakrishnan and J. Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 4963 of *Lecture Notes in Computer Science*, pages 351–366. Springer Berlin Heidelberg, 2008.
- [7] D. Brumley, C. Hartwig, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam, D. Song, and H. Yin. BitScope: Automatically dissecting malicious binaries. Technical report, In CMU-CS-07-133, 2007.
- [8] D. Brumley, J. Newsome, D. Song, H. Wang, and S. Jha. Towards automatic generation of vulnerability-based signatures. In *Proceedings of the 2006 IEEE Symposium on Security and Privacy (SP)*, pages 2–16. IEEE Computer Society, 2006.
- [9] S. Bucur, V. Ureche, C. Zamfir, and G. Cadea. Parallel symbolic execution for automated real-world software testing. In *EuroSys*, pages 183–198, 2011.
- [10] C. Cedar, 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)*, pages 209–224. USENIX Association, 2008.
- [11] C. Cedar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. EXE: automatically generating inputs of death. In *ACM Conference on Computer and Communications security*, pages 322–335. ACM, 2006.
- [12] S. Checkoway, D. McCoy, B. Kantor, D. Anderson, H. Shacham, S. Savage, K. Koscher, A. Czeskis, F. Roesner, and T. Kohno. Comprehensive experimental analyses of automotive attack surfaces. In *Proceedings of USENIX Security*, 2011.
- [13] V. Chipounov and G. Cadea. Reverse engineering of binary device drivers with RevNIC. In *EuroSys*, pages 167–180, 2010.
- [14] V. Chipounov, V. Kuznetsov, and G. Cadea. S2E: a platform for in-vivo multi-path analysis of software systems. *SIGPLAN Not.*, 46(3):265–278, Mar. 2011.
- [15] L. Ciortea, C. Zamfir, S. Bucur, V. Chipounov, and G. Cadea. Cloud9: a software testing service. *SIGOPS Oper. Syst. Rev.*, 43(4):5–10, Jan. 2010.
- [16] Codenomicon. Codenomicon defensics. <http://www.codenomicon.com>, 2013. Last accessed Jun 2013.
- [17] M. Costa, M. Castro, L. Zhou, L. Zhang, and M. Peinado. Bouncer: securing software by blocking bad input. In *Symposium on Operating System Principles (SOSP)*, pages 117–130, 2007.
- [18] A. Cui, M. Costello, and S. J. Stolfo. When firmware modifications attack: A case study of embedded exploitation. In *Network and Distributed System Security Symposium (NDSS)*, 2013.
- [19] A. Cui and S. J. Stolfo. A quantitative analysis of the insecurity of embedded network devices: results of a wide-area scan. In *Annual Computer Security Applications Conference (ACSAC)*, pages 97–106. ACM, 2010.
- [20] W. Frisby, B. Moench, B. Recht, and T. Ristenpart. Security analysis of smartphone point-of-sale systems. In *Proceedings of the 6th USENIX conference on Offensive Technologies (WOOT)*, pages 3–3, 2012.
- [21] D. Halperin, T. Heydt-Benjamin, B. Ransford, S. Clark, B. Deffenbacher, W. Morgan, K. Fu, T. Kohno, and W. Maisel. Pacemakers and implantable cardiac defibrillators: Software radio attacks and zero-power defenses. In *IEEE Symposium on Security and Privacy (SP)*, pages 129–142, 2008.
- [22] D. Halperin, T. Kohno, T. Heydt-Benjamin, K. Fu, and W. Maisel. Security and privacy for implantable medical devices. *Pervasive Computing, IEEE*, 7(1):30–39, 2008.
- [23] T. Hansen, P. Schachte, and H. Søndergaard. State joining and splitting for the symbolic execution of binaries. In *Runtime Verification, 9th International Workshop, RV 2009*, pages 76–92, 2009.
- [24] K. Koscher, A. Czeskis, F. Roesner, S. Patel, T. Kohno, S. Checkoway, D. McCoy, B. Kantor, D. Anderson, H. Shacham, et al. Experimental security analysis of a modern automobile. In *2010 IEEE Symposium on Security and Privacy*, pages 447–462. IEEE, 2010.
- [25] V. Kuznetsov, J. Kinder, S. Bucur, and G. Cadea. Efficient state merging in symbolic execution. In *PLDI*, pages 193–204, 2012.
- [26] M. J. Renzelmann, A. Kadav, and M. M. Swift. Symdrive: testing drivers without devices. In *Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation*, pages 279–292. USENIX Association, 2012.
- [27] I. Rouf, R. Miller, H. Mustafa, T. Taylor, S. Oh, W. Xu, M. Gruteser, W. Trappe, and I. Seskar. Security and privacy vulnerabilities of in-car wireless networks: a tire pressure monitoring system case study. In *Proceedings of the 19th USENIX conference on Security*, 2010.
- [28] R. Sasnauskas, O. Landsiedel, M. H. Alizai, C. Weise, S. Kowalewski, and K. Wehrle. Kleenet: Discovering insidious interaction bugs in wireless sensor networks before deployment. In *ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN)*, Stockholm, Sweden, April 2010.
- [29] P. Saxena, P. Poosankam, S. McCamant, and D. Song. Loop-extended symbolic execution on binary programs. In *International Symposium in Software Testing and Analysis (ISSTA)*, pages 225–236, 2009.
- [30] K. Sen, D. Marinov, and G. Agha. CUTET: A concolic unit testing engine for C. In *Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering, ESEC/FSE-13*, pages 263–272, New York, NY, USA, 2005. ACM.
- [31] A. Slowinska, T. Stancescu, and H. Bos. Howard: A dynamic excavator for reverse engineering data structures. In *Network and Distributed System Security Symposium (NDSS)*, 2011.
- [32] U. Stern and D. L. Dill. Improved probabilistic verification by hash compaction. In *In Advanced Research Working Conference on Correct Hardware Design and Verification Methods*, pages 206–224. Springer-Verlag, 1995.
- [33] Texas Instruments. Microcontroller projects website. <http://e2e.ti.com/group/microcontrollerprojects/msp430microcontrollerprojects/default.aspx>. Last accessed Jun 2013.
- [34] Texas Instruments. MSP430 for security applications. <http://www.ti.com/mcu/docs/mcuorphan.tsp?contentId=33485&DCMP=MSP430&HQ=0ther+0T+430security>, January 2012.
- [35] The Contiki Project. Contiki. <http://www.contiki-os.org/>. Last accessed Jun 2013.
- [36] C. Zamfir and G. Cadea. Execution synthesis: a technique for automated software debugging. In *EuroSys*, pages 321–334, 2010.

Sancus: Low-cost trustworthy extensible networked devices with a zero-software Trusted Computing Base

Job Noorman Pieter Agten Wilfried Daniels Raoul Strackx
Anthony Van Herrewege Christophe Huygens Bart Preneel Ingrid Verbauwhede
 Frank Piessens

iMinds-DistriNet and iMinds-COSIC, KU Leuven
{Job.Noorman, Pieter.Agtent, Wilfried.Daniels, Raoul.Strackx,
Christophe.Huygens, Frank.Piessens}@cs.kuleuven.be
{Anthony.VanHerrewege, Bart.Preneel, Ingrid.Verbauwhede}@esat.kuleuven.be

Abstract

In this paper we propose Sancus, a security architecture for networked embedded devices. Sancus supports extensibility in the form of remote (even third-party) software installation on devices while maintaining strong security guarantees. More specifically, Sancus can remotely attest to a software provider that a specific software module is running uncompromised, and can authenticate messages from software modules to software providers. Software modules can securely maintain local state, and can securely interact with other software modules that they choose to trust. The most distinguishing feature of Sancus is that it achieves these security guarantees without trusting *any* infrastructural software on the device. The Trusted Computing Base (TCB) on the device is *only* the hardware. Moreover, the hardware cost of Sancus is low.

We describe the design of Sancus, and develop and evaluate a prototype FPGA implementation of a Sancus-enabled device. The prototype extends an MSP430 processor with hardware support for the memory access control and cryptographic functionality required to run Sancus. We also develop a C compiler that targets our device and that can compile standard C modules to Sancus protected software modules.

1 Introduction

Computing devices and software are omnipresent in our society, and society increasingly relies on the correct and secure functioning of these devices and software. Two important trends can be observed. First, network connectivity of devices keeps increasing. More and more (and smaller and smaller) devices get connected to the Internet or local ad-hoc networks. Second, more and more devices support extensibility of the software they run – often even by third parties different from the device manufacturer or device owner. These two factors are important because they enable a vast array of interesting

applications, ranging from over-the-air updates on smart cards, over updateable implanted medical devices to programmable sensor networks. However, these two factors also have a significant impact on security threats. The combination of connectivity and software extensibility leads to malware threats. Researchers have already shown how to perform code injection attacks against embedded devices to build self-propagating worms [18, 19]. Viega and Thompson [45] describe several recent incidents and summarize the state of embedded device security as “a mess”.

For high-end devices, such as servers or desktops, the problems of dealing with connectivity and software extensibility are relatively well-understood, and there is a rich body of knowledge built up from decades of research; we provide a brief survey in the related work section.

However, for low-end, resource-constrained devices, no effective low-cost solutions are known. Many embedded platforms lack the standard security features (such as privilege levels or advanced memory management units that support virtual memory) present in high-end processors. Depending on the overall system security goals, as well as the context in which the system must operate, there may be more optimal solutions than just porting the general-purpose security features from high-end processors. Several recent results show that researchers are exploring this idea in a variety of settings. For instance, El Defrawy et al. propose SMART, a simple and efficient hardware-software primitive to establish a dynamic root of trust in an embedded processor [14], and Strackx et al. propose a simple program-counter based memory access control system to isolate software components [43].

In this paper we build on these primitives to propose a security architecture that supports secure third-party software extensibility for a network of low-end processors (the prototypical example of such a network is a sensor network). The architecture enables mutually distrusting parties to run their software modules on the same nodes in the network, while each party maintains strong assurance

that its modules run untampered. This kind of secure software extensibility is very useful for applications of sensor networks, for instance in the logistics and medical domains. We discuss some application areas in more detail in Section 2.4.

The main distinguishing feature of our approach is that we achieve these security guarantees without *any* software in the TCB on the device, and with only minimal hardware extensions. Our attacker model assumes that an attacker has *complete* control over the software state of a device, and even for such attackers our security architecture ensures that any results a party receives from one of its modules can be validated to be genuine. Obviously, with such a strong attacker model, we can not guarantee availability, so an attacker can bring the system down, but if results are received their integrity and authenticity can be verified.

More specifically, we make the following contributions:

- We propose Sancus¹, a security architecture for resource-constrained, extensible networked embedded systems, that can provide remote attestation and strong integrity and authenticity guarantees with a minimal (hardware) TCB.
- We implement the hardware required for Sancus as an extension of a mainstream microprocessor, and we show that the cost of these hardware changes (in terms of performance, area and power) is small.
- We implement a C compiler that targets Sancus-enabled devices. Building software modules for Sancus can be done by putting some simple annotations on standard C files, showing that the cost in terms of software development is also low.

To guarantee the reproducibility and verifiability of our results, all our research materials, including the hardware design of the processor, and the C compiler are publicly available.

The remainder of this paper is structured as follows. First, in Section 2 we clarify the problem we address by defining our system model, attacker model and the security properties we aim for. The next two sections detail the design of Sancus and some interesting implementation aspects. Section 5 reports on our evaluation of Sancus and the final two sections discuss related work and conclude.

2 Problem statement

2.1 System model

We consider a setting where a single infrastructure provider, IP , owns and administers a (potentially large)

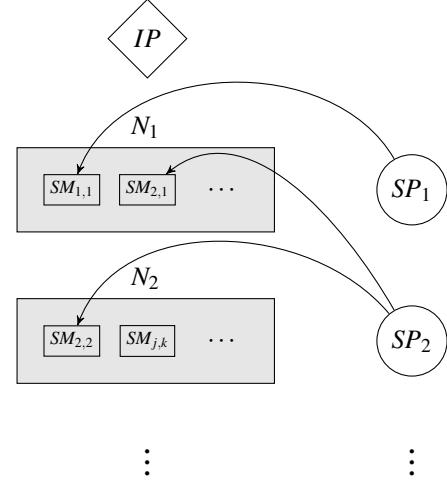


Figure 1: Overview of our system model. IP provides a number of nodes N_i on which software providers SP_j can deploy software modules $SM_{j,k}$.

set of microprocessor-based systems that we refer to as *nodes* N_i . A variety of third-party *software providers* SP_j are interested in using the infrastructure provided by IP . They do so by deploying *software modules* $SM_{j,k}$ on the nodes administered by IP . Figure 1 provides an overview.

This abstract setting is an adequate model for many ICT systems today, and the nodes in such systems can range from high-performance servers (for instance in a cloud system), over smart cards (for instance in GlobalPlatform-based systems) to tiny microprocessors (for instance in sensor networks). In this paper, we focus on the low end of this spectrum, where nodes contain only a small embedded processor.

Any system that supports extensibility (through installation of software modules) by several software providers must implement measures to make sure that the different modules can not interfere with each other in undesired ways (either because of bugs in the software, or because of malice). For high- to mid-end systems, this problem is relatively well-understood and good solutions exist. Two important classes of solutions are (1) the use of virtual memory, where each software module gets its own virtual address space, and where an operating system or hypervisor implements and guards communication channels between them (for instance shared memory sections or inter-process communication channels), and (2) the use of a memory-safe virtual machine (for instance a Java VM) where software modules are deployed in memory-safe bytecode and a security architecture in the VM guards the interactions between them.

For low-end systems with cheap microprocessors, providing adequate security measures for the setting sketched

¹Sancus was the ancient Roman god of trust, honesty and oaths.

above is still an open problem, and an active area of research [16]. One straightforward solution is to transplant the higher-end solutions to these low-end systems: one can extend the processor with virtual memory, or one can implement a Java VM. This will be an appropriate solution in some contexts, but there are two important disadvantages. First, the cost (in terms of required resources such as chip surface, power or performance) is non-negligible. And second, these solutions all require the presence of a sizable trusted software layer (either the OS or hypervisor, or the VM implementation).

The problem we address in this paper is the design, implementation and evaluation of a novel security architecture for low-end systems that is inexpensive and that does not rely on any trusted software layer. The TCB on the networked device is *only* the hardware. More precisely, a software provider needs to trust only his own software modules; he does not need to trust any infrastructural or third-party software on the nodes, only the hardware of the infrastructure and his own modules.

2.2 Attacker model

We consider attackers with two powerful capabilities.

First, we assume attackers can manipulate *all* the software on the nodes. In particular, attackers can act as a software provider and can deploy malicious modules to nodes. Attackers can also tamper with the operating system (for instance because they can exploit a buffer overflow vulnerability in the operating system code), or even install a completely new operating system.

Second, we assume attackers can control the communication network that is used by *IP*, software providers and nodes to communicate with each other. Attackers can sniff the network, can modify traffic, or can mount man-in-the-middle attacks.

With respect to the cryptographic capabilities of the attacker, we follow the Dolev-Yao attacker model [11]: attackers can not break cryptographic primitives, but they can perform protocol-level attacks.

Finally, attacks against the hardware are out of scope. We assume the attacker does not have physical access to the hardware, can not place probes on the memory bus, can not disconnect components and so forth. While physical attacks are important, the addition of hardware-level protections is an orthogonal problem that is an active area of research in itself [2, 6, 25, 26]. The addition of hardware-level protection will be useful for many practical applications (in particular for sensor networks) but does not have any direct impact on our proposed architecture or on the results of this paper.

2.3 Security properties

For the system and attacker model described above, we want our security architecture to enforce the following security properties:

- *Software module isolation.* Software modules on a node run *isolated* in the sense that no software outside the module can read or write its runtime state, and no software outside the module can modify the module’s code. The only way for other software on the node to interact with a module is by calling one of its designated entry points.
- *Remote attestation.* A software provider can verify with high assurance that a specific software module is loaded on a specific node of *IP*.
- *Secure communication.* A software provider can receive messages from a specific software module on a specific node with authenticity, integrity and freshness guarantees. For simplicity we do not consider confidentiality properties in this paper, but our approach could be extended to also provide confidentiality guarantees.
- *Secure linking.* A software module on a node can link to and call another module on the same node with high assurance that it is calling the intended module. The runtime interactions between a module *A* and a module *B* that *A* links to can not be observed or tampered with by other software on the same node.

Obviously, these security properties are not entirely independent of each other. For instance, it does not make sense to have secure communication but no isolation: given the power of our attackers, any message could then simply be modified right after its integrity was verified by a software module.

2.4 Application scenarios

This section illustrates some real-world application scenarios where the security properties above are relevant. Today’s ICT environments involve many parties using shared resources. This is not different for the sensor space where applications have moved from the monolithic, often static, single application domain (such as wildlife [13] or volcano monitoring [46]) to a dynamic and long-lived setting characterized by platform-application decoupling [24] and resource sharing [33].

We present two illustrating scenarios. First, consider the logistics domain [48]. Given node cost and complexity, powerful nodes can be attached to containers, but nodes attached to packages are low-end and resource-constrained. The package is under control of the package

owner, the *IP*, a pharmaceutical company in this example scenario. This pharmaceutical wants a software module for continuous cold-chain visibility of the package.² In the warehouse, the shipping company wants to load a radio-location software module to expedite package processing. In the harbor, because of customs regulations like C-TPAT [44], the container owner needs to attest manifest validity and package integrity, requiring yet a different software module on the package node.

Another representative scenario is found in the medical domain, where a hospital is equipped with a variety of nodes used for many processes simultaneously, with most of those processes security sensitive [29]. Building nodes, for example, support facility management with software modules for Heating, Ventilation, and Air Conditioning (HVAC) or fire control and physical security, but are also used for patient tracking and monitoring of vital signals, an application where strong security requirements are present with respect to health information. The same nodes can even automate the supply chain by supporting asset and inventory management of medical goods through a localization and tracking software module.

The above scenarios establish a clear need for isolation, attestation, secure communication and secure linking of the various software modules reflecting the dynamic objectives of the various stakeholders. We believe these scenarios are strong evidence for the value of the Sancus architecture.

3 Design of Sancus

The main design challenge is to realize the desired security properties *without trusting any software on the nodes*, and under the constraint that nodes are low-end resource constrained devices. An important first design choice that follows from the resource constrained nature of nodes is that we limit cryptographic techniques to symmetric key. While public key cryptography would simplify key management, the cost of implementing public key cryptography in hardware is too high [31].

We present an overview of our design, and then we zoom in on the most interesting aspects.

3.1 Overview

Nodes. Nodes are low-cost, low-power microcontrollers (our implementation is based on the TI MSP430). The processor in the nodes uses a von Neumann architecture with a single address space for instructions and data. To distinguish actual nodes belonging to *IP* from fake nodes set up by an attacker, *IP* shares a symmetric

key with each of its nodes. We call this key the *node master key*, and use the notation K_N for the node master key of node N . Given our attacker model where the attacker can control all software on the nodes, it follows that this key must be managed by the hardware, and it is only accessible to software in an indirect way.

Software Providers. Software providers are principals that can deploy software to the nodes of *IP*. Each software provider has a unique public ID *SP*.³ *IP* uses a key derivation function *kdf* to compute a key $K_{N,SP} = \text{kdf}(K_N, SP)$, which *SP* will later use to setup secure communication with its modules. Since node N has key K_N , nodes can compute $K_{N,SP}$ for any *SP*. The node will include a hardware implementation of *kdf* so that the key can be computed without trusting any software.

Software Modules. Software modules are essentially simple binary files containing two mandatory sections: a *text section* containing protected code and constants and a *protected data section*. As we will see later, the contents of the latter section are not attested and are therefore vulnerable to malicious modification before hardware protection is enabled. Therefore, the processor will zero-initialize its contents at the time the protection is enabled to ensure an attacker can not have *any* influence on a module’s initial state. Next to the two protected sections discussed above, a module can opt to load a number of *unprotected sections*. This is useful to, for example, limit the amount of code that can access protected data. Indeed, allowing code that does not need it access to protected data increases the possibility of bugs that could leak data outside of the module. In other words, this gives developers the opportunity to keep the trusted code of their own modules *as small as possible*. Each section has a header that specifies the start and end address of the section.

The *identity* of a software module consists of (1) the content of the text section and (2) the start and end addresses of the text and protected data sections. We refer to this second part of the identity as the *layout* of the module. It follows that two modules with the exact same code and data can coexist on the same node and will have different identities as their layout will be different. We will use notations such as *SM* or *SM*₁ to denote the identity of a specific software module.

Software modules are always loaded on a node on behalf of a specific software provider *SP*. The loading proceeds as expected, by loading each of the sections of the module in memory at the specified addresses. For each module loaded, the processor maintains the layout information in a *protected storage* area inaccessible from

²That is, the continuous monitoring of a temperature-controlled supply chain.

³Throughout this text, we will often refer to a software provider using its ID *SP*.

$$\begin{aligned}
K_N &= \text{Known by } IP \\
K_{N,SP} &= \text{kdf}(K_N, SP) \\
K_{N,SP,SM} &= \text{kdf}(K_{N,SP}, SM)
\end{aligned}$$

Figure 2: Overview of the keys used in Sancus. The node key K_N is only known by IP and the hardware. When SP is registered, it receives its key $K_{N,SP}$ from IP which can then be used to create module specific keys $K_{N,SP,SM}$.

software. It follows that the node can compute the identity of all modules loaded on the node: the layout information is present in protected storage and the content of the text section is in memory.

An important sidenote here is that the loading process is *not* trusted. It is possible for an attacker to intervene and modify the module during loading. However, this will be detected as soon as the module communicates with its provider or with other modules (see Section 3.3).

Finally, the node computes a symmetric key $K_{N,SP,SM}$ that is specific to the module SM loaded on node N by provider SP . It does so by first computing $K_{N,SP} = \text{kdf}(K_N, SP)$ as discussed above, and then computing $K_{N,SP,SM} = \text{kdf}(K_{N,SP}, SM)$. All these keys are kept in the protected storage and will only be available to software indirectly by means of new processor instructions we discuss later. Figure 2 gives an overview of the keys used by Sancus.

Note that the provider SP can also compute the same key, since he received $K_{N,SP}$ from IP and since he knows the identity SM of the module he is loading on N . This key will be used to attest the presence of SM on N to SP and to protect the integrity of data sent from SM on N to SP .

Figure 3 shows a schematic picture of a node with a software module loaded. The picture also shows the keys and the layout information that the node has to manage.

Memory protection on the nodes. The various modules on a node must be protected from interfering with each other in undesired ways by means of some form of memory protection. We base our design on the recently proposed *program-counter based memory access control* [43], as this memory access control model has been shown to support strong isolation [42] as well as remote attestation [14]. Roughly speaking, isolation is implemented by restricting access to the protected data section of a module such that it is only accessible while the program counter is in the corresponding text section of the same module. Moreover, the processor instructions that use the keys $K_{N,SP,SM}$ will be program counter dependent. Essentially the processor offers a special instruction

to compute a Message Authentication Code (MAC). If the instruction is invoked from within the text section of a specific module SM , the processor will use key $K_{N,SP,SM}$ to compute the MAC. Moreover, the instruction is only available after memory protection for module SM has been enabled. It follows that only a well-isolated SM installed on behalf of SP on N can compute MACs with $K_{N,SP,SM}$, and this is the basis for implementing both remote attestation and secure (integrity-protected) communication to SP .

Secure linking. A final aspect of our design is how we deal with secure linking. When a software provider sends a module SM_1 to a node, this module can specify that it wants to link to another module SM_2 on the same node, so that SM_1 can call services of SM_2 locally. SM_1 specifies this by including a MAC of (the identity of) SM_2 computed using the key K_{N,SP,SM_1} in an unprotected section.⁴ The processor includes a new special instruction that SM_1 can call to check that (1) there is a module loaded (with memory protection enabled) at the address of SM_2 and (2) the MAC of the identity of that module has the expected value.

This initial authentication of SM_2 is needed only once. In Section 3.5, we will discuss a more efficient procedure for subsequent authentications.

We currently do not incorporate *caller authentication* in our design. That is, SM_2 can not easily verify that it has been called by SM_1 . Note that this can in principle be implemented in software: SM_1 can call SM_2 providing a secret nonce as parameter. SM_2 can then call-back SM_1 , passing the same nonce, asking for acknowledgement that it had indeed been called by SM_1 . Future work will include caller authentication in the core of Sancus' design to make it more efficient and transparent.

Separating the various uses of MACs. Sancus uses MACs for a variety of integrity checks as well as for key derivation. Our design includes a countermeasure to avoid attacks where an attacker replays a MAC computed for one purpose in another context. In order to achieve separation between the different applications of MAC functions, we make sure the first byte of the input to the MAC function is different for each use case: 01 for the derivation of $K_{N,SP}$, 02 for the derivation of $K_{N,SP,SM}$, 03 for attestation and 04 for MAC computations on data.

Confidentiality. As mentioned in Section 2.3, we decided to not include confidentiality of communication in our design. However, since we provide attestation of modules and authentication of messages, confidentiality can

⁴Note that since this MAC depends on the load addresses of SM_1 and SM_2 , it may not be known until SM_1 has been deployed. If this is the case, SP can simply send the MAC after SM_1 is deployed and the load addresses are known.

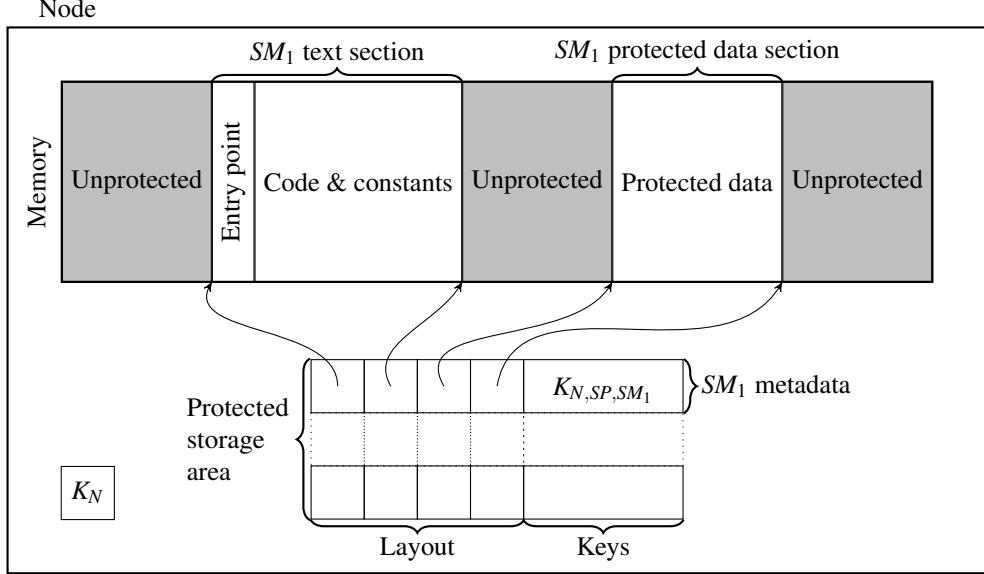


Figure 3: A node with a software module loaded. Sancus ensures the keys can never leave the protected storage area by only making them available to software in indirect ways through new processor instructions.

be implemented in software if necessary. One possibility is deploying a module with the public key of SP and a software implementation of the necessary cryptographic primitives. Another possibility is establishing a shared secret after deployment using a method such as Diffie-Hellman key exchange with authenticated messages. Note that implementing this last method is non-trivial due to the lack of a secure source of randomness. However, in the context of wireless sensor networks, methods have been devised to create cryptographically secure random number generators using only commonly available hardware [17].

Since the methods outlined above are expensive in terms of computation time and increase the TCB of modules, we are currently considering adding confidentiality to the core of Sancus' design. Exploring this is left as future work.

This completes the overview of our design. We now zoom in on the details of key management, memory access control, secure communication, remote attestation and secure linking.

3.2 Key management

We handle key management without relying on public-key cryptography [32]. IP is a trusted authority for key management. All keys are generated and/or known by IP . There are three types of keys in our design (Figure 2):

- Node master keys K_N shared between node N and IP .

- Software provider keys $K_{N,SP}$ shared between a provider SP and a node N .
- Software module keys $K_{N,SP,SM}$ shared between a node N and a provider SP , and the hardware of N makes sure that only SM can use this key.

We have considered various ways to manage these keys. A first design choice is how to generate the node master keys. We considered three options: (1) using the same node master key for every node, (2) randomly generating a separate key for every node using a secure random number generator and keeping a database of these keys at IP , and (3) deriving the master node keys from an IP master key using a key derivation function and the node identity N .

We discarded option (1) because for this choice the compromise of a single node master key breaks the security of the entire system. Options (2) and (3) are both reasonable designs that trade off the amount of secure storage and the amount of computation at IP 's site. Our prototype uses option (2).

The software provider keys $K_{N,SP}$ and software module keys $K_{N,SP,SM}$ are derived using a key derivation function as discussed in the overview section.

Finally, an important question is how compromised keys can be handled in our scheme. Since any secure key derivation function has the property that deriving the master key from the derived key is computationally infeasible, the compromise of neither a module key $K_{N,SP,SM}$ nor a provider key $K_{N,SP}$ needs to lead to the revocation of K_N . If $K_{N,SP}$ is compromised, provider SP should receive a

new name SP' since an attacker can easily derive $K_{N,SP,SM}$ for any SM given $K_{N,SP}$. If $K_{N,SP,SM}$ is compromised, the provider can still safely deploy other modules. SM can also still be deployed if the provider makes a change to the text section of SM .⁵ If K_N is compromised, it needs to be revoked. Since K_N is different for every node, this means that only one node needs to be either replaced or have its key updated.

3.3 Memory access control

Memory can be divided into (1) memory belonging to modules, and (2) the rest, which we refer to as unprotected memory. Memory allocated to modules is divided into two sections, the *text* section, containing code and public constants, and the *protected data* section containing all the data that should remain confidential and should be integrity protected. Modules can also have an *unprotected data* section that is considered to be part of unprotected memory from the point of view of the memory access control system.

Apart from application-specific data, run-time metadata such as the module’s call stack should typically also be included in the protected data section. Indeed, if a module’s stack were to be shared with untrusted code, confidential data may leak through stack variables or control-data might be corrupted by an attacker. It is the module’s responsibility to make sure that its call stack and other run-time metadata is in its protected data section, but our implementation comes with a compiler that ensures this automatically (see Section 4.2).

The memory access control logic in the processor enforces that (1) data in the protected data section of a module is only accessible while code in the text section of that module is being executed, and (2) the code in the text section can only be executed by jumping to a well-defined entry point. The second part is important since it prevents attackers from misusing code chunks in the text section to extract data from the protected data section. For example, without this guarantee, an attacker might be able to launch a Return-Oriented Programming (ROP) attack [7] by selectively combining gadgets found in the text section. Note that, as shown in Figure 3, our design allows modules to have a single entry point only. This may seem like a restriction but, as we will show in Section 4.2, it is not since multiple logical entry points can easily be dispatched through a single physical entry point. Table 1 gives an overview of the enforced access rights.

Memory access control for a module is enabled at the time the module is loaded. First, untrusted code (for instance the node operating system) will load the module

⁵For example, a random byte could be appended to the text section without changing the semantics of the module.

Table 1: Memory access control rules enforced by Sancus using the traditional Unix notation. Each entry indicates how code executing in the “from” section may access the “to” section.

From/to	Entry	Text	Protected	Unprotected
Entry	r-x	r-x	rw-	rwx
Text	r-x	r-x	rw-	rwx
Unprotected/				
Other SM	r-x	r--	---	rwx

in memory as discussed in Section 3.1. Then, a special instruction is issued:

`protect layout,SP`

This processor instruction has the following effects:

- the layout is checked not to overlap with existing modules, and a new module is registered by storing the layout information in the protected storage of the processor (see Section 3.1 and Figure 3);
- memory access control is enabled as discussed above; and
- the module key $K_{N,SP,SM}$ is created – using the text section and layout of the actually loaded module – and stored in the protected storage.

This explains why we do not need to trust the operating system that loads the module in memory: if the content of the text section, or the layout information would be modified before execution of the `protect` instruction, then the key generated for the module would be different and subsequent attestations or authentications performed by the module would fail. Once the `protect` instruction has succeeded, the hardware-implemented memory access control scheme ensures that software on the node can no longer tamper with SM .

The only way to lift the memory access control is by calling the processor instruction:

`unprotect`

The effect of this instruction is to lift the memory protection of the module *from which the unprotect instruction is called*. A module should only call `unprotect` after it has cleared the protected data section.

Finally, it remains to decide how to handle memory access violations. We opt for the simple design of resetting the processor and clearing memory on every reset. This has the advantage of clearly being secure for the security properties we aim for. However an important disadvantage is that it may have a negative impact on availability of the node: a bug in the software may cause

the node to reset and clear its memory. An interesting avenue for future work is to come up with strategies to handle memory access violations in less severe ways. Invalid reads could return some default value as in secure multi-execution [10]. Invalid writes or jumps could be dropped or modified to actions that are allowed as in edit-automata [35]. For instance, an invalid memory read might just return zero, and an invalid jump might be redirected to an exception handler.

3.4 Remote attestation and secure communication

The module key $K_{N,SP,SM}$ is managed by the hardware of the node, and it can only be used by software in two ways. The first way is by means of the following processor instruction (we discuss the second way in Section 3.5):

`MAC-seal start address, length, result address`

This instruction can only be called from within the text section of a protected module, and the effect is that the processor will compute the MAC of the data in memory starting at *start address* and up to *start address + length* using the module key of the module performing the instruction. The resulting MAC value is written to *result address*.

Modules can use this processor instruction to protect the integrity of data they send to their provider. The data plus the corresponding MAC can be sent using the untrusted operating system over an untrusted network. If the MAC verifies correctly (using $K_{N,SP,SM}$) upon receipt by the provider *SP*, he can be sure that this data indeed comes from *SM* running on *N* on behalf of *SP* as the node's hardware makes sure only this specific module can compute MACs with the module key $K_{N,SP,SM}$.

To implement remote attestation, we only need to add a freshness guarantee (i.e. protect against replay attacks). Provider *SP* sends a fresh nonce *No* to the node *N*, and the module *SM* returns the MAC of this nonce using the key $K_{N,SP,SM}$, computed using the `MAC-seal` instruction. This gives the *SP* assurance that the correct module is running on that node at this point in time.

Building on this scheme, we can also implement secure communication. Whenever *SP* wants to receive data from *SM* on *N*, it sends a request to the node containing a nonce *No* and possibly some input data *I* that is to be provided to *SM*. This request is received by untrusted code on the node which passes *No* and *I* as arguments to the function of *SM* to be called. When *SM* has calculated the output *O*, it asks the processor to calculate a MAC of $No||I||O$ using the `MAC-seal` instruction. This MAC is then sent along with *O* to *SP*. By verifying the MAC with its own copy of the module key, the provider has strong assurance that *O* has been produced by *SM* on node *N* given input *I*.

3.5 Secure linking and local communication

In this section, we discuss how we assure the secure linking property mentioned in Section 2.3. More specifically, we consider the situation where a module *SM₁* wants to call another module *SM₂* and wants to be ensured that (1) the integrity of *SM₂* has not been compromised, and (2) *SM₂* is correctly protected by the processor. The second point is important, and is the reason why *SM₁* can not just verify the integrity of the text section of *SM₂* by itself. *SM₁* will need help from the processor to give assurance that *SM₂* is loaded with the expected layout and that protection for *SM₂* is enabled.

In our design, if module *SM₁* wants to link securely to *SM₂*, *SM₁* should be deployed with a MAC of *SM₂* created with the module key K_{N,SP,SM_1} . The processor provides a special instruction to check the existence and integrity of a module at a specified address:

`MAC-verify address, expected MAC.`

This instruction will:

- verify that a module is loaded (with protection enabled) at the provided address;
- compute the MAC of the identity of that module using the module key of the module calling this instruction;
- compare the resulting MAC with the *expected MAC* parameter of the instruction; and
- if the MACs were equal, return the module's ID (to be explained below), otherwise return zero.

This is the second (and final) way in which a module can use its module key (next to the `MAC-seal` instruction discussed in Section 3.4).

Using this processor instruction, a module can securely check for the presence of another expected module, and can then call that other module.

Since this authentication process is relatively expensive (it requires the computation of a MAC), our design also includes a more efficient mechanism for repeated authentication. The processor will assign sequential IDs⁶ to modules that it loads, and will ensure that – within one boot cycle – it never reuses these IDs. A processor instruction:

`get-id address`

⁶To avoid confusion between the two different identity concepts used in this text, we will refer to the hardware-assigned number as *ID* while the text section and layout of a module is referred to as *identity*.

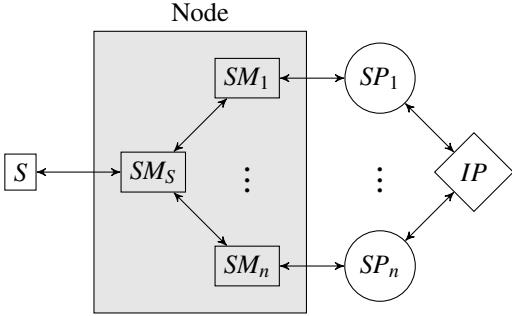


Figure 4: Setup of the sensor node example discussed in Section 3.6. Sancus ensures only module SM_S is allowed to directly communicate with the sensor S . Other modules securely link to SM_S to receive sensor data in a trusted way.

checks that a protected module is present at *address* and returns the ID of the module. Once a module has checked using the initial authentication method that the module at a given address is the expected module, it can remember the ID of that module, and then for subsequent authentications it suffices to check that the same module is still loaded at that address using the *get-id* instruction.

3.6 An end-to-end example

To make the discussion in the previous sections more concrete, this section gives a small example of how our design may be applied in the area of sensor networks. Figure 4 shows our example setup. It contains a single node to which a sensor S is attached; communication with S is done through memory-mapped I/O. The owner of the sensor network, IP , has deployed a special module on the processor, SM_S , that is in charge of communicating with S . By ensuring that the protected data section of SM_S contains the memory-mapped I/O region of S , IP ensures that no software outside of SM_S is allowed to configure or communicate directly with S ; all requests to S need to go through SM_S .

Figure 4 also shows a number of software providers (SP_1, \dots, SP_n) who have each deployed a module (SM_1, \dots, SM_n). In the remainder of this section, we walk the reader through the life cycle of a module in this example setup.

The first step for a provider SP is to contact IP and request permission to run a module on the sensor node. If IP accepts the request, it provides SP with its provider key for the node, $K_{N,SP}$.

Next, SP creates the module SM , that he wants to run on the processor and calculates the associated module key, $K_{N,SP,SM}$. Since SM will communicate with SM_S , SP requests the identity of SM_S from IP . A MAC of this iden-

tity, created with $K_{N,SP,SM}$, is included in an unprotected section of SM so that SM can use it to authenticate SM_S . Then SM is sent to the node for deployment.

Once SM is received on the node, it is loaded, by untrusted software like the operating system, into memory and the processor is requested to protect SM , using the *protect* processor instruction. As discussed, the processor enables memory protection, computes the key $K_{N,SP,SM}$ and stores it in hardware.

Now that SM has been deployed, SP can start requesting data from it. We will assume that SM 's function is to request data from S through SM_S , perform some transformation, filtering or aggregation on it and return the result to SP . The first step is for SP to send a request containing a nonce No to the node. Once the request is received (by untrusted code) on the node, SM is called passing No as an argument.

Before SM calls SM_S , it needs to verify the integrity of module SM_S . It does this by executing the *MAC-verify* instruction passing the address of the known MAC of SM_S and the address of the entry point it is about to call. The ID of SM_S is then returned to SM and, if it is non-zero, SM calls SM_S to receive the sensor data from S . SM will usually also store the returned ID of SM_S in its protected data section so that future authentications of SM_S can be done with the *get-id* instruction.

Once the received sensor data has been processed into the output data O , SM will request the processor to seal $No || O$ using the *MAC-seal* instruction. SM then passes this MAC together with O to the (untrusted) network stack to be sent to SP . When SP receives the output of SM , it can verify its integrity by recalculating the MAC.

4 Implementation

This section discusses the implementation of Sancus. We have implemented hardware support for all security features discussed in Section 3 as well as a compiler that can create software modules suitable for deployment on the hardware.

4.1 The processor

Our hardware implementation is based on an open source implementation of the TI MSP430 architecture: the openMSP430 from the OpenCores project [20]. We have chosen this architecture because both GCC and LLVM support it and there exists a lot of software running natively on the MSP430, for example the Contiki operating system.

The discussion is organized as follows. First, we explain the features added to the openMSP430 in order to implement the isolation of software modules. Then, we discuss how we added support for the attestation related

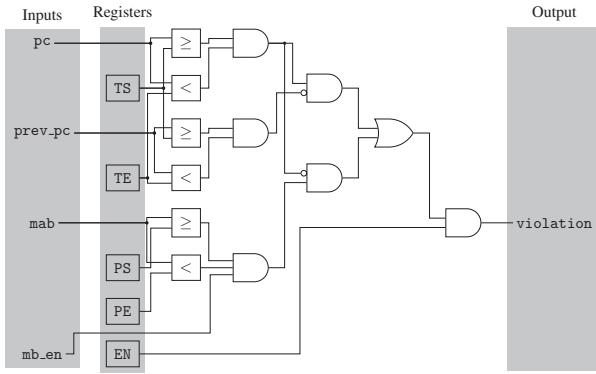


Figure 5: Schematic of the Memory Access Logic (MAL), the hardware used to enforce the memory access rules for a single protected module.

operations. Finally, we describe the modifications we made to the openMSP430 core itself.

Isolation. This part of the implementation deals with enforcing the access rights shown in Table 1. For this purpose, the processor needs access to the layout of every software module that is currently protected. Since the access rights need to be checked on every instruction, accessing these values should be as fast as possible. For this reason, we have decided to store the layout information in special registers inside the processor. Note that this means the total number of software modules that can be protected at any particular time has a fixed upper bound. This upper bound, N_{SM} , can be configured when synthesizing the processor.

Figure 5 gives an overview of the Memory Access Logic (MAL) circuit used to enforce the access rights of a single software module. This MAL circuit is instantiated N_{SM} times in the processor. It has four inputs: pc and prev_pc are the current and previous values of the program counter, respectively. The input mab is the memory address bus – the address currently used for load or store operations⁷ – while mb_en indicates whether the address bus is enabled for the current instruction. The MAL circuit has one output, violation, that is asserted whenever one of the access rules is violated.

Apart from the input and output signals, the MAL circuit also keeps state in registers. The layout of the protected software module is captured in the TS (start of text section), TE (end of text section), PS (start of protected section) and PE (end of protected section) registers. The EN register is set to 1 if there is currently a module being protected by this MAL circuit instantiation. The layout is saved in the registers when the protect instruction is

called at which time EN is also set. When the unprotect instruction is called, we just unset EN which disables all checks.

In our prototype we load new modules through a debug interface on the node and only the debug unit is allowed to write to the memory region where text sections are loaded. Therefore, the read-only nature of text sections is already enforced and the MAL does not need to check this. In a production implementation this check should be added and would cost two additional comparators in the MAL circuit.

Since the circuit is purely combinational, no extra cycles are needed for the enforcement of access rights. As explained above, this is exactly what we want since these rights need to be checked for every instruction. The only downside this approach might have is that this large combinational circuit may add to the length of the critical path of the processor. We will show in Section 5 that this is not the case. Note that since the MAL circuits are instantiated in parallel, N_{SM} does not influence the length of the critical path.

Apart from hardware circuit blocks that enforce the access rights, we also added a single hardware circuit to control the MAL circuit instantiations. It implements three tasks: (1) combine the violation signals from every MAL instantiation into a single signal; (2) keep track of the value of the current and previous program counter; and (3) when the protect instruction is called, select a free MAL instantiation to store the layout of the new software module and assign it a unique ID.

Attestation. As explained in Section 3, two cryptographic features are needed to implement our design: the ability to create MACs and a key derivation function. Since our implementation is based on a small microprocessor, one of our main goals here is to make the implementation of these features as small as possible.

The MAC algorithm we have chosen is HMAC, the hash-based message authentication code. One of the reasons we have chosen HMAC is its simplicity: only two calls of a hash function are needed to calculate a MAC. Another reason is that it serves as the basic building block for HKDF [28], a key derivation function. This means a lot of hardware can be shared between the implementations of the MAC and the key derivation function. For the hash function, we have chosen to use SPONGENT because it is one of the hash functions with the smallest hardware footprint published to date [5]. More specifically, we use the variant SPONGENT-128/128/8 implemented using a bit-parallel, word-serial architecture, which has a small footprint while maintaining acceptable throughput. Since SPONGENT-128/128/8 requires 8 bit inputs and the openMSP430 architecture is 16 bit, an 8 bit buffer and a tiny finite state machine are required to make the hash

⁷Of course, this includes implicit memory accesses like a call instruction.

implementation and the processor work together.

All the keys used by the processor are 128 bits long. The node key K_N is fixed when the hardware is synthesized and should be created using a secure random number generator. When a module SM is loaded, the processor will first derive $K_{N,SP}$ using the HKDF implementation which is then used to derive $K_{N,SP,SM}$. The latter key will then be stored in the hardware MAL instantiation for the loaded module. Note that we have chosen to cache the module keys instead of calculating them on the fly whenever they are needed. This is a trade-off between size and speed which we feel is justified because SPONGENT-128/128/8 needs about 8.75 cycles per input bit. Since the module key is needed for every remote attestation and whenever the module’s output needs to be signed, having to calculate it on the fly would introduce a runtime overhead that we expect to be too high for most applications.

Under assumptions on the underlying hash function, HMAC is known to be a pseudo-random function [4]. It is shown [28, Section 3] that this is sufficient for a key derivation function, provided that the key to the pseudo-random function (in our notation the first input to $kdf(.,.)$) is uniformly random or pseudo-random. This is the case in our application, hence there is no need to use the more elaborate “extract-and-expand” construction [28].

Core modifications. The largest modification that had to be made to the core is the decoding of the new instructions. We have identified a range of opcodes, starting at 0x1380, that is unused in the MSP430 instruction set and mapped the new instructions in that range.

Further modifications include routing the needed signals, like the memory address bus, into the access rights modules as well as connecting the violation signal to the internal reset. Note that the violation signal is stored into a register before connecting it to the reset line to avoid the asynchronous reset being triggered by combinational glitches from the MAL circuit.

Figure 6 gives an overview of the added hardware blocks when synthesized with support for two protected modules. In order to keep the figure readable, we did not add the input and output signals of the MAL blocks shown in Figure 5.

4.2 The compiler

Although the hardware modifications enable software developers to create protected modules, doing this correctly is tedious, as the module can have only one entry point, and as modules may need to implement their own call-stack to avoid leaking the content of stack allocated variables to unprotected code or to other modules. Hence, we have implemented a compiler extension based on LLVM [37] that deals with these low-level details. We

have also implemented a support library that offers an API to perform some commonly used functions like creating a MAC of data.

Our compiler compiles standard C files.⁸ To benefit from Sancus, a developer only needs to indicate which functions should be part of the protected module being created, which functions should be entry points and what data should be inside the protected section. For this purpose, we offer three attributes – SM_FUNC, SM_ENTRY and SM_DATA – that can be used to annotate functions and global variables.

Entry points. Since the hardware supports a single entry point per module only, the compiler implements multiple logical entry points on top of the single physical entry point by means of a jump table. The compiler assigns every logical entry point a unique ID. When calling one of the logical entry points, the ID of that entry point is placed in a register before jumping to the physical entry point of the module. The code at the physical entry point then jumps to the correct function based on the ID passed in the register.

When a module calls a function outside its text section, the same entry point is also used when this function returns. This is implemented by using a special ID for the “return entry point”. If this ID is provided when entering the module, the address to return to is loaded from the module’s stack. Of course, this is only safe if stack switching is also used.

Stack switching. As discussed in Section 3.3, it is preferable to place the runtime stack of software modules inside the protected data section. Our compiler automatically handles everything needed to support multiple stacks. For every module, space is reserved at a fixed location in its protected section for the stack. The first time a module is entered, the stack pointer is loaded with the address of this start location of the stack. When the module is exited, the current value of the stack pointer is stored in the protected section so that it can be restored when the module is reentered.

Exiting modules. Our compiler ensures that no data is leaked through registers when exiting from a module. When a module exits, either by calling an external function or by returning, any register that is not part of the calling convention is cleared. That is, only registers that hold a parameter or a return value retain their value.

Secure linking. Calls to protected modules are automatically instrumented to verify the called module. This

⁸We use Clang [36] as our compiler frontend. This means any C-dialect accepted by Clang is supported.

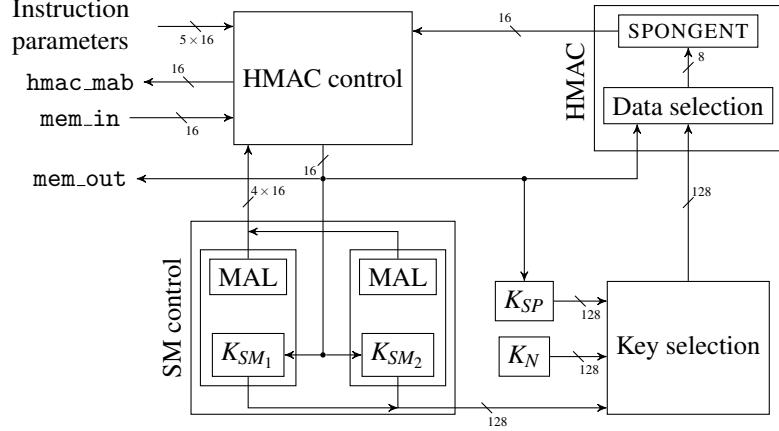


Figure 6: Overview of the hardware blocks added to the openMSP430 core.

includes automatically calculating any necessary module keys and MACs. Of course, a software provider needs to provide its key to the compiler for this function to work.

4.3 Deployment

Since the identity of a SM is dependent on its load addresses on node N , SP must be aware of these addresses in order to be able to calculate $K_{N,SP,SM}$. Moreover, any MACs needed for secure linking will also be dependent on the load addresses of other modules. Enforcing static load addresses is obviously not a scalable solution given that we target systems supporting dynamic loading of software modules by third-party software providers.

Given these difficulties, we felt the need to develop a proof-of-concept software stack providing a scalable deployment solution. Our stack consists of two parts: a set of tools used by SP to deploy SM on N and host software running on N . Note that this host software is *not* part of any protected module and, hence, does not increase the size of the TCB.

We will now describe the deployment process implemented by our software stack. First, SP creates a relocatable Executable and Linkable Format (ELF) file of SM and sends it to N . The host software on N receives this file, finds a free memory area to load SM and relocates it using a custom made dynamic ELF loader. Then, hardware protection is enabled for SM and a symbol table is sent back to SP . This symbol table contains the addresses of any global functions⁹ as well as the load addresses of all protected modules on N . Using this symbol table, SP is able to reconstruct the exact same image of SM as the one loaded on N . This image can then be used to calculate $K_{N,SP,SM}$ and any needed MACs. These MACs can

then be sent to N to be loaded in memory. Note that this deployment process has been *fully* automated.

After SM has been deployed, the host software on N provides an interface to be able to call its entry points. This can be used by SP to attest that SM has not been compromised during deployment and that the hardware protection has been correctly activated.

5 Evaluation

In this section we evaluate Sancus in terms of runtime performance, power consumption, impact on chip size and provided security. All experiments were performed using a Xilinx XC6SLX9 Spartan-6 FPGA running at 20MHz.

Performance A first important observation from the point of view of performance is that our hardware modifications do not impact the processor’s critical path. Hence, the processor can keep operating at the same frequency, and any code that does not use our new instructions runs at the same speed. This is true independent of the number of software modules N_{SM} supported in the processor.¹⁰ The performance results below are also independent of N_{SM} .

To quantify the impact on performance of our extensions, we first performed microbenchmarks to measure the cost of each of the new instructions. The `get-id` and `unprotect` instructions are very fast: they both take one clock cycle. The other three instructions compute hashes or key derivations, and hence their run time cost depends linearly on the size of the input they handle. We summarize their cost in Table 2. Note that since `MAC-seal` and `MAC-verify` both compute the HMAC of the input data, one might expect that they would need the same number

⁹For example, `libc` functions and I/O routines.

¹⁰We verified this experimentally for values of N_{SM} up to 8.

Table 2: The number of cycles needed by the new instructions for various input sizes. The input for the instructions is as follows: `protect`: the text section of the software module being protected; `MAC-seal`: the data to be signed; and `MAC-verify`: the text section of the software module to be verified.

Instruction	256B	512B	1024B
<code>protect</code>	30,344	48,904	86,016
<code>MAC-seal</code>	24,284	42,848	79,968
<code>MAC-verify</code>	24,852	43,416	80,536

of cycles. However, since `MAC-verify` includes the layout of the module to be verified in the input to HMAC, it has a fixed overhead of 568 cycles.

To give an indication of the impact on performance in real-world scenarios, we performed the following macro benchmark. We configured our processor as in the example shown in Figure 4. We measured the time it takes from the moment a request arrives at the node until the response is ready to be sent back. More specifically, the following operations are timed: (1) The original request is passed, together with the nonce, to SM_i ; (2) SM_i requests SM_S for sensor data; (3) SM_i performs some transformation on the received data; and (4) SM_i signs its output together with the nonce. The overhead introduced by Sancus is due to a call to `MAC-verify` in step (2) and a call to `MAC-seal` in step (4) as well as the entry and exit code introduced by the compiler. Since this overhead is fixed, the amount of computation performed in step (3) will influence the *relative overhead* of Sancus. Note that the size of the text section of M_S is 218 bytes and that nonces and output data signed by M_i both have a size of 16 bits.

By using the `Timer_A` module of the MSP430, we measured the fixed overhead to be 28,420 cycles for the first time data is requested from the module. Since the call to `MAC-verify` in step (2) is not needed after the initial verification, we also measured the overhead of any subsequent requests, which is 6,341 cycles. Given these values, the relative overhead can be calculated in function of the number of cycles used during the computation in step (3). The result is shown in Figure 7.

We believe that these numbers are clear evidence of the practicality of our approach.

Area The unmodified Spartan-6 FPGA implementation of the openMSP430 uses 998 slice registers and 2,322 slice LUTs. The fixed overhead¹¹ of our modification is 586 registers and 1,138 LUTs. For each protected module, there is an additional overhead of 213 registers and 307 LUTs.

¹¹That is, the overhead when $N_{SM} = 0$.

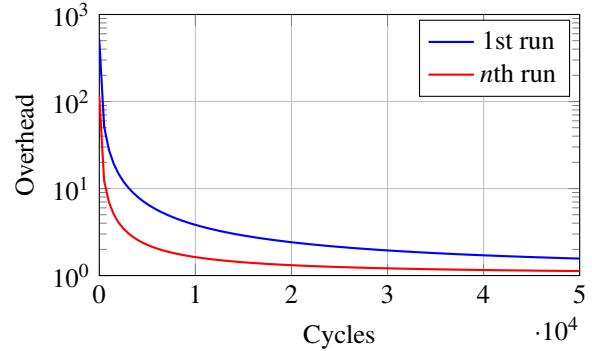


Figure 7: Relative overhead, in function of the number of cycles used for calculations, of Sancus on the macro benchmark. The n th run is significantly faster due to the secure linking optimization discussed in Section 3.5.

There are two easy ways to improve these numbers. First, if computational overhead is of lesser concern, the module key may be calculated on the fly instead of storing it in registers. Second, in applications with lower security requirements, smaller keys may be used reducing the number registers used for storage as well as the internal state of the SPONGENT implementation. Exploring other improvements is left as future work.

Power Our static power analysis tool¹² predicts an increase of power consumption for the processor of around 6% for the processor running at 20MHz. We measured power consumption experimentally, but could not detect a significant difference between an unmodified processor and our Sancus prototype. Of course, since Sancus introduces a runtime overhead, the total overhead in *energy* consumption will be accordingly.

Security We provide an informal security argument for each of the security properties Sancus aims for (see Section 2.3).

First, *software module isolation* is enforced by the memory access control logic in the processor. Both the access control model as well as its implementation are sufficiently simple to have a high assurance in the correctness of the implementation. Moreover, Agten et al. [1] have shown that higher-level isolation properties (similar to isolation between Java components) can be achieved by compiling to a processor with program-counter dependent memory access control. Sancus does *not* protect against vulnerabilities in the implementation of a module. If a module contains buffer-overflows or other memory safety related vulnerabilities, attackers can exploit them using well-known techniques [15] to get unintended access to

¹²We used Xilinx XPower Analyzer.

data or functionality in the module. Dealing with such vulnerabilities is an orthogonal problem, and a wide range of countermeasures for addressing them has been developed over the past decades [47].

The security of *remote attestation and secure communication* both follow from the following key observation: the computation of MACs with the module key is only possible by a module with the correct identity running on top of a processor configured with the correct node key (and of course by the software provider of the module). As a consequence, if an attacker succeeds in completing a successful attestation or communication with the software provider, he must have done it with the help of the actual module. In other words, within our attacker model, only API-level attacks against the module are possible, and it is indeed possible to develop modules that are vulnerable to such attacks, for instance if a module offers a function to compute MACs with its module key on arbitrary input data. But if the module developer avoids such API-level attacks, the security of Sancus against attackers conforming to our attacker model follows.

The security of *secure linking* is the most intricate security property of Sancus. It follows again from the fact that computation of MACs with the module key is only possible by a module with the correct identity running on top of a processor configured with the correct node key, or by the software provider of the module. Hence, an attacker can not forge MACs of other modules that a module wants to link to and call. Because of our technique for separation of uses of MACs (Section 3.1), he can also not do this by means of an API level attack against the module. As a consequence, if a module implements a MAC-verify check for any module it calls¹³, this verification can only be successful for modules for which the software provider has deployed the MAC. Hence the module will only call modules that its provider has authorized it to call.

6 Related Work

Ensuring strong isolation of code and data is a challenging problem. Many solutions have been proposed, ranging from hardware-only to software-only mechanisms, both for high-end and low-end devices.

Isolation in high-end devices. The Multics [9] operating system marked the start of the use of protection rings to isolate less trusted software. Despite decades of research, high-end devices equipped with this feature are still being attacked successfully. More recently, research has switched to focus on the isolation of software modules with a minimal TCB by relying on recently added hardware support. McCune et al. propose Flicker [39], a

system that relies on a TPM chip and trusted computing functionality of modern CPUs, to provide strong isolation of modules with only a TCB of 250 LOCs. Subsequent research [3, 38, 40, 42] focuses on various techniques to reduce the number of TPM accesses and significantly increase performance, for example by taking advantage of hardware support for virtual machines.

The idea of deriving module specific keys from a master key using (a digest of) the module’s code is also used by the On-board Credentials project [27]. They use existing hardware features to enforce the isolated execution of *credential programs* and securely store secret keys. Only one credential program can effectively be loaded at any single moment but the concept of *families* is introduced to be able to share secrets between different programs. Although secure communication is implemented using symmetric cryptography, they rely on public key cryptography during the deployment process.

Isolation in low-end devices. While recent research results on commodity computing platforms are promising, the hardware components they rely on require energy levels that significantly exceed what is available to many embedded devices such as pacemakers [22] and sensor nodes. A lack of strong security measures for such devices significantly limits how they can be applied and vendors may be required to develop closed systems or leave their system vulnerable to attack.

Sensor operating systems and applications, for example, were initially compiled into a monolithic and static image without safety considerations, as in early versions of TinyOS [34]. The reality that sensor deployments are long-lived, and that the full set of modules and their detailed functionality is often unknown at development time, resulted in dynamic modular operating systems such as SOS [23] or Contiki [12]. As stated in the introduction of this paper, the availability of networked modular update capability creates new threats, particularly if the software modules originate from different stakeholders and can no longer be fully trusted. Many ideas have been put forward to address the safety concerns of these shared environments, and solutions to provide memory protection, isolation and (fair) multithreading have appeared. t-kernel [21] rewrites code on the sensor at load time. Coarse-grained memory protection (basically MMU emulation) is available for the SOS operating system by sandboxing in the Harbor system [30] through a combination of backend compile time rewriting and run time checking on the sensor. Safe TinyOS [8] equally uses a combination of backend compile time analysis and minimal run time error handlers to provide type and memory safety. Java’s language features and the Isolate mechanism are used on the Sun SPOT embedded platform using the Squawk VM [41]. SenShare [33] provides a virtual

¹³Note that our compiler automatically adds these checks.

machine for TinyOS applications. While these proposed solutions do not require any hardware modifications, they all incur a software-induced overhead. Moreover, third-party software providers must rely on the infrastructure provider to correctly rewrite modules running on the same device.

To increase security of embedded devices, Strackx et al. [43] introduced the idea of a program-counter based access control model, but without providing any implementation. Agten et al. [1] prove that isolation of code and data within such a model only relies on the vendor of the module and cannot be influenced by other modules on the same system. More recently El Defrawy et al. [14] implemented hardware support that allows attestation that a module executed correctly without any interference, based on a similar access control model. While this is a significant step forward, it does not provide isolation as sensitive data cannot be kept secret from other modules between invocations.

7 Conclusion

The increased connectivity and extensibility of networked embedded devices as illustrated for instance by the trend towards decoupling applications and platform in sensor networks leads to exciting new applications, but also to significant new security threats. This paper proposed a novel security architecture called Sancus, that is low-cost yet provides strong security guarantees with a very small, hardware-only, TCB.

8 Availability

To ensure reproducibility and verifiability of our results, we make the hardware design and the software of our prototype publicly available. All source files, as well as binary packages and documentation can be found at <https://distrinet.cs.kuleuven.be/software/sancus/>.

9 Acknowledgments

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. With the financial support from the Prevention of and Fight against Crime Programme of the European Union (B-CENTRE).

References

- [1] AGTEN, P., STRACKX, R., JACOBS, B., AND PIJSESENS, F. Secure compilation to modern processors. In *2012 IEEE 25th Computer Security Foundations Symposium (CSF 2012)* (Los Alamitos, CA, USA, 2012), IEEE Computer Society, pp. 171–185.
- [2] ANDERSON, R. J., AND KUHN, M. G. Low cost attacks on tamper resistant devices. In *Proceedings of the 5th International Workshop on Security Protocols* (London, UK, UK, 1998), Springer-Verlag, pp. 125–136.
- [3] AZAB, A., NING, P., AND ZHANG, X. Sice: a hardware-level strongly isolated computing environment for x86 multi-core platforms. In *Proceedings of the 18th ACM conference on Computer and communications security* (2011), ACM, pp. 375–388.
- [4] BELLARE, M., CANETTI, R., AND KRAWCZYK, H. Keying hash functions for message authentication. In *Proceedings of the 16th Annual International Cryptology Conference on Advances in Cryptology* (London, UK, UK, 1996), CRYPTO '96, Springer-Verlag, pp. 1–15.
- [5] BOGDANOV, A., KNEZEVIC, M., LEANDER, G., TOZ, D., VARICI, K., AND VERBAUWHEDE, I. Spongent: The design space of lightweight cryptographic hashing. vol. 99, IEEE Computer Society, p. 1.
- [6] BONEH, D., DEMILLO, R. A., AND LIPTON, R. J. On the importance of eliminating errors in cryptographic computations. *J. Cryptology* 14 (2001), 101–119.
- [7] CASTELLUCIA, C., FRANCILLON, A., PERITO, D., AND SORIENTE, C. On the difficulty of software-based attestation of embedded devices. In *Proceedings of the 16th ACM conference on Computer and communications security* (New York, NY, USA, 2009), CCS '09, ACM, pp. 400–409.
- [8] COOPRIDER, N., ARCHER, W., EIDE, E., GAY, D., AND REGEHR, J. Efficient memory safety for TinyOS. In *Proceedings of the 5th international conference on Embedded networked sensor systems* (New York, NY, USA, 2007), SenSys '07, ACM, pp. 205–218.
- [9] CORBATÓ, F., AND VYSSOTSKY, V. Introduction and overview of the Multics system. In *Proceedings of the November 30–December 1, 1965, Fall joint computer conference, part I* (1965), ACM, pp. 185–196.
- [10] DEVRIESE, D., AND PIJSESENS, F. Noninterference Through Secure Multi-Execution. In *Proceedings of the IEEE Symposium on Security and Privacy* (2010), pp. 109–124.
- [11] DOLEV, D., AND YAO, A. C. On the security of public key protocols. *IEEE Transactions on Information Theory* 29, 2 (1983), 198–208.
- [12] DUNKELS, A., FINNE, N., ERIKSSON, J., AND VOIGT, T. Runtime dynamic linking for reprogramming wireless sensor networks. In *Proceedings of the 4th international conference on Embedded networked sensor systems* (New York, NY, USA, 2006), SenSys '06, ACM, pp. 15–28.
- [13] DYK, V., ELLWOOD, S. A., MACDONALD, D. W., MARKHAM, A., MASCOLO, C., PÁSZTOR, B., TRIGONI, N., AND WOHLERS, R. Wildlife and environmental monitoring using RFID and WSN technology. In *Proceedings of the 7th ACM Conference on Embedded Networked Sensor Systems* (New York, NY, USA, 2009), SenSys '09, ACM, pp. 371–372.
- [14] ELDEFRAWY, K., FRANCILLON, A., PERITO, D., AND TSUDIK, G. SMART: Secure and Minimal Architecture for (Establishing a Dynamic) Root of Trust. In *NDSS 2012, 19th Annual Network and Distributed System Security Symposium, February 5-8, San Diego, USA* (San Diego, UNITED STATES, 02 2012).
- [15] ERLINGSSON, U., YOUNAN, Y., AND PIJSESENS, F. Low-level software security by example. In *Handbook of Information and Communication Security*. Springer, 2010.
- [16] FAROOQ, M. O., AND KUNZ, T. Operating systems for wireless sensor networks: A survey. *Sensors* 11, 6 (2011), 5900–5930.

- [17] FRANCILLON, A., AND CASTELLUCCIA, C. TinyRNG: A cryptographic random number generator for wireless sensors network nodes. In *In Modeling and Optimization in Mobile, Ad Hoc and Wireless Networks and Workshops, 2007. WiOpt 2007. 5th International Symposium on* (2007), pp. 1–7.
- [18] FRANCILLON, A., AND CASTELLUCCIA, C. Code injection attacks on Harvard-architecture devices. In *Proceedings of the 15th ACM conference on Computer and communications security* (New York, NY, USA, 2008), CCS '08, ACM, pp. 15–26.
- [19] GIANNETOS, T., DIMITRIOU, T., AND PRASAD, N. R. Self-propagating worms in wireless sensor networks. In *Proceedings of the 5th international student workshop on Emerging networking experiments and technologies* (New York, NY, USA, 2009), Co-Next Student Workshop '09, ACM, pp. 31–32.
- [20] GIRARD, O. openMSP430. <http://opencores.org/project/openmsp430>.
- [21] GU, L., AND STANKOVIC, J. A. t-kernel: providing reliable OS support to wireless sensor networks. In *Proceedings of the 4th international conference on Embedded networked sensor systems* (Boulder, Colorado, USA, 2006), ACM, pp. 1–14.
- [22] HALPERIN, D., HEYDT-BENJAMIN, T., RANSFORD, B., CLARK, S., DEFEND, B., MORGAN, W., FU, K., KOHNO, T., AND MAISEL, W. Pacemakers and implantable cardiac defibrillators: Software radio attacks and zero-power defenses. In *Security and Privacy, 2008. SP 2008. IEEE Symposium on* (2008), Ieee, pp. 129–142.
- [23] HAN, C.-C., KUMAR, R., SHEA, R., KOHLER, E., AND SRIVASTAVA, M. A dynamic operating system for sensor nodes. In *Proceedings of the 3rd international conference on Mobile systems, applications, and services* (New York, NY, USA, 2005), MobiSys '05, ACM, pp. 163–176.
- [24] HEINZELMAN, W. B., MURPHY, A. L., CARVALHO, H. S., AND PERILLO, M. A. Middleware to support sensor network applications. *IEEE Network* 18, 1 (2004), 6–14.
- [25] KOCHER, P. C. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In *Proceedings of the 16th Annual International Cryptology Conference on Advances in Cryptology* (London, UK, UK, 1996), CRYPTO '96, Springer-Verlag, pp. 104–113.
- [26] KOCHER, P. C., JAFFE, J., AND JUN, B. Differential power analysis. In *Proceedings of the 19th Annual International Cryptology Conference on Advances in Cryptology* (London, UK, UK, 1999), CRYPTO '99, Springer-Verlag, pp. 388–397.
- [27] KOSTAINEN, K., EKBERG, J.-E., ASOKAN, N., AND RANTALA, A. On-board credentials with open provisioning. In *Proceedings of the 4th International Symposium on Information, Computer, and Communications Security* (New York, NY, USA, 2009), ASIACCS '09, ACM, pp. 104–115.
- [28] KRAWCZYK, H., AND ERONEN, P. HMAC-based extract-and-expand key derivation function (HKDF). <http://tools.ietf.org/html/rfc5869>.
- [29] KUMAR, P., AND LEE, H.-J. Security issues in healthcare applications using wireless medical sensor networks: A survey. *Sensors* 12, 1 (2011), 55–91.
- [30] KUMAR, R., KOHLER, E., AND SRIVASTAVA, M. Harbor: software-based memory protection for sensor nodes. In *Proceedings of the 6th international conference on Information processing in sensor networks* (New York, NY, USA, 2007), IPSN '07, ACM, pp. 340–349.
- [31] LEE, Y. K., SAKIYAMA, K., BATINA, L., AND VERBAUWHEDE, I. Elliptic-curve-based security processor for RFID. *Computers, IEEE Transactions on* 57, 11 (nov. 2008), 1514–1527.
- [32] LEIGHTON, F. T., AND MICALI, S. Secret-key agreement without public-key cryptography. In *Proceedings of the 13th Annual International Cryptology Conference on Advances in Cryptology* (London, UK, UK, 1994), CRYPTO '93, Springer-Verlag, pp. 456–479.
- [33] LEONTIADIS, I., EFSTRATIOU, C., MASCOLO, C., AND CROWCROFT, J. Senshare: transforming sensor networks into multi-application sensing infrastructures. In *Proceedings of the 9th European conference on Wireless Sensor Networks* (Berlin, Heidelberg, 2012), EWSN'12, Springer-Verlag, pp. 65–81.
- [34] LEVIS, P. Experiences from a decade of tinyos development. In *Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation* (Berkeley, CA, USA, 2012), OSDI'12, USENIX Association, pp. 207–220.
- [35] LIGATTI, J., BAUER, L., AND WALKER, D. Edit automata: enforcement mechanisms for run-time security policies. *International Journal of Information Security* 4, 1-2 (2005), 2–16.
- [36] LLVM DEVELOPER GROUP. Clang. <http://clang.llvm.org/>.
- [37] LLVM DEVELOPER GROUP. LLVM. <http://llvm.org/>.
- [38] MCCUNE, J. M., LI, Y., QU, N., ZHOU, Z., DATTA, A., GLIGOR, V., AND PERRIG, A. TrustVisor: Efficient TCB reduction and attestation. In *Proceedings of the IEEE Symposium on Security and Privacy* (May 2010).
- [39] MCCUNE, J. M., PARNO, B., PERRIG, A., REITER, M. K., AND ISOZAKI, H. Flicker: An execution infrastructure for TCB minimization. In *Proceedings of the ACM European Conference in Computer Systems (EuroSys)* (Apr. 2008), ACM, pp. 315–328.
- [40] SAHITA R, WARRIOR U., D. P. Protecting Critical Applications on Mobile Platforms. *Intel Technology Journal* 13 (2009), 16–35.
- [41] SIMON, D., CIFUENTES, C., CLEAL, D., DANIELS, J., AND WHITE, D. JavaTM on the bare metal of wireless sensor devices: the squawk java virtual machine. In *VEE* (2006), H.-J. Boehm and D. Grove, Eds., ACM, pp. 78–88.
- [42] STRACKX, R., AND PIJSESENS, F. Fides: Selectively hardening software application components against kernel-level or process-level malware. In *Proceedings of the 19th ACM Conference on Computer and Communications Security (CCS 2012)*, (Oct. 2012), ACM Press, pp. 2–13.
- [43] STRACKX, R., PIJSESENS, F., AND PRENEEL, B. Efficient isolation of trusted subsystems in embedded systems. In *Lecture Notes of the Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering: Security and Privacy in Communication Networks* (September 2010), vol. 50, Springer, pp. 1–18.
- [44] U.S. CUSTOMS AND BORDER PROTECTION. C-TPAT. <http://www.cbp.gov/ctpat>.
- [45] VIEGA, J., AND THOMPSON, H. The state of embedded-device security (spoiler alert: It's bad). *Security Privacy, IEEE* 10, 5 (Sept.–Oct. 2012), 68–70.
- [46] WERNER-ALLEN, G., LORINCZ, K., JOHNSON, J., LEES, J., AND WELSH, M. Fidelity and yield in a volcano monitoring sensor network. In *Proceedings of the 7th symposium on Operating systems design and implementation* (Berkeley, CA, USA, 2006), OSDI '06, USENIX Association, pp. 381–396.
- [47] YOUNAN, Y., JOOSEN, W., AND PIJSESENS, F. Runtime countermeasures for code injection attacks against c and c++ programs. *ACM Comput. Surv.* 44, 3 (June 2012), 17:1–17:28.
- [48] ZÖLLER, S., REINHARDT, A., MEYER, M., AND STEINMETZ, R. Deployment of wireless sensor networks in logistics: potential, requirements, and a testbed. In *Proceedings of the 9th GI/ITG KuVS Fachgespräch Drahtlose Sensornetze* (Sep 2010), R. Kolla, Ed., Julius-Maximilians-Universität Würzburg, pp. 67–70.

Securing Computer Hardware Using 3D Integrated Circuit (IC) Technology and Split Manufacturing for Obfuscation

Frank Imeson, Arij Emtenan, Siddharth Garg, and Mahesh V. Tripunitara

ECE, University of Waterloo, Canada
{fcimeson,aemtenan,siddharth.garg,tripunit}@uwaterloo.ca

Abstract

The fabrication of digital Integrated Circuits (ICs) is increasingly outsourced. Given this trend, security is recognized as an important issue. The threat agent is an attacker at the IC foundry that has information about the circuit and inserts covert, malicious circuitry. The use of 3D IC technology has been suggested as a possible technique to counter this threat. However, to our knowledge, there is no prior work on how such technology can be used effectively. We propose a way to use 3D IC technology for security in this context. Specifically, we obfuscate the circuit by lifting wires to a trusted tier, which is fabricated separately. This is referred to as split manufacturing. For this setting, we provide a precise notion of security, that we call k -security, and a characterization of the underlying computational problems and their complexity. We further propose a concrete approach for identifying sets of wires to be lifted, and the corresponding security they provide. We conclude with a comprehensive empirical assessment with benchmark circuits that highlights the security versus cost trade-offs introduced by 3D IC based circuit obfuscation.

1 Introduction

The security of digital integrated circuits (ICs), the building blocks of modern computer hardware systems, can be compromised by covertly inserted malicious circuits. The threat from such maliciously inserted hardware is of increasing concern to government and military agencies [2] and commercial semiconductor vendors. Recently, Skorobogatov et al. [28] demonstrated the presence of a backdoor in a military grade FPGA manufactured by Actel that enabled access to configuration data on the chip. The authors initially conjectured that the backdoor was maliciously inserted since the key used to trigger the backdoor was undocumented. Actel has since clarified that the backdoor was inserted by design for in-

ternal test purposes [23]. Nonetheless, this incident has further heightened the perceived threat from maliciously inserted hardware, and effective counter-measures to deter or prevent such attacks are of increasing importance.

The threat of maliciously inserted hardware arises from two factors. First, owing to their complexity, digital ICs are designed at sites across the world. In addition, parts of the design are often outsourced or purchased from external vendors. Second, a majority of semiconductor design companies are fabless, i.e., they outsource IC manufacturing to a potentially untrusted external fabrication facility (or foundry). Both factors make it easier for a malicious attacker in a design team or a malicious foundry (or a collusion between the two) to insert covert circuitry in a digital IC.

Three-dimensional (3D) integration, an emerging IC manufacturing technology, is a promising technique to enhance the security of computer hardware. A 3D IC consists of two or more independently manufactured ICs that are vertically stacked on top of each other — each IC in the stack is referred to as a *tier*. Interconnections between the tiers are accomplished using vertical metal pillars referred to as through-silicon vias (TSV).

3D IC manufacturing can potentially enhance hardware security since each tier can be manufactured in a separate IC foundry, and vertically stacked in a secure facility. Thus, a malicious attacker at any one foundry has an incomplete view of the entire circuit, reducing the attacker's ability to alter the circuit functionality in a desired manner.

Tezarron, a leading commercial provider of 3D stacking capabilities, has alluded to the enhanced security offered by 3D integration in a white paper [1]. The white paper notes that “A multi-layer circuit may be divided among the layers in such a way that the function of each layer becomes obscure. Assuming that the TSV connections are extremely fine and abundant, elements can be scattered among the layers in apparently random fashion.” However, the paper does not provide any formal

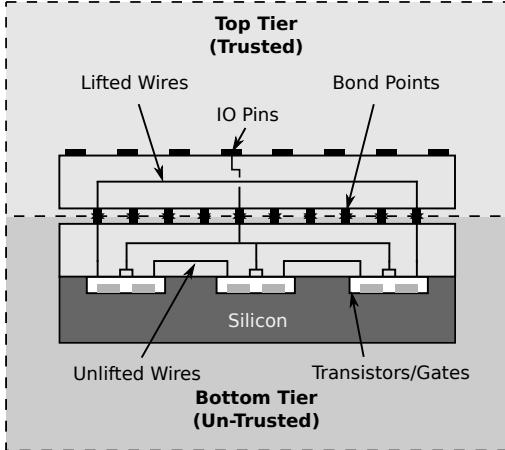


Figure 1: A two tier 3D IC. In this instance, the top tier is an interposer, i.e., it only implements metal wires, while the bottom tier has both transistors/gates and wires.

notion of security for split manufacturing, nor does it propose techniques to quantify security or achieve a certain security level. These are the open challenges that we address in this paper.

Our threat model assumes a malicious attacker in an IC foundry who wants to modify the functionality of a digital IC in a specific, targeted manner. The attack proposed by King et al. [19] modifies the state of hardware registers in a processor to raise the privilege level of the attacker — this is an example of a targeted attack since it requires the attacker to determine the gate or wire in the circuit that corresponds to the privilege bit. Fault insertion attacks in cryptographic hardware also require that certain vulnerable bits be targeted. For example, it has been shown that if the LSB bit of the 14th round of a DES implementation is set to logic zero, the secret key can be recovered in as few as two messages [9]. However, to succeed, the attacker must be able to determine which gate corresponds to the LSB bit of the 14th round.

To effect a targeted attack, an attacker must first identify specific logic gates or wires in the circuit that implement the functionality that he wants to monitor and/or modify; for example, the gate or wire that corresponds to the privilege bit for the privilege escalation attack proposed in [19]. A malicious foundry can identify the functionality of every gate and wire in the circuit if it gets to fabricate the entire chip, i.e., if a conventional planar, 2D fabrication process is used. On the other hand, as we show in this paper, 3D integration significantly reduces the ability of an attacker in a malicious foundry to correctly identify gates or wires in the circuit that he wants to attack.

The specific 3D integration technology that we exploit in this work, since it is the only one that is currently in

large volume commercial production [8], splits a design into two tiers. The bottom tier consists of digital logic gates and metal wires used to interconnect logic gates. The top tier, also referred to as an *interposer*, only consists of metal wires that provide additional connections between logic gates on the bottom tier.

The bottom tier — this tier is expensive to fabricate since it implements active transistor devices and passive metal — is sent to an external, untrusted foundry for fabrication. This is referred to as the untrusted tier. The top tier implements only passive metal and can be fabricated at lower cost in a trusted fabrication facility. We refer to this tier as the trusted tier.

Assume, for the sake of argument, that all interconnections between logic gates are implemented on the trusted tier, the attacker (who only has access to the untrusted tier) observes only a “sea” of disconnected digital logic gates. From the perspective of the attacker, gates of the same type, for example all NAND gates, are therefore indistinguishable from each other. (Assuming that the relative size or placement of gates reveals no information about interconnections between gates. This is addressed in Section 4.) Assume also that the attacker wants to attack a specific NAND gate in the circuit, and not just *any* NAND gate. The attacker now has two choices: (a) the attacker could randomly pick one NAND gate to attack from the set of indistinguishable NAND gates, and only succeed in the attack with a certain probability; or (b) the attacker could attack all indistinguishable NAND gates, primarily in cases where the attacker wants to monitor but not modify gates in the circuit, at the expense of a larger malicious circuit and thus, an increased likelihood of the attack being detected. In either instance, the attacker’s ability to effect a malicious, targeted attack on the circuit is significantly hindered. We refer to this technique as *circuit obfuscation*.

In general, we define a *k*-secure gate as one that, from the attacker’s perspective, cannot be distinguished from $k - 1$ other gates in the circuit. Furthermore, a *k*-secure circuit is defined as one in which each gate is at least *k*-secure.

Contributions We make the following contributions:

- We propose a concrete way of leveraging 3D IC technology to secure digital ICs from an active attacker at the foundry. Whereas the use of 3D IC technology for security has been alluded to before, we are not aware of prior work like ours that discusses how it can be used meaningfully.
- We propose a formal notion of security in this context that we call *k*-security. We give a precise characterization of the underlying technical problems — computing *k*-security and deciding which wires to lift — and identify their computational complexity.

- We have devised a concrete approach to addressing the problem of lifting wires, which comprises a greedy heuristic to identify a candidate set of wires to be lifted, and the use of a constraint (SAT) solver to compute k -security.
- We have conducted a thorough empirical assessment of our approach on benchmark circuits, including a case-study of a DES circuit, that illustrates the inability of an attacker to effectively attack circuits secured using 3D IC based obfuscation.

2 Preliminaries and Related Work

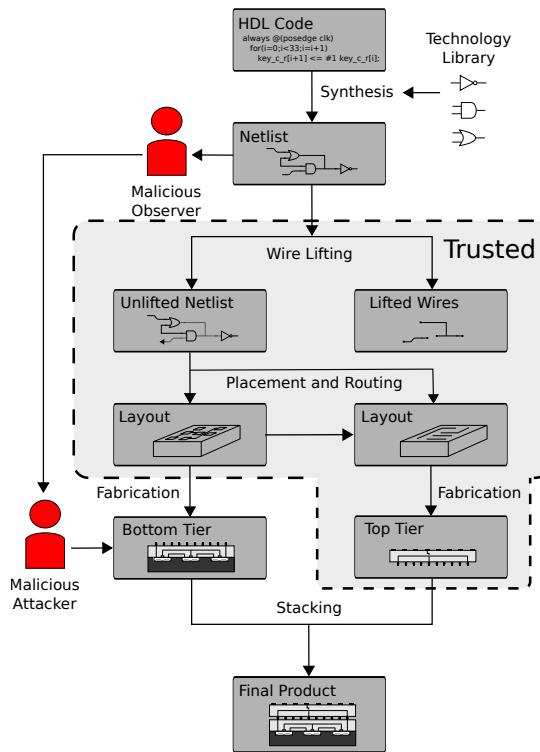


Figure 2: Secure 3D IC design and fabrication flow.

In this section, we overview the IC manufacturing process in the specific context of 3D integration, and discuss the attack model that we assume in this paper. We also discuss related work on hardware security including both attacks and countermeasures, and on the use 3D integration for enhancing the security of computer hardware.

2.1 3D IC Design and Fabrication

Digital ICs consist of a network of inter-connected digital logic gates. This network of gates is often referred to as a netlist. Digital logic gates are built using complementary metal-oxide-semiconductor (CMOS) transis-

tors. In a conventional planar/2D IC, CMOS transistors, and by extension digital logic gates, lie in a single layer of silicon. In addition, there are several layers of metal wires used to inter-connect the gates.

3D integration enables the vertical stacking of two or more planar ICs. Each IC in the vertical stack is referred to as a tier. Vertical interconnects (TSVs) are provided to allow the transistors and metal wires in each tier to connect to each other.

The initial motivation for 3D integration came from the potential reduction in the average distance between logic gates — in a 3D IC, the third, vertical dimension can be used to achieve a tighter packing of logic gates [6]. However, a number of issues, including high power density, temperature and cost, have plagued high volume, commercial availability of logic-on-logic 3D ICs [13].

A more practical 3D IC technology that has been demonstrated in a commercial product (a Xilinx FPGA [8]) is shown in Figure 1. It consists of two tiers. The bottom tier contains both transistors/gates and metal wires, while the top tier, the interposer, contains only metal wires. The two tiers are interfaced using uniformly spaced metallic bond-points. TSVs make use of these bond-points to provide connections between wires in the top and bottom tiers. This technology has also been referred to as 2.5D integration [14]. In the rest of this paper, we use 3D instead of 2.5D since our techniques can easily be generalized to full 3D.

Since the bottom tier consists of CMOS transistors, it is fabricated at one of the few foundries worldwide with advanced lithographic capabilities at high cost. The top tier, i.e., the interposer, only contains passive metal and can be fabricated at significantly reduced cost [21].

Figure 2 shows a 3D IC design flow with appropriate modifications for security. The design flow begins with the design specified using a hardware description language (HDL), which is then synthesized to a netlist of gates. The types of gates allowed in the gate netlist are specified in a technology library.

In the **wire lifting** stage, the edges (or wires) in the netlist that are to be implemented on the top tier are selected. These are referred to as **lifted wires**. The rest of the netlist, implemented on the bottom tier, is referred to as the **unlifted netlist** and consists of unlifted gates and unlifted wires.

The unlifted gates are then placed on the surface of the bottom tier, i.e., the (x, y) co-ordinates for each gate are selected. Unlifted wires are routed using the bottom tier metal layers. Two bond-points are selected for every lifted wire; one each for the two gates that the wire connects. The gates are connected to the corresponding bond-points. Finally, lifted wires are routed between pairs of bond-points in the top tier using the top tier routing resources.

Finally, the two tiers are fabricated at separate foundries. The chips from the two foundries are vertically stacked to create the final 3D IC chip that is shipped to the vendor.

We now discuss the attack model that we address in this paper, in the context of the 3D design and fabrication flow outlined above.

2.2 Attack Model

The attack model that we address in this paper is that of a **malicious attacker** in the foundry. This attack model has been commonly used in the hardware security literature because of the serious threat it presents [18]. We further strengthen the attack by assuming a **malicious observer** in the design stage, working in collusion with the malicious attacker in the foundry.¹ The malicious observer has full knowledge of the circuit as it goes through the design process, but can not effect any changes. The malicious attacker in the foundry can, on the other hand, effect changes in the circuit layout before the chip is fabricated.

To defend against this attack, the following steps of the design and fabrication flow are assumed to be secure, i.e., executed by a trusted party: (a) the wire lifting, placement and routing steps in the design, and (b) the fabrication of the top tier (therefore also referred to as the trusted tier).

Discussion Three aspects of the attack and defense models deserve further mention. First, we note that the attack model described above subsumes a number of other practically feasible attack models. It is stronger than a malicious attacker in the foundry working by himself. It is also stronger than a malicious attacker in the foundry with partial design knowledge — for example, the attacker is likely to know the functionality and input/output behaviour of circuit he is attacking (an ALU or a DES encryption circuit, *etc.*). Providing the attacker with the precise circuit netlist can only strengthen the attack.

Second, the steps in the design and fabrication process that are assumed to be trusted are also relatively easy to perform in a secure manner, compared to the untrusted steps. Wire lifting and placement/routing (in the design stage) are performed using automated software tools, the former based on algorithms that we propose in this paper, and the latter using commercially available software from electronic design automation (EDA) vendors. In comparison, writing the HDL code is manually intensive, time-consuming and costly. Furthermore, only the top tier is fabricated in a trusted foundry. The top tier only

¹Note that 3D IC based circuit obfuscation cannot, and is not intended to, defend against malicious attackers in the design stage who can alter the HDL or circuit netlist.

consists of passive metal wires that are inexpensive compared to the active CMOS transistors and metal wires in the untrusted, bottom tier [21].

Finally, we assume that all IC instances are manufactured before being sent out for stacking. If this were not the case, an attacker could intercept a stacked IC and reverse engineer the connections on the top tier. Armed with this knowledge, the attacker could then insert malicious hardware in future batches of the IC as they are being fabricated in the foundry.

2.3 Related Work

In this section, we discuss related work in the literature on hardware security and, specifically, the use of 3D ICs in this context. We also discuss the relationship of our work to database and graph anonymizing mechanisms.

Hardware Security Malicious circuits are expected to consist of two components, a trigger and the attack itself. The trigger for the attack can be based on data, for example when a specific cheat code appears at selected wires in the circuit [19], or on time, i.e., the trigger goes off after a certain period of time once the IC is shipped [33].

Once triggered, the malicious attack can either transmit or leak sensitive information on the chip, modify the circuit functionality or degrade the circuit performance. Tehranipoor and Koushanfar discuss a number of specific backdoors that fall within one of these categories [31].

Countermeasures against malicious attacks can be categorized in various ways. Design based countermeasures modify or add to the design of the circuit itself to provide greater security. These include N-variant IC design [4], data encryption for computational units [33] and adding run-time monitors to existing hardware [32]. Our work falls within this category. In contrast, testing based counter-measures use either pre-fabrication or post-fabrication testing and validation to detect, and in some cases, disable malicious circuits. A survey of these techniques can be found in [11].

Another way to categorize countermeasures is by their impact on the attack. Countermeasures to detect malicious circuits include IC fingerprinting [3] and unused circuit identification [17]. Some countermeasures can be used to disable malicious circuitry; for example, the power cycling based defense against timer triggers [33]. The proposed defense mechanism aims to deter attackers by hiding a part of the circuit and making it more difficult for the attacker to effect a successful attack.

3D Integration for Hardware Security Valamehr et al. [32] also exploit 3D integration capabilities to enhance the security of computer hardware, although in a manner orthogonal to ours. Their proposal involves adding a “control tier” on top of a regular IC to moni-

tor the activity of internal wires in the IC in a cost effective way. By monitoring internal wires on the chip, the control tier is able to detect potentially malicious activity and take appropriate recourse. Adding the monitors vertically on top of the IC to be protected reduces the power and performance cost of monitoring the IC. A similar technique was proposed by Bilzor [7].

Our technique exploits 3D integration in a different way, i.e., we use it to provide a malicious attacker in an IC foundry with an incomplete view of the circuit netlist, thus deterring the attack. Although the potential for this kind of defense mechanism has been alluded to before by Tezarron [1], ours is the first work, to our knowledge, to address this technique in any consequential way.

Hardware Obfuscation Hardware obfuscation techniques have been proposed to make circuits more difficult to reverse engineer. In particular, Roy et al. [26] augment a combinational circuit with key bits in such a way that the circuit only provides correct outputs when the key bits are set to pre-determined values. Rajendran et al. [24] further strengthen this defense mechanism by increasing the bar on the attacker to determine the secret key.

A difference between key-based circuit obfuscation mechanisms and circuit obfuscation via split manufacturing is that the notion of security in the former is conditioned on the computational capabilities of the attacker. In contrast, our notion of security is unconditional in that no matter the computational capabilities of the attacker, he cannot distinguish each gate from $k - 1$ other gates. We note that these mechanisms are not necessarily mutually exclusive — it might be possible to leverage split manufacturing based circuit obfuscation to further strengthen key-based circuit obfuscation, or vice-versa.

Independent of this work, Rajendran et al. [25] have recently examined the security obtained from split manufacturing. However, the authors provide no well-founded notion of security for split manufacturing, as we do in this paper. The authors do not address the wire lifting problem at all, and implicitly assume that the circuit is partitioned using traditional min-cut partitioning heuristics. Finally, it is assumed that the attacker reconstructs the circuit by simply connecting the closest gates with disconnected inputs/outputs.

Anonymizing Databases and Social Networks Our work bears relationship to prior work on anonymizing databases and social network graphs, but also has significant differences. A database is k -anonymous if the information for each individual is indistinguishable from $k - 1$ other individuals [30] in the database. The notion of k -anonymity for a social network is similar, except that instead of operating on relational data, it operates on a graph. Two individuals in a social network

are indistinguishable if their local neighbourhoods are the same [34].

In our setting, the similarity of the local neighborhood of two gates is only a necessary but not sufficient condition for indistinguishability. This is because the attacker is assumed to have access to the original circuit netlist and an incomplete view of the same netlist, and must thus match *all* gates in the incomplete netlist to gates in the original netlist.

The circuit obfuscation problem also introduces a number of distinct practical issues. These include the additional information that might be conveyed by the circuit layout (for example, the physical proximity of gates), and the role of the number of gate types in the technology library.

3 Problem Formulation

In this section, we formulate the circuit obfuscation problem that we address in this paper as a problem in the context of directed graphs. We begin by discussing the example circuit for a full adder that we show in Figure 3.

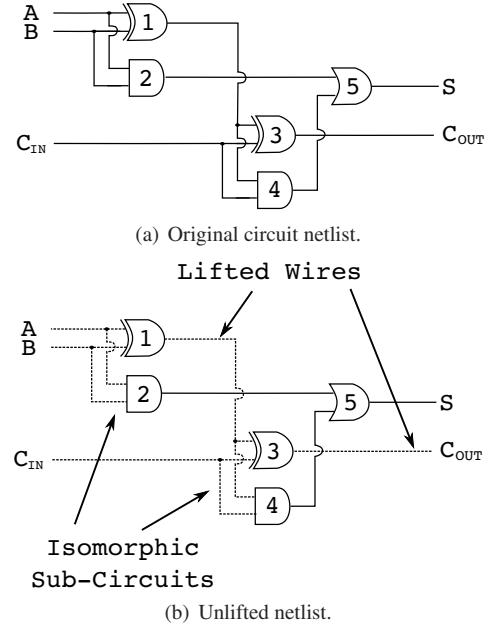


Figure 3: Original and unlifted netlists corresponding to a full adder circuit. Grey wires in the unlifted netlist are lifted and are not observed by the attacker.

Example As we mention in Section 1, in the most powerful attack model we consider, an attacker is in possession of two pieces of information: the originally designed (complete) circuit netlist, and the layout of the circuit that is sent to the foundry for fabrication, which we call the unlifted netlist. The latter results from the

defender lifting wires from the former. Assume that the defender chooses to lift the wires $A \rightarrow \{1, 2\}$, $B \rightarrow \{1, 2\}$, $C_{IN} \rightarrow \{3, 4\}$, $1 \rightarrow \{3, 4\}$ and $3 \rightarrow C_{OUT}$.

Note that gates in the unlifted netlist in Figure 3(b) are labeled differently from those in the original circuit in Figure 3(a). This reflects the fact that the attacker obtains the original circuit netlist and the unlifted netlist in completely different formats. The original netlist is a set of gates and wires in HDL format. On the other hand, the unlifted netlist is reconstructed from the circuit layout, which is a set of shapes and their locations on the surface of the chip, as also discussed in Section 4.3. The labeling and ordering of objects in the circuit layout file is unrelated to that in the netlist of the original circuit. Although not required, the defender can perform an additional random re-labeling and re-ordering step before the layout of H is sent to the foundry.

Given these two pieces of information, the attacker now seeks a bijective mapping of gates in the unlifted netlist to gates in the complete circuit netlist. If the attacker is successful in obtaining the correct mapping, he can carry out a targeted attack on any gate (or gates) of his choosing. The security obtained from lifting wires in the context of this example can be explained as follows. From the attacker’s perspective, either Gate u or Gate w in the unlifted netlist could correspond to Gate 1 in the original netlist. Thus the attacker’s ability to carry out a targeted attack on Gate 1 is hindered. The same can be said for the attacker’s ability to carry out a targeted attack on Gate 2, 3 or 4. However, note that the attacker can determine the identity of Gate 5 with certainty — it must correspond to Gate y since this is the only OR gate in the netlist. Thus, in this example, the lifting does not provide any security for Gate 5.

Informally, our notion of security is based on the existence of multiple isomorphisms (mappings) between gates in the unlifted netlist and the original netlist. In our example, there exist 4 distinct bijective mappings between the gates in the unlifted and original netlists. However, this notion of security may be seen as too permissive. It can be argued that given the fact that across all mappings, gate 5 is mapped uniquely, we have no security at all (i.e., security of 1). A more restrictive notion of security, one that we adopt in this paper, requires that for *each* gate in the original netlist, there exist at least k different gates in the unlifted netlist that map to it over all isomorphisms. This is intended to capture the intuition that the attacker is unable to uniquely identify even a single gate. We now formalize our notion of security.

3.1 Formulation as a Graph Problem

We now formulate our problem as a graph problem. A circuit can be perceived as a directed graph — gates are

vertices, and wires are edges. The direction of an edge into or out of a vertex indicates whether it is an input or output wire to the gate that corresponds to the vertex. If G is a graph, we denote its set of vertices as $V[G]$, and its set of edges as $E[G]$. Each vertex in the graph is associated with a color that is used to distinguish types of gates (e.g, AND and OR) from one another. Consequently, a graph G is a 3-tuple, $\langle V, E, c \rangle$, where V is the set of vertices, E the set of edges and the function $c: V \rightarrow \mathbb{N}$ maps each vertex to a natural number that denotes its color. For example, the circuit in Figure 3 and its unlifted portion can be represented by the graphs in Figure 4.

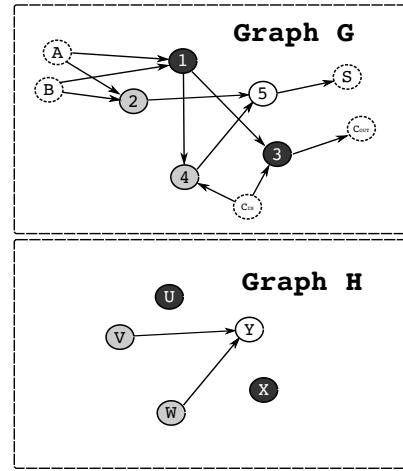


Figure 4: Full adder graphs: G is the full graph representation of the full adder circuit, H is the remaining graph after wires have been lifted.

A main challenge for the defender is to lift wires in a way that provides security. Our notion of security corresponds to a certain kind of subgraph isomorphism.

Definition 1 (Graph isomorphism). *Given two graphs $G_1 = \langle V_1, E_1, c_1 \rangle, G_2 = \langle V_2, E_2, c_2 \rangle$, we say that G_1 is isomorphic to G_2 if there exists a bijective mapping $\phi: V_1 \rightarrow V_2$ such that $\langle u, v \rangle \in E_1$ if and only if $\langle \phi(u), \phi(v) \rangle \in E_2$ and $c_1(u) = c_2(\phi(u)), c_1(v) = c_2(\phi(v))$. That is, if we rename the vertices in G_1 according to ϕ , we get G_2 . A specific such mapping ϕ is called an isomorphism.*

Definition 2 (Subgraph isomorphism). *We say that $G_1 = \langle V_1, E_1, c_1 \rangle$ is a subgraph of $G_2 = \langle V_2, E_2, c_2 \rangle$ if $V_1 \subseteq V_2$, and $\langle u, v \rangle \in E_1$ only if $\langle u, v \rangle \in E_2$. We say that G is subgraph isomorphic to H if a subgraph of G is isomorphic to H . The corresponding mapping is called a subgraph isomorphism.*

For example, in Figure 4, a subgraph isomorphism, ϕ , is $\phi(1) = U, \phi(2) = V, \phi(3) = X, \phi(4) = W, \phi(5) = Y$.

Intuition Let G be the graph that represents the original circuit with all wires, and H the graph of the circuit

after wires have been lifted. Then, the attacker knows that G is subgraph isomorphic to H . What he seeks is the correct mapping of vertices in G to H (or vice versa). This is equivalent to him having reconstructed the circuit, and now, he can effect his malicious modifications to the circuit that corresponds to H .

From the defender's standpoint, therefore, what we seek intuitively is that there be several subgraph isomorphisms between G and H . As we mention in Section 1, this then gives the kind of security in a k -anonymity sense — the attacker cannot be sure which of the mappings is the correct one, and therefore is able to reconstruct the circuit with probability $1/k$ only. As we mention there and discuss in more detail in the related work Section, though our notion of security has similarities to k -anonymity, there are important differences, and we call it k -security instead.

k -security We now specify our notion of security. We do this in three stages. (1) We first define a problem that captures our intuition of a gate being indistinguishable from another gate. We do this by requiring the existence of a particular kind of subisomorphic mapping between graphs that represent circuits. (2) We then define the notion of a k -secure gate. Such a gate is indistinguishable from at least $k - 1$ other gates in the circuit. (3) Finally, we define the notion of k -security, which is security across all gates in the circuit. This definition requires simply that every gate in the circuit is k -secure.

In the following definition, we characterize the problem GATE-SUBISO, which captures (1) above — a notion of what it means for a gate to be indistinguishable from another.

Definition 3 (GATE-SUBISO). *Given as input $\langle G, E', u, v \rangle$, where G is a DAG, $E' \subseteq E[G]$, and two distinct vertices $u, v \in V[G]$, let H be the graph we get by removing the edges that are in E' from G . Then, GATE-SUBISO is the problem of determining whether there exists a mapping $\phi: V[G] \rightarrow V[H]$ that is a subgraph isomorphism from G to H such that $\phi(u) = v$.*

The above definition is a special case of the well-known subgraph isomorphism problem [16]. In the subgraph isomorphism problem, we take as input two graphs A, B , and ask whether B is subgraph isomorphic to A . In GATE-SUBISO, both the graphs G, H are restricted to be DAGs, and H is a specific subgraph of G — one with some edges removed from G . Of course, we know that H is subgraph isomorphic to G , with the identity mapping from a vertex to itself serving as evidence (a certificate). However, in the GATE-SUBISO problem, we require the existence of a subgraph isomorphism that is different from the identity mapping, and furthermore, require that the vertex u be mapped under that subgraph isomorphism to a specific vertex v .

The intuition behind GATE-SUBISO is the following. G is the graph that corresponds to the original circuit, and H is the graph that corresponds to the circuit after wires are lifted. The above definition for GATE-SUBISO asks whether there exists a mapping under which the vertex u in the original circuit is indistinguishable from v in the unlifted circuit. That is, given that $u \neq v$, an attacker does not know whether u in G corresponds to u or v in H .

Based on GATE-SUBISO above, we now define the notion of a k -secure gate. It captures the intuition that the gate is indistinguishable from at least $k - 1$ other gates.

Definition 4 (k -secure gate). *Given a DAG, G , a vertex u in it, and a subgraph H of G constructed from G by removing some edges, $E' \subseteq E[G]$ only. We say that u is k -secure if there exist k distinct vertices v_1, \dots, v_k in G (and therefore in H), and mappings ϕ_1, \dots, ϕ_k from $V[G]$ to $V[H]$ such that every ϕ_i is a subgraph isomorphism from G to H , and for all $i \in [1, k]$, $\phi_i(u) = v_i$.*

The above definition expresses that u is indistinguishable from each of the v_i 's. Of course, one of the v_i 's may be u itself. Therefore, every gate is 1-secure, and if a gate is not 2-secure, then that gate is uniquely identifiable, for this particular choice of E' . The maximum that k can be is $|V[G]|$, the number of vertices in G .

Given the above definition for a k -secure gate, it is now straightforward to extend it to the entire graph (circuit). We do this with the following definition.

Definition 5 (k -security). *Given a DAG G , and a DAG H that we get from G by removing the edges from a set $E' \subseteq E[G]$. We say that $\langle G, E' \rangle$ is k -secure if every vertex in G is k -secure.*

The above definition is a natural extension of the notion of a k -secure gate, to every gate in the circuit. What it requires for k -security is that every vertex in the corresponding graph is indistinguishable from at least k vertices. We point out that some gates may be more than k -secure; k -security is a minimum across all gates. As the maximum k for any gate is $|V[G]|$, a graph can be, at best, $|V[G]|$ -secure. Every graph is 1-secure, which is the minimum.

We denote as $\sigma(G, E')$ the maximum k -security we are able to achieve with G, E' . In Figure 4, for example, we know that σ evaluates to 1, because the node 5 can be mapped to itself only. The nodes 1, 2, 3 and 4, however, are 2-secure gates. The reason is that each can be mapped either to itself, or to another node.

Computational complexity We now consider the computational complexity of determining the maximum k -security, σ . We consider a corresponding decision problem, k -SECURITY-DEC, which is the following. We are given as input $\langle G, E', k \rangle$ where G is a DAG, $E' \subseteq E[G]$

is a set of edges in G , and $k \in [1, |V[G]|]$. The problem is to determine whether lifting the edges in E' results in k -security.

We point out that if we have an oracle that decides k -SECURITY-DEC, then we can compute the maximum k -security we can get by lifting E' from G using binary search on k . That is, the problem of computing σ is easy if deciding k -SECURITY-DEC is easy.

Theorem 1. k -SECURITY-DEC $\in \text{NP}$ -complete under polynomial-time Turing reductions.

To prove the above theorem, we need to show that k -SECURITY-DEC is in NP , and that it is NP -hard. For the former, we need to present an efficiently (polynomial-sized) certificate that can be verified efficiently. Such a certificate is k mappings each of which is a subgraph isomorphism, for each vertex $u \in V[G]$. Each such mapping can be encoded with size $O(|V[G]|)$, and there are at most $k|V[G]|$ such mappings, and therefore the certificate is efficiently-sized. The verification algorithm simply checks that each mapping is indeed a subgraph isomorphism, and that u is mapped to a distinct vertex in each of the k mappings that corresponds to it. This can be done in time $O(|V[G]|^3)$.

We show that k -SECURITY-DEC is NP -hard under polynomial-time Turing reductions in the Appendix. (Henceforth, we drop the qualification “under polynomial-time Turing reductions,” and simply say NP -complete and NP -hard.) Indeed, our proof demonstrates that deciding even 2-security is NP -hard. The knowledge that k -SECURITY-DEC is NP -complete immediately suggests techniques for approaches for solving k -SECURITY-DEC, and thereby computing k -security. We discuss this further in the next section.

Choosing E' Lifting edges E' from G incurs a cost $c(G, E')$. A simple cost metric, one that we adopt in this paper is $c(G, E') = |E'|$, i.e., the cost is proportional to the number of lifted edges. Given the cost of lifting edges, the defender’s goal is to determine E' , the set of edges that should be lifted, such that $\sigma(G, E') \geq k$ and $c(G, E')$ is minimized.

We observe that from the standpoint of computational complexity, the problem of determining E' given G, k , where G is the graph and E' is the set of edges to be lifted so we get k -security, is no harder than k -SECURITY-DEC. That is, that problem is also in NP .

To prove this, we need to show that there exists an efficiently sized certificate that can be verified efficiently. Such a certificate is E' , and k subgraph isomorphisms for every vertex. The latter component of the certificate is the same as the one we used in our proof above for k -SECURITY-DEC’s membership in NP . The verification algorithm, in addition to doing what the verification al-

gorithm for k -SECURITY-DEC above does, also checks that E' is indeed a subset of G ’s edges.

We note that the k -security from lifting all the edges in G is no worse than lifting any other set of edges, and the k -security from lifting no edges in G is no better than lifting any other set of edges. More generally, given any n_1, n_2 such that $|E[G]| \geq n_1 > n_2$, we know that for every G , there exists a set of edges of size n_1 that if lifted, provides at least as much security as every set of edges of size n_2 . That is, there is a natural trade-off between the number of edges we lift, i.e., cost, and the security we achieve. In Section 4, we outline an approach to determine the cost-security trade-off using a greedy wire lifting procedure.

3.2 Discussion

Given our notion of k -security, a natural question to ask is whether there are stronger or different attack models for which k -security would be inadequate. We discuss this in the context of two attack models that differ from the one assumed. Finally, we also discuss a related question — that of the computational capabilities of the attacker.

General targeted attack models The notion of k -security is premised on an attack model in which the attacker needs to precisely identify one or more gates in the unlifted netlist, for example, the privilege escalation bit in a microprocessor [19] or the LSB of the 14th round in a DES implementation [9]. However, one can imagine a scenario in which the attack would be successful if the attacker correctly identifies any one of n gates. For example, there could be multiple privilege escalation bits in the microprocessor implementation.

More concretely, in the example in Figure 3, assume that the attacker wants to change the circuit functionality by inverting the output of Gate 2. The same objective can be accomplished by inverting the output of Gate 4. However, as we observe before, Gate v in the unlifted netlist must correspond to either Gate 2 or Gate 4. Thus, although this gate is 2-secure, the attack would be successful with probability 1.

Although our notion of security does not directly address the alternate attack model described above, it can be easily modified to do so. Say that the defender is aware that Gate v and Gate x are each equally vulnerable to the same kind of attack. Then, the defender can insist that Gate v is k -secure if and only if it is indistinguishable from $k - 1$ other gates excluding Gate x . Such information that the defender may have about the relative vulnerability of gates can be built into the notion of k -security.

Access to lifting procedure Our attack model strengthens the attacker with access to the original cir-

cuit netlist, G , along with the unlifted netlist H . Since the attacker has access to G , it is reasonable to ask if an even stronger attacker with access to G and the procedure used to lift wires would compromise security. It would not.

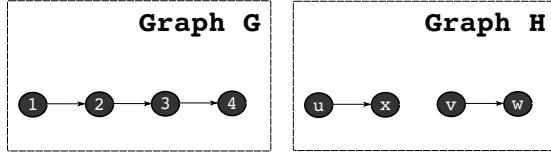


Figure 5: Example illustrating that the unlifted netlist H is 2-secure even if the attacker knows that edge $2 \rightarrow 3$ was lifted from original netlist G .

In fact, even if there is a deterministic choice of edges that must be lifted to provide a certain security level, knowledge of which edges are lifted does not compromise security, as long as G and H are differently labeled. We illustrate this with an example in Figure 5, where wire $2 \rightarrow 3$ must be lifted to provide 2-security. This knowledge does not compromise the security obtained from lifting. When there is choice, i.e., lifting two or more edges provides the same security, the choice is made uniformly at random. This is discussed in Section 4.

Computational capabilities of the attacker Our notion of k -security is not predicated on the computational capabilities of the attacker. In fact, we assume that the attacker is able to identify (all) subgraph isomorphisms from H to G . Nonetheless, given that the attacker’s goal might be to identify a single gate in the netlist, it is natural to ask why (and whether) the attacker needs determine a mapping for each gate in H .

In particular, the attacker can identify all gates in H with the same type and connectivity, i.e., number and type of gates it connects to, as the one he is interested in attacking. Prior work on k -anonymity for social network graphs assumes this kind of attack strategy. From the perspective of the attacker, this strategy is sub-optimal. This is because, for any gate in G that the attacker wants to target, this strategy will provide at least as many candidate mappings in H as the strategy in which the attacker enumerates all subgraph isomorphisms.

4 Approach

Having considered the computational complexity of the problem that underlies our work in the previous section, in this section, we propose a concrete approach for it. As our discussions in the prior section reveal, there are two parts to the solution: (a) computing the maximum

k -security for $\langle G, E' \rangle$, given the graph G that represents the complete circuit, and, (b) choosing the set E' .

We propose an approach for each in this section. For the problem of computing security, we employ constraint-solving. We discuss this in Section 4.1. For the problem of choosing E' , we propose a greedy heuristic. We discuss that in Section 4.2. We conclude this section with Section 4.3 with some practical considerations, specifically, scalability and layout-anonymization.

4.1 Computing Security

As shown in Section 3, the problem of determining the security level of circuit G , given the unlifted netlist H is **NP**-complete. Given the relationship of the problem to subgraph isomorphism, a natural approach to solving this problem would be to use graph (sub)isomorphism algorithms proposed in literature — of these, the VF2 algorithm [12] has been empirically shown to be the most promising [15]. However, in our experience, VF2 does not scale for circuits with > 50 gates (more on scalability in Section 4.3).

Instead, motivated by the recent advances in the speed and efficiency of SAT solvers, we reduce the sub-isomorphism problem to a SAT instance and use an off-the-shelf SAT solver to decide the instance.

Reduction to SAT Given graphs G and H , we define a bijective mapping ϕ from the vertex set of H to the vertex set of G as follows: Boolean variable ϕ_{ij} is true if and only if vertex $q_i \in H$ maps to a vertex $r_j \in G$. Here $V[G] = \{r_1, r_2, \dots, r_{|V[G]|}\}$ and $V[H] = \{q_1, q_2, \dots, q_{|V[H]|}\}$

We now construct a Boolean formula that is true if and only if graphs G and H are sub-isomorphic for the mapping ϕ . We will construct the formula in parts.

First, we ensure that each vertex in G maps to only one vertex in H :

$$F_1 = \prod_i^{|V[H]|} \sum_j^{|V[G]|} \left(\phi_{i,j} \prod_{k \neq i}^{|V[G]|} \neg \phi_{i,k} \right)$$

and vice-versa:

$$F_2 = \prod_j^{|V[G]|} \sum_i^{|V[H]|} \left(\phi_{i,j} \prod_{k \neq i}^{|V[H]|} \neg \phi_{k,j} \right)$$

Finally we need to ensure that each edge in H maps to an edge in G . Let $E[H] = \{e_1, e_2, \dots, e_{|E[H]|}\}$ and $E[G] = \{f_1, f_2, \dots, f_{|E[G]|}\}$. Furthermore, let $e_k = \langle q_{src(e_k)}, q_{dest(e_k)} \rangle \in E[H]$ and $f_k = \langle r_{src(f_k)}, r_{dest(f_k)} \rangle \in E[G]$. This condition can be expressed as follows:

$$F_3 = \prod_k^{|E[H]|} \sum_l^{|E[G]|} \phi_{src(e_k), src(f_l)} \wedge \phi_{dest(e_k), dest(f_l)}$$

The formula F that is input to the SAT solver is then expressed as a conjunction of the three formulae above: $F = F_1 \wedge F_2 \wedge F_3$. The formula F has $O(|V[H]| |V[G]|)$ variables and $O(|E[H]| |E[G]|)$ clauses.

4.2 Wire Lifting Procedure

To determine a candidate set of edges, E' , to lift, we employ a greedy heuristic. Our heuristic is shown as Algorithm 1.

```

1  $E' \leftarrow E[G]$ 
2 while  $|E'| > 0$  do
3    $s \leftarrow 0$ 
4   foreach  $e \in E'$  do
5      $E' \leftarrow E' - \{e\}$ 
6     if  $\sigma(G, E') > s$  then
7        $s \leftarrow \sigma(G, E')$ 
8        $e_b \leftarrow e$ 
9      $E' \leftarrow E' \cup \{e\}$ 
10    if  $s < k$  then return  $E'$ 
11     $E' \leftarrow E' - \{e_b\}$ 
12 return  $E'$ 
```

Algorithm 1: `lift_wires(G, k)`

In our heuristic, we begin with the best security we can achieve. This occurs when we lift every edge in $E[G]$; that is, we set E' to $E[G]$ at the start in Line 1. We then progressively try to remove edges from E' , in random order. We do this if not lifting a particular edge e still gives us sufficient security.

That is, we iterate while we still have candidate edges to add back (Line 2). If we do, we identify the “best” edge that we can add back, i.e., the one that gives us the greatest security level if removed from E' . If even the best edge cannot be removed from E' , then we are done (Line 10).

The heuristic does not necessarily yield an optimal set of edges. The reason is that we may greedily remove an edge e_1 from E' in an iteration of the above algorithm. And in later iterations, we may be unable to remove edges e_2 and e_3 . Whereas if we had left e_1 in E' , we may have been able to remove both e_2 and e_3 . Note that removing as many edges from E' is good, because our cost is monotonic in the size of E' (set of edges being lifted).

4.3 Practical Considerations

From a graph-theoretic perspective, the wire lifting procedure outlined provides a set of wires to lift that guarantees a certain security level. However, two practical considerations merit further mention — the scalability of the

proposed techniques to “large” circuits, and the security implication of the attacker having access to the layout of H , as opposed to just the netlist.

Scalability Although the SAT based technique for computing security scales better than the VF2 algorithm, we empirically observe that it times out for circuits with > 1000 gates. To address this issue, we propose a circuit partitioning approach that scales our technique to larger circuits of practical interest. We note that circuit partitioning is, in fact, a commonly used technique to address the scalability issue for a large number of automated circuit design problems.

Algorithm 2 is a simplified description of the partitioning based wire lifting procedure. The function $\text{partition}(G)$ recursively partitions the vertex set of the graph into P mutually exclusive subsets and returns subgraphs $\{G_1, G_2, \dots, G_P\}$ of size such that they can be tractably solved by the SAT based greedy wire lifting procedure. The final set of lifted wires includes the union of all wires that cross partitions, and those returned by P calls to Algorithm 1. We have used this technique to lift wires from circuits with as many as 35000 gates (see Section 5).

```

1  $\{G_1, G_2, \dots, G_P\} \leftarrow \text{partition}(G)$ 
2  $E_R \leftarrow E - \bigcup_{i \in [1, P]} E_i$ 
3 for  $i \in [1, P]$  do
4    $E_R \leftarrow E_R \bigcup \text{lift\_wires}(G_i, s_{req})$ 
5 return  $E_R$ 
```

Algorithm 2: `lift_wires_big(G, s_{req})`

Layout anonymization We have, so far, assumed that the unlifted circuit H is a netlist corresponding to the unlifted gates and wires. However, in practice, the attacker observes a layout corresponding to H , from which he reconstructs the netlist of H . We therefore need to ensure that the layout does not reveal any other information to the attacker besides the reconstructed netlist.

Existing commercial layout tools place gates on the chip surface so as to minimize the average distance, typically measured as the Manhattan distance, between all connected gates in the circuit netlist. Thus, if the complete circuit G is used to place gates, the physical proximity of gates will reveal some information about lifted wires — gates that are closer in the bottom tier are more likely to be connected in the top tier. The attacker can use this information to his advantage.

Instead of using the netlist G to place gates, we instead use the netlist H . Since this netlist does not contain *any* lifted wires, these wires do not have any impact on the resulting placement. Conversely, we expect the physical proximity of gates to reveal no information about hidden

wires in the top tier. In Section 5, we empirically validate this fact. However, anonymizing the layout with respect to the hidden wires does result in increased wire-length between gates, which has an impact on circuit performance. This impact is also quantified in Section 5.

5 Results

We conduct our experimental study using two exemplar benchmarks, the c432 circuit from the ISCAS-85 benchmark suite [10] (a 27-channel bus interrupt controller) with ≈ 200 gates, and a larger DES encryption circuit with ≈ 35000 gates. We use the c432 circuit to investigate security-cost trade-offs obtained from the proposed techniques and use the larger DES circuit for a case study.

All experimental results are obtained using an IBM 0.13μ technology. For 3D integration, bond points are assumed to be spaced at a pitch of $4\mu m$, allowing for one bond-point per $16\mu m^2$. This is consistent with the design rules specified in the Tezzaron $0.13\mu m$ technology kit.

Circuit synthesis was performed using the Berkeley SIS tool [27]. Placement and routing is performed using Cadence Encounter. Finally, we used miniSAT as our SAT solver [29].

5.1 Security-Cost Trade-offs

Figure 6 graphs the security level for the c432 circuit as a function of $E[H]$, the number of unlifted wires in the untrusted tier. $E[H] = 0$ corresponds to a scenario in which *all* wires are lifted, while $E[H] = E[G]$ corresponds to a case in which all wires are in the untrusted tier.

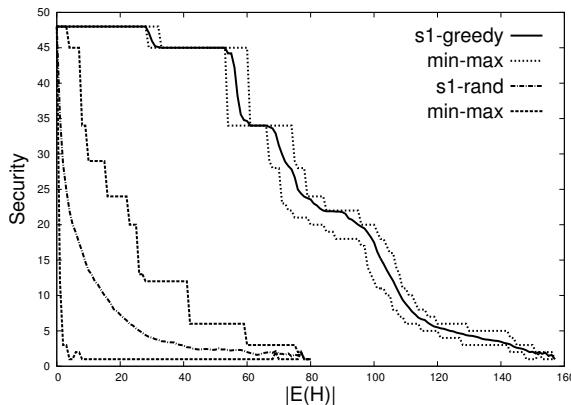


Figure 6: Maximum, average and minimum security levels for the c432 circuit using the proposed greedy wire lifting procedure and random wire lifting.

Proposed Vs. Random Wire Lifting Figure 6 compares the proposed greedy wire lifting technique with a

baseline technique in which wires are lifted at random. In both cases, we show the maximum, average and minimum security achieved by these techniques over all runs.

Observe that greedy wire lifting provides significantly greater security compared to random wire lifting. With 80 unlifted wires, the greedy solution results in a 23-secure circuit, while *all* random trials resulted in 1-secure (equivalently, completely insecure) circuits.

Number of Lifted Edges vs. Security Figure 6 reveals that, for c432, at least 145 of the 303 ($\approx 47\%$) wires must be lifted to get any meaningful degree of security. If any fewer wires are lifted, circuit obfuscation provides no security at all. However, once more than this minimum number of wires is lifted, the security offered increases quite rapidly.

Another observation that merits mention are the plateaus in security level, for example between $E[H] = 30$ and $E[H] = 55$. In other words, in some cases, wires can be retained in the untrusted tier without any degradation in security.

Impact of Layout Anonymization Figure 7 shows three layouts for the c432 circuit. The far left corresponds to the original 1-secure c432 circuit without any wire lifting. The other two layouts correspond to the top and bottom tiers of an 8-secure version of c432 with $\approx 66\%$ lifted wires. Of particular interest is the wire routing in the trusted top tier — because the placement of the corresponding gates in the untrusted bottom tier have been anonymized, the lifted wires are routed seemingly randomly. This is in stark contrast to the wire routing in the original circuit that is far more structured.

Figure 8 shows the histogram of wire lengths for the three layouts shown in Figure 7. Note that, in the original 1-secure circuit, a large majority of wires are short; in other words, connected gates are placed closer together. Wire lengths on the bottom untrusted tier of the 8-secure circuit also skew towards shorter values — however, these wires are already observable to the attacker and he gains no additional information from their lengths. On the other hand, the wire length distribution of the top tier is more evenly spread out. This reflects that fact that the physical proximity of gates in the bottom tier reveals very little information about the lifted wires.

A Chi Square test was performed to determine if the distribution of wirelengths in the top tier is different from one that would be obtained from a random placement of gates. The test does not provide any evidence to reject the null hypothesis ($N = 11$, $\chi^2 = 0.204$ and $p = 0.999$), i.e., it does not reveal any significant difference between the two distributions.

Area, Delay and Power Cost Area, delay (inversely proportional to clock frequency) and power consumption are important metrics of circuit performance. 3D integra-

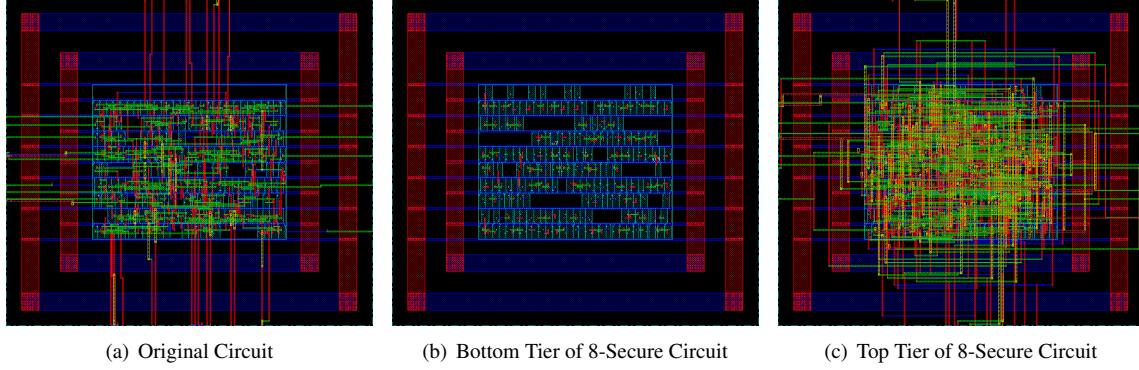


Figure 7: Layout of c432 without any lifting (left), and the bottom (middle) and top (right) tiers of an 8-secure version of c432. Green and red lines correspond to metal wires.

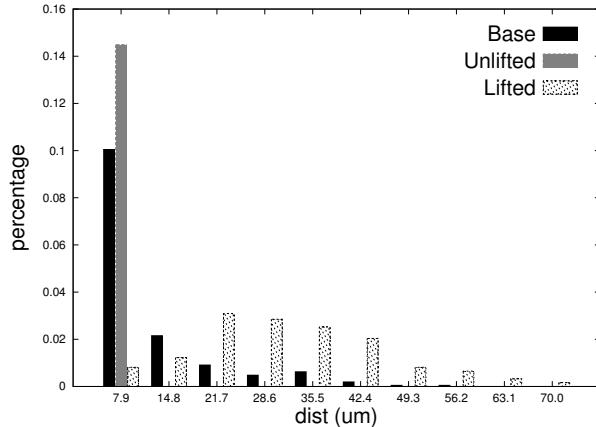


Figure 8: Comparison of the c432 circuit wire lengths for the original 1-secure circuit and the bottom and top tiers of the 8-secure circuit.

tion based circuit obfuscation introduces overheads on all three metrics.

The area of a 3D circuit is determined by the larger of two areas: the area consumed by the standard cells in the bottom tier, and the area consumed by the bond-points required to lift wires to the top tier. The bond-point density is limited by technology (1 bond-point per $16\mu\text{m}^2$ in our case) and therefore more lifted wires correspond to increased area.

Delay and power are strong functions of wire length, as increased wire length results in increased wire capacitance and resistance. Layout anonymization results in increased wire length as we have observed before.

Table 1 shows the area, power and delay for the c432 circuit for different security levels. Compared to the original circuit, the 8-secure circuit has $1.6 \times$ the power consumption, $1.8 \times$ delay, and about $3 \times$ the area.

Choice of Technology Library The technology library determines the type of gates that are allowed in the

circuit netlist. Diverse technology libraries with many different gate types allow for more optimization, but also hurt security. Figure 9 shows the security levels achievable for c432 for five different technology libraries with between three and seven gates.

5.2 Case Study: DES Circuit

We use the DES encryption benchmark circuit to demonstrate that applicability of our techniques, including circuit partitioning based wire lifting, to larger circuits. The DES circuit takes as input a fixed-length string of plaintext and transforms the string into cipher text using 16 rounds of obfuscation, as shown in the block-level circuit diagram in Figure 10.

The original, 1-secure implementation of DES that we synthesized has ≈ 35000 logic gates, which results in an intractable SAT instance. However, using recursive circuit partitioning, we are able to lift wires to obtain a 64-secure implementation. We note that a security level of 16 is obtained in the first few rounds of partitioning by

Table 1: Power, delay, wire length and area analysis for different levels of security on the c432 circuit. 1^* is the base circuit with no wires lifted and 48^* has all of the wires lifted.

Security	Power Ratio	Delay Ratio	Total Wire Length (μm)	Total Area (μm^2)
1^*	1.00	1.00	2739	1621
2	1.54	1.73	6574	4336
4	1.55	1.76	7050	4416
8	1.61	1.82	8084	4976
16	1.62	1.86	8161	5248
24	1.71	1.98	9476	6048
32	1.73	1.99	9836	6368
48^*	1.92	2.14	13058	8144

Table 2: Technology libraries used for the experiment in Figure 9. lib- x corresponds to a library with x different gate types.

Library	$\max(S_1)$	$ V(G) $	$ E(G) $	Gates
lib-3	48	209	303	inv, nor, nand
lib-4	24	181	271	+nand_3
lib-5	13	169	259	+nor_3
lib-6	7	165	252	+nand_4
lib-7	4	159	246	+nor_4

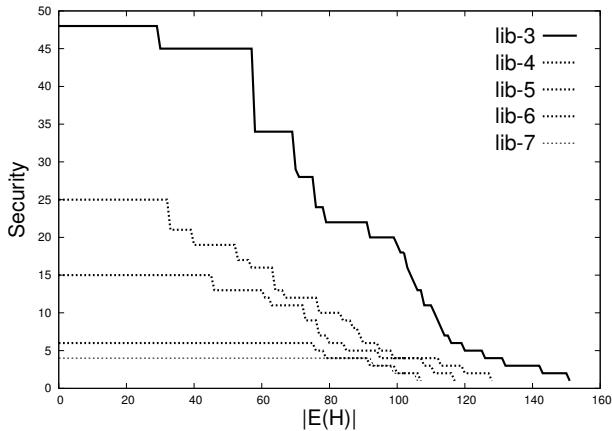


Figure 9: Obtainable security levels for the c432 circuit with different technology libraries.

removing only 13% of the wires, i.e., all wires that lie between successive DES rounds. This is because the circuit description of each DES round is identical — thus, once the wires between the rounds have been removed, each round can be confused for any other round. The final 64-secure implementation has only 30% of the wires unlifted, and consumes $2.38\times$ the area of the original 1-secure circuit.

Attack Scenario Boneh et al. [9] have shown that specific bits in a DES implementation are particularly susceptible to fault attacks. For example, if the attacker is able to insert an attack such that the LSB output of the 14th round is stuck at logic zero, the secret key can be recovered using as few as two messages.

Figure 11 shows how such an attack might be effected using a trigger (we do not address here how this trigger may be activated) and three additional gates in an insecure (or 1-secure) circuit. When the trigger is set, the output is set to zero, but is equal to the correct value when the trigger is at logic zero.

Now, assume that wire lifting is performed to make the circuit 64-secure. Given the set of lifted wires, we note that the LSB of the 14th round is, in fact, 256-secure, i.e., there are 255 other gates in the circuit that are indistinguishable from the LSB of the 14th round.

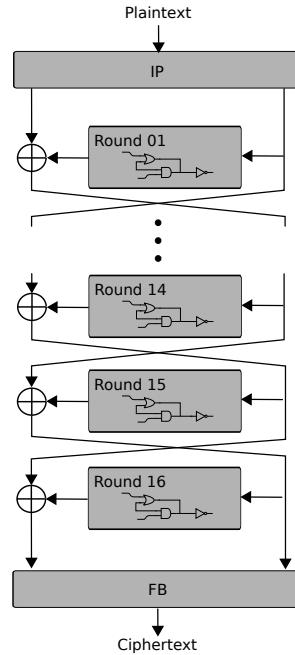


Figure 10: Block diagram of the DES encryption circuit.

The attacker now has two choices. he can either attack one of the 256 options, and only succeed with probability $\frac{1}{256}$, or he can choose to carry out a multiplexed attack on all 256 gates. This is shown in Figure 11. In this attack, the trigger transmits a sequence of 8-bits that identify which of the 256 signals the attacker wants to attack. These 8-bits feed an 8:256 demultiplexer that generates individual triggers for each of the 256 signals that are indistinguishable.

The attacker can now iteratively insert attacks in each gate one at a time and conceivably determine which iteration actually corresponds to the LSB of the 14th round. However, in doing so, the attacker incurs two costs: (i) the modified attack circuit now requires 1280 gates instead of just 3, a $420\times$ overhead; (ii) the attacker would require, in the worst case $255\times$ more messages to recover the key.

5.3 Discussion

We have so far illustrated the quantitative trade-off between cost and security using benchmark circuits. We now discuss this trade-off qualitatively. In particular, we address aspects relating to both the security that 3D IC based split manufacturing can provide and the cost that it incurs in doing so.

From a security standpoint, we note that our notion of k -security is conservative. This is for two reasons. First, we have assumed a strong attack model in which the attacker has access to the original circuit netlist. In prac-

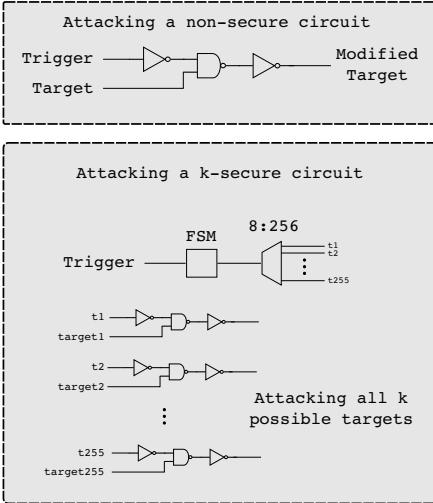


Figure 11: Attack scenarios of 1- and k -secure circuits.

tice, the attacker might only have access to the Boolean functionality of the circuit under attack, but not its gate level implementation. Second, in realistic attack scenarios, the attacker might need to identify more than one gate in the netlist. In both settings k -security serves as a lower bound on the security obtained from 3D IC based split manufacturing.

Furthermore, hardware attacks that are inserted in the foundry are different from other attack scenarios in that they are single shot, and require more effort, risk and expense to carry out. Thus, even relatively low values of k are likely to act as a significant deterrent for the attacker. If the attacker picks one gate to attack at random from the candidate set, he is only successful with probability $\frac{1}{k}$ and receives a payoff which is greater than his cost. However, with probability $\frac{k-1}{k}$, the attacker incurs a (significant) cost and receives no payoff. With $k = 100$ for example, the attacker's payoff must be $> 99\times$ his cost for him to break even (on average). Alternatively, the attacker could try attacking all 100 gates that are candidate mappings for his desired target (as shown in Figure 11), but this would incur a significantly increased risk of detection during post-fabrication testing.

From a cost standpoint, our empirical evaluations suggest a $1.5 \times - 2 \times$ overhead in area, performance (performance is proportional to circuit delay) and power consumption, which is the price we pay for security. Although there is relatively little work in this area, these overheads compare well to those of competing solutions such as field programmable gate arrays (FPGAs). In an FPGA, the desired circuit netlist is programmed on the FPGA *after* fabrication, so an attacker in a foundry receives no information about the circuit the designer wants to implement. However, benchmark studies have

shown that FPGAs are $20\times$, $12\times$ and $4\times$ worse than custom digital ICs in terms of area, power and performance, respectively [20]. In addition, the FPGA itself could be attacked during fabrication in a way that allows an attacker in the field (after fabrication) to recover the circuit that has been programmed on it.

Finally, we note that the proposed technique can be selectively applied to only small, security critical parts of the design. Thus the area, performance and power overheads of split manufacturing would be amortized over the parts of the design that are conventionally implemented. It might also be possible to use split manufacturing in conjunction with other security techniques proposed in the literature such as key-based obfuscation [26, 24]. Key-based obfuscation is only conditionally secure, conditioned on the attacker's computational capabilities. We believe that split manufacturing can be used to further strengthen key-based obfuscation and make it unconditionally secure, although we leave this investigation as future work.

6 Conclusion

In this paper, we have proposed the use of 3D integration circuit technology to enhance the security of digital ICs via circuit obfuscation. The specific 3D technology we exploit allows gates and wires on the bottom tier, and only metal wires on the top. By implementing a subset of wires on the top tier, which is manufactured in a trusted fabrication facility, we obfuscate the identity of gates in the bottom tier, thus deterring malicious attackers.

We introduce a formal notion of security for 3D integration based circuit obfuscation and characterize the complexity of computing security under this notion. We propose practical approaches to determining the security level given a subset of lifted wires, and of identifying a subset of wires to lift to achieve a desired security level. Our experimental results on the c432 and DES benchmark circuits allow us to quantify the power, area and delay costs to achieve different security levels. In addition, we show, using a DES circuit case study, that 3D IC based circuit obfuscation can significantly reduce the ability of an attacker to carry out an effective attack.

Acknowledgements

We thank our shepherd, Cynthia Sturton, and the anonymous reviewers for their feedback and comments. We thank also Vijay Ganesh and Supreeth Achar for their inputs at the initial stages of this research. The work was supported by funding from the NSERC Discovery and Strategic grant programs.

References

- [1] 3D-ICs and Integrated Circuit Security. Tech. rep., Tezzaron Semiconductors, 2008.
- [2] ADEE, S. The hunt for the kill switch. *IEEE Spectrum* 45, 5 (may 2008), 34–39.
- [3] AGRAWAL, D., BAKTIR, S., KARAKOYUNLU, D., ROHATGI, P., AND SUNAR, B. Trojan detection using IC fingerprinting. In *Proceedings of the IEEE Symposium on Security and Privacy* (2007), IEEE, pp. 296–310.
- [4] ALKABANI, Y., AND KOUSHANFAR, F. N-variant IC design: Methodology and applications. In *Proceedings of the 45th annual Design Automation Conference* (2008), ACM, pp. 546–551.
- [5] ARORA, S., AND BARAK, B. *Computational Complexity: A Modern Approach*. Cambridge University Press, 2009.
- [6] BANERJEE, K., SOURI, S. J., KAPUR, P., AND SARASWAT, K. C. 3-D ICs: A novel chip design for improving deep-submicrometer interconnect performance and systems-on-chip integration. *Proceedings of the IEEE* 89, 5 (2001), 602–633.
- [7] BILZOR, M. 3D execution monitor (3D-EM): Using 3D circuits to detect hardware malicious inclusions in general purpose processors. In *Proceedings of the 6th International Conference on Information Warfare and Security* (2011), Academic Conferences Limited, p. 288.
- [8] BOLSENS, I. 2.5D ICs: Just a stepping stone or a long term alternative to 3D? In *Keynote Talk at 3-D Architectures for Semiconductor Integration and Packaging Conference* (2011).
- [9] BONEH, D., DEMILLO, R., AND LIPTON, R. On the importance of checking cryptographic protocols for faults. In *Advances in Cryptology – EUROCRYPT* (1997), Springer, pp. 37–51.
- [10] BRGLEZ, F. Neutral netlist of ten combinational benchmark circuits and a target translator in fortran. In *Special session on ATPG and fault simulation, Proc. IEEE International Symposium on Circuits and Systems, June 1985* (1985), pp. 663–698.
- [11] CHAKRABORTY, R. S., NARASIMHAN, S., AND BHUNIA, S. Hardware trojan: Threats and emerging solutions. In *Proceedings of the IEEE International Workshop on High Level Design Validation and Test (HLDVT)* (2009), IEEE, pp. 166–171.
- [12] CORDELLA, L. P., FOGGIA, P., SANSONE, C., AND VENTO, M. Performance evaluation of the vf graph matching algorithm. In *Proceedings of the International Conference on Image Analysis and Processing* (1999), IEEE, pp. 1172–1177.
- [13] DAVIS, W. R., WILSON, J., MICK, S., XU, J., HUA, H., MINEO, C., SULE, A. M., STEER, M., AND FRANZON, P. D. Demystifying 3D ICs: the pros and cons of going vertical. *Design & Test of Computers, IEEE* 22, 6 (2005), 498–510.
- [14] DENG, Y., AND MALY, W. 2.5 D system integration: a design driven system implementation schema. In *Proceedings of the Asia and South Pacific Design Automation Conference (ASP-DAC)* (2004), IEEE, pp. 450–455.
- [15] FOGGIA, P., SANSONE, C., AND VENTO, M. A performance comparison of five algorithms for graph isomorphism. In *Proceedings of the 3rd IAPR TC-15 Workshop on Graph-based Representations in Pattern Recognition* (2001), pp. 188–199.
- [16] GARY, M., AND JOHNSON, D. Computers and intractability: A guide to the theory of np-completeness, 1979.
- [17] HICKS, M., FINNICUM, M., KING, S. T., MARTIN, M. M., AND SMITH, J. M. Overcoming an untrusted computing base: Detecting and removing malicious hardware automatically. In *Proceedings of the IEEE Symposium on Security and Privacy* (2010), IEEE, pp. 159–172.
- [18] IRVINE, C. E., AND LEVITT, K. Trusted hardware: Can it be trustworthy? In *Proceedings of the 44th Annual Design Automation Conference* (2007), ACM, pp. 1–4.
- [19] KING, S., TUCEK, J., COZZIE, A., GRIER, C., JIANG, W., AND ZHOU, Y. Designing and implementing malicious hardware. In *Proceedings of the 1st USENIX Workshop on Large-scale Exploits and Emergent Threats* (2008), USENIX Association, pp. 1–8.
- [20] KUON, I., AND ROSE, J. Measuring the gap between fpgas and asics. In *Proceedings of the 2006 ACM/SIGDA 14th international Symposium on Field programmable gate arrays* (2006), ACM, pp. 21–30.
- [21] LAU, J. H. TSV interposer: The most cost-effective integrator for 3D IC integration. *Chip Scale Review* (2011), 23–27.
- [22] MICCIANCIO, D., AND GOLDWASSER, S. *Complexity of Lattice Problems: a cryptographic perspective*, vol. 671 of *The Kluwer International Series in Engineering and Computer Science*. Kluwer Academic Publishers, Boston, Massachusetts, Mar. 2002.
- [23] MICROSEMI. Microsemi ProASIC3 FPGA security overview, 2012. Available from www.microsemi.com/documents/.
- [24] RAJENDRAN, J., PINO, Y., SINANOGLU, O., AND KARRI, R. Security analysis of logic obfuscation. In *Proceedings of the 49th Annual Design Automation Conference* (2012), ACM, pp. 83–89.
- [25] RAJENDRAN, J., SINANOGLU, O., AND KARRI, R. Is split manufacturing secure? In *Proceedings of the Design, Automation & Test in Europe Conference & Exhibition (DATE)* (2013), IEEE, pp. 1259–1264.
- [26] ROY, J. A., KOUSHANFAR, F., AND MARKOV, I. L. Epic: Ending piracy of integrated circuits. In *Proceedings of the conference on Design, Automation and Test in Europe* (2008), ACM, pp. 1069–1074.
- [27] SENTOVICH, E. M., SINGH, K. J., MOON, C., SAVOJ, H., BRAYTON, R. K., AND SANGIOVANNI-VINETTELLI, A. Sequential circuit design using synthesis and optimization. In *Proceedings of the IEEE International Conference on Computer Design (ICCD)* (1992), IEEE, pp. 328–333.
- [28] SKOROBOGATOV, S., AND WOODS, C. Breakthrough silicon scanning discovers backdoor in military chip. *Cryptographic Hardware and Embedded Systems—CHES* (2012), 23–40.
- [29] SORENSSON, N., AND EEN, N. Minisat v1. 13-a sat solver with conflict-clause minimization. *SAT 2005* (2005), 53.
- [30] SWEENEY, L. k-anonymity: A model for protecting privacy. *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems* 10, 05 (2002), 557–570.
- [31] TEHRANIPOOR, M., AND KOUSHANFAR, F. A survey of hardware trojan taxonomy and detection. *Design & Test of Computers, IEEE* 27, 1 (2010), 10–25.
- [32] VALAMEHR, J., TIWARI, M., SHERWOOD, T., KASTNER, R., HUFFMIRE, T., IRVINE, C., AND LEVIN, T. Hardware assistance for trustworthy systems through 3-D integration. In *Proceedings of the 26th Annual Computer Security Applications Conference* (2010), ACM, pp. 199–210.
- [33] WAKSMAN, A., AND SETHUMADHAVAN, S. Silencing hardware backdoors. In *Proceedings of the IEEE Symposium on Security and Privacy* (2011), IEEE, pp. 49–63.
- [34] ZHOU, B., AND PEI, J. Preserving privacy in social networks against neighborhood attacks. In *Proceedings of the 24th IEEE International Conference on Data Engineering (ICDE)* (2008), IEEE, pp. 506–515.

A k -SECURITY-DEC is NP-hard

In this section, we provide outlines of the proofs that underlie our assertion in Section 3 that k -SECURITY-DEC is NP-hard under polynomial-time Turing, or Cook, reductions [5]. Such reductions work the following way. Suppose we want to reduce problem A to B. We show that if we have an oracle for B, then $A \in \text{P}$.

Such reductions are unlikely to be as strong as Karp-reductions [5], that are customarily used to show NP-hardness. Indeed, the Karp-reduction is a special case of the Cook-reduction, and some of our reductions below are Karp-reductions. Nevertheless, the existence of a Cook-reduction from a problem that is NP-hard is evidence of intractability [22]. In particular, in the above example, if A reduces to B, then if $B \in \text{P}$, then $A \in \text{P}$.

Recall from Section 3 that k -SECURITY-DEC is the following decision problem. Given as input $\langle G, E', k \rangle$ where $E' \subseteq E[G]$, does lifting the edges in E' give us k -security? We show that k -SECURITY-DEC is NP-hard in three steps. First, we show that SUB-Iso-SELF (defined below) is NP-hard. We then reduce SUB-Iso-SELF to GATE-SUBISO (see Section 3), thereby showing that GATE-SUBISO is NP-hard. Finally, we reduce GATE-SUBISO to k -SECURITY-DEC.

All graphs we consider are directed, acyclic (DAGs). Thus, all subisomorphisms we consider are for the special case that the graphs are DAGs. It turns out that the subgraph isomorphism problem is NP-hard for even the restricted case, SUB-Iso-9, below.

Definition 6 (SUB-Iso-9). *SUB-Iso-9 is the following special case of the subgraph isomorphism problem. Given as input $\langle G, H \rangle$ where G is a DAG and H is a directed tree, SUB-Iso-9 is the problem of determining whether there exists a subgraph of G that is isomorphic to H.*

SUB-Iso-9 is known to be NP-hard [16].

Definition 7 (SUB-Iso-SELF). *Given as input $\langle G, H \rangle$ such that G is a DAG and H is obtained from G by removing the edges in a set $E' \subseteq E[G]$, SUB-Iso-SELF is the problem of determining whether there exists a subgraph isomorphism ϕ from G to H that is not the identity mapping.*

Theorem 2. *SUB-Iso-SELF $\in \text{NP-hard}$.*

Note that the above theorem is not qualified that it is under Cook-reductions. This is because we have a Karp-reduction from SUB-Iso-9 to SUB-Iso-SELF. The reduction proceeds in several steps. First, we show that SUB-Iso-9 restricted to the case that $|V[G]| = |V[H]|$ leaves the problem NP-hard. We do this by first observing that for any prospective instance $\langle G, H \rangle$ of SUB-Iso-9, we can assume that $|V[H]| \leq |V[G]|$. We simply add $|V[G]| - |V[H]|$ vertices to H.

Then, we show that if we add the further restriction that G and H are strongly connected (i.e., every vertex reachable from every other vertex), the problem is still NP-hard. For this reduction, we first check whether the two graphs are strong connected. If not, we introduce a new vertex of a colour distinct from every vertex in the graphs which has an edge to and from every other vertex.

We then show that SUB-Iso-SELF is NP-hard as follows. We introduce into G an exact copy of H that is disjoint from G . We call this new graph G' , and the subgraph of G' that is the copy of H , H' . We further restrict H and H' to not have any automorphisms. To achieve this, we introduce $|V[H]|$ vertices each of a distinct colour, associated with each $u \in V[H]$. Call this vertex v_u . We connect u and v_u with an edge. We do the same in H' . We also add a subgraph G'' to H which has $|V[G]|$ vertices and no edges. (This guarantees that the new subgraph is subgraph isomorphic to G .) We call this new graph H'' .

We use the same technique as above of adding coloured vertices to ensure that G (within G') and G'' in H'' are not automorphic. Finally, we connect every new vertex added above to the vertices of G , to every original vertex of H' , and every new vertex added to H' to every original vertex of G . We do the same in H'' . We now are able to show that $\langle G, H \rangle$ is a true instance of SUB-Iso-9 if and only if $\langle G', H'' \rangle$ is an instance of SUB-Iso-SELF.

Theorem 3. *GATE-SUBISO $\in \text{NP-hard}$ under Cook-reductions.*

Recall that GATE-SUBISO comprises those instances $\langle G, E', u, v \rangle$, where, if H is produced from G by removing the edges in E' , and u, v are distinct vertices in G (and therefore H), there is a subgraph isomorphism from G to H that maps u to v . In our reduction, we assume that we have an oracle for GATE-SUBISO. We simply invoke it for every pair of vertices $u, v \in G$. If any of them is true, then we know that $\langle G, H \rangle$ is a true instance of SUB-Iso-SELF. Otherwise, it is not.

Theorem 4. *k -SECURITY-DEC $\in \text{NP-hard}$ under Cook-reductions.*

We Karp-reduce GATE-SUBISO to k -SECURITY-DEC. Let $\langle G, E', k \rangle$ be a prospective instance of k -SECURITY-DEC, and H is produced from G by removing the edges in E' . We first ensure that every vertex other than u is 2-secure. We do this by introduce a new vertex for every vertex other than u that has exactly the same connectivity. Then, in G , we introduce a new vertex of a completely new colour and attach it to u and v . We include the edge between v and this new vertex in E' . Call the G so modified G'' , and the new set of edges E'' . We can now show that $\langle G'', E'', 2 \rangle$ is a true instance of k -SECURITY-DEC if and only if $\langle G, E', u, v \rangle$ is a true instance of GATE-SUBISO.

KI-Mon: A Hardware-assisted Event-triggered Monitoring Platform for Mutable Kernel Object

Hojoon Lee¹, Hyungon Moon², Daehee Jang¹, Kihwan Kim¹, Jihoon Lee², Yunheung Paek², and Brent ByungHoon Kang^{*1}

¹Graduate School of Information Security, KAIST

{hojoon.lee,daehee87,abc,brentkang}@kaist.ac.kr

²Department of Electrical and Computer Engineering, Seoul National University
{hgmoon,jhlee}@sor.snu.ac.kr and ypaek@snu.ac.kr

Abstract

Kernel rootkits undermine the integrity of system by manipulating its operating system kernel. External hardware-based monitors can serve as a root of trust that is resilient to rootkit attacks. The existing external hardware-based approaches lack an event-triggered verification scheme for mutable kernel objects. To address the issue, we present KI-Mon, a hardware-based platform for event-triggered kernel integrity monitor. A refined form of bus traffic monitoring efficiently verifies the update values of the objects, and callback verification routines can be programmed and executed for a designated event space. We have built a KI-Mon prototype to demonstrate the efficacy of KI-Mon’s event-triggered mechanism in terms of performance overhead for the monitored host system and the processor usage of the KI-Mon processor.

1 Introduction

Kernel rootkits are a special class of malware that compromise an OS kernel; they pose severe threat to the monitored host system as they can hide their attack traces to stay undetected while persisting in their malicious activities. Since rootkits place themselves in the lowest kernel layer that has the highest privilege level in a system, they can trick and compromise any host-based intrusion detection system running on the above layer, making the detection system ineffective. Many researchers have made active efforts to address rootkit attacks by providing a safe execution environment where kernel integrity monitors can run with the root of trust established below the kernel OS layer. These efforts can be categorized into two types of approaches: *Virtual Machine Monitor (VMM)* based [19, 34, 31, 28, 37], and hardware-based [29, 26, 10, 40]. Both VMM and hardware platforms are used as safe execution environments

for integrity monitoring, as a root of trust under the OS kernel. However, since they are implemented in software, VMMs also have to suffer from software vulnerabilities. As the discoveries of VMM vulnerability continue [5, 4, 2, 3], more attacks can subvert the VMM layer underneath the OS kernel [32].

External hardware-based approaches [29, 26] attempt to utilize the underlying hardware as another root of trust for integrity monitors, seeking physical isolation from the monitored system. By deploying the integrity monitor on an external hardware device, the monitoring can persist even when the entire OS on the monitored host system is compromised. One of the earlier external hardware-based monitors, Copilot [29] presented a *snapshot-based* kernel integrity monitor implemented as a peripheral device. It utilized periodically collected snapshots of memory contents of the kernel static region to perform a hash value comparison with a known good value. In such approaches, increasing the frequency of snapshot to monitor all the modifications of a rapidly changing target leads to significant performance overhead [26]. Therefore, we believe that event-triggered verification is needed for monitoring mutable kernel objects.

Event-triggered monitoring techniques are relatively common in VMM-based approaches. Hypercall interception, page fault interception, exception handling interception, and other techniques using *VM Exits* in *Hardware Virtual Machines (HVM)* [37, 17, 34, 38] are well-known examples. By inserting additional codes into the handlers of those events, a preset verifier routine can be executed upon the occurrence of the events. However, in contrast to VMM-based approaches, the hardware-based event-triggered approaches are still in their infancy.

The first external hardware-based event-triggered monitoring scheme was introduced in Vigilare [26]. Vigilare is an *immutable* region snooper that is limited to the detection of the existence of any write traffic, destined for the monitored memory region on the host bus. In

*corresponding author

other words, an event in Vigilare only signifies an occurrence of a memory modification while it does not provide any ability to extract the data value in the write traffic for the invariant verification, nor does it provide any callback mechanism that could further verify the event for consistent modification with respect to other related data objects. Vigilare’s rudimentary scheme has been sufficient for the immutable regions. However, it is incapable of monitoring *mutable* kernel objects.

The contents of mutable objects in dynamic regions, or dynamic data structures, are frequently modified by the operating system kernel. Such a characteristic introduces complexities in monitoring the mutable kernel objects. Since the modifications made to the mutable objects could be legitimate changes, resulting from the normal operations of a kernel, simply detecting the occurrence of modification to these structures does not provide decisive evidence in determining whether the modifications are malicious or benign. In addition, there are cases in which verifying the update value against a known good value is not sufficient for integrity verification. Consider the example of a linked list manipulation attack, where the adversary removes an entry from a linked list to hide the entry. Inspecting the linked list will reveal that the entry has been removed. However, from this observation alone, we cannot determine if the entry was removed by an adversary or legitimately removed by the kernel. In these cases, additional *semantic verification* to check the consistent modification of other related kernel data structures is required to confirm the legitimacy of these changes.

We propose an external hardware-based Kernel Integrity Monitoring platform, called *KI-Mon*. To explore possibilities of monitoring mutable kernel objects with an event-triggered mechanism, KI-Mon presents architectural foundations of hardware-assisted event-triggered detection and verification mechanism. KI-Mon is capable of generating an event which reports the address and value pair of memory modification, occurred on the monitored object. Event generation is refined with a support for whitelist-based filtering to eliminate unnecessary software involvement in value verification. KI-Mon also allows an event-triggered callback verification routine to be programmed and executed for a designated event space. In addition, we developed the KI-Mon API to ensure the programmability of the platform, which supports development of monitoring rules. Example monitoring rules were developed and tested against attacks from real-world rootkits to confirm the effectiveness of the platform. Also, our evaluation shows the efficacy of event-triggered monitoring in terms of the performance overhead to the monitored system using benchmarking tools.

We built the KI-Mon prototype on a FPGA-based de-

velopment board, and evaluated the effectiveness of KI-Mon with experiments. We used the STREAMBENCH and RAMSPEED benchmarking tools for measuring the performance overhead on the monitored system’s memory bandwidth. The results showed that the snapshot-only monitor incurred a significant overhead to the monitored host system’s memory bandwidth while KI-Mon consumed significantly less CPU cycles due to its event-triggered mechanism. This is because KI-Mon detects memory modifications at hardware level using VTMU which features an event filtering mechanism to eliminate CPU cycles consumed by snapshot-based polling by 6 orders of magnitude.

2 KI-Mon Design

KI-Mon is an external hardware-based Kernel Integrity Monitor that adapts an event-triggered mechanism to enable monitoring of dynamic-content data structures. To achieve the desired functionality, we designed and implemented a prototype of a platform that includes both hardware and software components. The design objectives for KI-Mon are summarized as the following:

O1. Safe Execution Environment: The most fundamental requirement for any kernel integrity monitor is a safe execution environment. That is, a kernel integrity monitor should be designed to be resilient to any type of interference from the compromised monitored system.

O2. Event-triggered Monitoring: For an external monitor to trace mutable kernel objects, it should be able to identify any modification as an event that is comprised of an address and value pair. As previously mentioned, the update value is essential for verification of the legitimacy of the modification. In addition, there needs to be a mechanism that allows a semantic verification routine to be executed when the value of an event alone cannot serve as proof that the modification is malicious. Furthermore, KI-Mon deviates from periodic state capturing techniques such as memory snapshots, implementing a hardware platform that focuses on events, rather than states. We further define the desiderata for an event-triggered monitoring mechanism as below, in O2.1 to O2.4.

O2.1 Refined event generation: For an external monitor to trace mutable kernel objects, it should be able to identify any modification as an event, comprised of an address and a value pair. Furthermore, a refined event can be generated from raw events by suppressing commonly occurring benign updates at the snooping hardware module, so that the verifier can be engaged only when it is necessary.

O2.2 Event-triggered semantic verification: As previously mentioned, the value is essential for

verification of the legitimacy of the modification. In addition, there needs to be a mechanism that allows a semantic verification routine to be executed when the value of an event alone cannot serve as a proof that the modification is malicious. The routine should reference other related kernel objects in order to verify the semantic consistency.

O2.3 Minimal overhead on monitored system:

KI-Mon deviates from periodic state capturing techniques such as memory snapshots, implementing a hardware platform that focuses on events, rather than states. An event-triggered mechanism should also minimize performance overhead inflicted on the monitored system during its operation.

O2.4 Efficient monitoring processor usage:

An event-triggered scheme is expected to minimize the workload, imposed on the monitoring processor. This minimization can be beneficial when the amount of monitored data is larger and the hardware cost of the monitoring processor needs to be limited.

O3. Programmability: The operating systems maintain a large number of various dynamic data structures during run-time, and the format and usage of these data structures vary across different operating systems. Moreover, kernel updates to the operating systems often change the behavior of kernel operations that are related to the data structures or the format of the data structures. For this reason, KI-Mon needs to be highly programmable, in order to guarantee a certain degree of portability across different operating system versions and to support development of new monitoring algorithms. The details of the KI-Mon design that address the above design objectives will be explained in the rest of this section. Design objective *O1* is achieved using KI-Mon’s hardware platform by design. We developed KI-Mon API to provide programmability to KI-Mon. This programmability satisfies design objective *O3*. Design objective *O2.1* is addressed by KI-Mon’s HAW mechanism; *O2.2* is achieved by the event-triggered Semantic Verification mechanism. *O2.3* and *O2.4* will be further evaluated in Section 4.

2.1 Safe Execution Environment

The KI-Mon hardware platform is a complete microprocessor-based system like those in existing external independent processor approaches [26]. While KI-Mon operates independently from the monitored host system, it is capable of monitoring host memory modifications with a bus traffic monitoring module called *Value Table Management Unit (VTMU)* and a

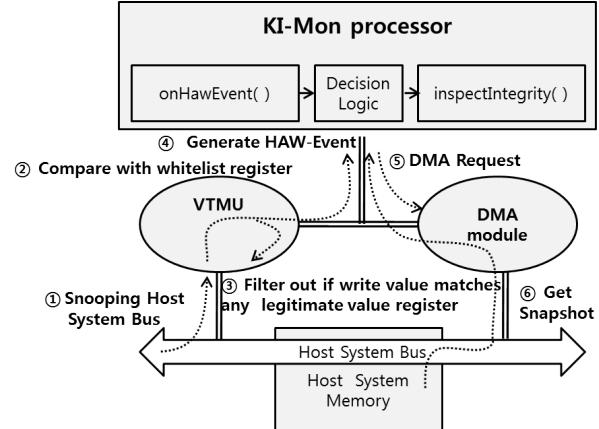


Figure 1: KI-Mon Monitoring Mechanism

Direct Memory Access (DMA) Module for the monitored system. The in-depth capabilities of VTMU and the use of DMA will be further discussed in the rest of this section, but it should be noted that their operations do not involve the monitored system’s processor, nor any other components on the monitored system. This is made possible by the shared bus architecture, which enables KI-Mon to inspect the monitored system. On the other hand, the monitored system has no physical connection to KI-Mon through which it could interact with. In fact, the monitored system is not aware of the existence of KI-Mon. Hence, KI-Mon ensures that its monitoring activities are safe even when the monitored host system is compromised by a rootkit. In this way, KI-Mon achieves its first design objective *O1: Safe Execution Environment*.

2.2 Event-triggered Monitoring

KI-Mon incorporates its hardware and software platform. The hardware platform generates events when modifications occur in the monitored regions. The software platform verifies events as shown in Figure 1. The explanation of this mechanism will start from the capturing of host bus traffic in the hardware platform. It will then explore how these captured instances of traffic are passed up to the software platform for the further verification.

2.2.1 Refined Event Generation

VTMU is the core component that monitors the host memory bus traffic to generate events. Its operation can be divided into three stages: bus traffic snooping, address filtering, and value filtering. The bus of the monitored system is fed into VTMU, and VTMU extracts only write signals from the stream of the host’s memory

I/O traffic. As the collected write signals pass through the address filter, all signals except the ones corresponding to the monitored region are discarded. Finally, the signals are once again filtered in the comparator units. The signals are compared against the preloaded values in the whitelist registers. The signals with the address and value pair, that survived the two-stage filtering, are reported to the software platform, running on the KI-Mon processor. We call this mechanism *hardware-assisted whitelisting (HAW)*; the reports, sent to the software platform, are called *HAW-Events*.

Also, it should be noted that the VTMU is a highly configurable hardware component, and our software platform can readily adjust the monitored regions and the whitelisted values. For instance, the whitelist registers can be configured to be inactive, so that all write signals to the monitored regions generate HAW-Events. In addition to VTMU, the DMA module is also implemented and included in the KI-Mon hardware platform. The module steals memory cycles of host processor to fetch the contents from the host memory on an on-demand basis. When the software platform requests the contents of a certain region of the host memory, the DMA module takes a snapshot of the region and provides it to the kernel integrity monitor. In summary, VTMU is capable of monitoring host memory without constantly polling host memory. It can also reduce the generation of benign events by using a whitelist.

2.2.2 KI-Veri and MonitoringRules

Kernel Integrity Verifier (KI-Veri) is the main component in the software platform, enabling the event-triggered monitoring mechanism. It interfaces with *MonitoringRules*, which are high-level objects implemented on top of the KI-Mon API. Each MonitoringRule defines the target regions to be monitored by VTMU, and such regions are called *critical regions*. VTMU generates HAW-Events when the contents of these regions are modified. For this reason, the regions should be chosen prudently so that a modification of the regions will serve as an effective trigger to the monitoring mechanism. Critical regions and their whitelists are stored in VTMU upon the registration of MonitoringRules. A MonitoringRule is also required to have predetermined actions such as an *HAW-Event Handler* and an *Integrity Verifier*, to be executed when HAW-Events occur in the critical regions. These actions are fetched and executed by KI-Veri. HAW-Event Handlers verify HAW-Events in order to invoke other actions, such as Integrity Verifiers, as needed.

In summary, VTMU monitors critical regions registered by MonitoringRules in KI-Veri, and generates HAW-Events when a write signal that does not match any

of the values in the whitelist registers appears in critical regions. Upon receiving a HAW-Event, KI-Veri executes the HAW-event handler of the MonitoringRule, that is responsible for the HAW-Event. Then, the HAW-event handler triggers an action that corresponds to the pair of the address and the update value.

2.2.3 Detection Methodology of *MonitoringRule* Templates

The main focus of the current implementation of KI-Mon is to propose an event-triggered monitoring scheme for mutable kernel objects. Rootkit attacks on mutable kernel objects can be classified into two categories: *control flow components* and *data components* [19]. Control-flow components are usually function pointers that store the addresses of kernel functions. Since such control flow components are referenced to execute the functions located at the addresses, rootkits often place *hooks* on such components to inject their routine into the control flow.

Many data components or non-control-flow components, store critical pieces of information that reflect the current state of the kernel. Critical data components such as lists of processes, kernel modules, and network connections lists can be subverted by rootkits so that the traces of rootkits are hidden. KI-Mon deploys two types of MonitoringRule templates in its prototype for monitoring of control flow and data components: *Hardware-Assisted Whitelisting (HAW)-based Verification* for control flow components and *Callback-based Semantic Verification* for data components.

Hardware-Assisted Whitelisting (HAW)-based Verification: As we discussed in the previous section, update value verification can serve as an indication of malicious manipulations in some cases; semantic verification is otherwise imperative. Recall that a semantic verification references other semantically related kernel objects to find semantic inconsistencies. We observe that value verification is particularly effective against attacks on control flow components. All control flow components should point to the functions in the kernel code section, or functions in the known kernel drivers loaded via loadable kernel modules. More specifically, many control flow components in kernel dynamic data structures always point to one possible landing site. We define such property as the value set invariant of a kernel object. We take advantage of this property in modeling the monitoring scheme for control flow components. HAW-based Verification is a MonitoringRule, where the address of the control flow component is set as a critical region and its possible landing sites as a whitelist in VTMU. HAW-events, generated from this type of MonitoringRule, are simply considered malicious.

Callback-based Semantic Verification: Callback-based Semantic Verification is a type of MonitoringRule, which is designed to serve as a template for monitoring kernel data components. The monitoring scheme for control flow components is not suitable for monitoring of modifications on data components that require semantic verification because the processes of identifying memory modifications and their values are inadequate for detecting manipulation attacks on semantic information. The HAW-Event handler can invoke the Integrity Verifier for further inspection, which involves acquisition of semantically related data structures. This type of Integrity checking is called the enforcement of *semantic invariants* [12]. Note that the HAW-Event handler can be programmed to call functions other than Integrity Verifiers. This feature can be used to update the information on the monitored data structure. For example, detection of a newly inserted entry in a linked list can be programmed and invoked by the HAW-Event handler.

2.3 KI-Mon API for Programmability

As previously mentioned, the MonitoringRules that operate in KI-Mon are built with the KI-Mon API. The KI-Mon API, as shown in Figure 4, includes high-level software stacks and low-level drivers for the hardware platform, to enable convenient and rapid development of kernel integrity monitoring rules. KI-Mon API is developed so that writing new MonitoringRules, based on our detection methodology, become convenient. It is even possible to create entirely new algorithms. Thus, KI-Mon API corresponds to our third design objective: *O3:Programmability*. A more detailed explanation of the internals of the API will be given in the following section.

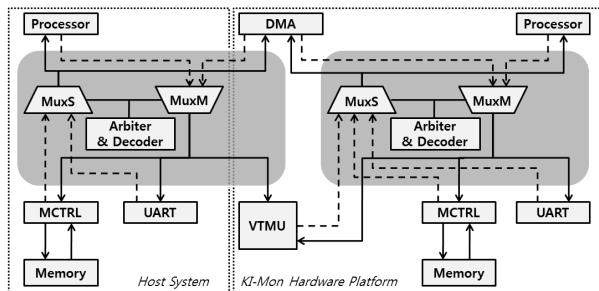


Figure 2: KI-Mon Hardware Platform. (Gray box shows bus architecture)

3 Prototype Implementation

3.1 KI-Mon Hardware Platform Prototype

The KI-Mon platform, including the monitored host system, is implemented as a *System on a Chip (SoC)* on an FPGA-based prototyping system for rapid prototyping. Figure 2 shows the overall structure of our SoC implementation. The monitored system, running on a Leon3 [7] processor, configured to operate at 50 MHz, *Snapgear Linux* with a kernel version of 2.6.21.1 [18], provided from the provider of the Leon3 processor, was used as the operating system for the monitored system. Both KI-Mon and the host processor use an S-compatible shared bus [9] as an interconnection network. As can be seen from Figure 2, the KI-Mon hardware platform has been built on the same architecture base as that of the host processor system, being augmented with new features with event-triggered monitoring capabilities.

Other than VTMU, the hardware platform also includes a DMA module and a hash accelerator to support snapshot-related features. As previously discussed, the DMA module takes snapshots of the monitored system's memory and stores them in KI-Mon's private memory. The DMA module has two master interfaces and one slave interface. One of the two master interfaces is connected to the monitored system's bus. The other is connected to KI-Mon's bus. With the master interfaces, the module is capable of reading any regions of the monitored system's memory; it can then copy the contents to the designated space in KI-Mon. The slave interface, which is connected to the KI-Mon bus, is used for KI-Veri in the software platform to make requests for snapshots. The hash accelerator generates SHA-1 hash values from given memory contents. The hash accelerator has both slave and master interfaces to the KI-Mon bus. The slave interface is used to receive requests for hashing a certain region and returning the calculated hash value to KI-Mon, and the master interface is used to read the memory regions to be hashed.

VTMU is a core component of the KI-Mon hardware platform. It generates HAW-events by snooping the host bus traffic for modifications, filtering the traffic based on the addresses and the values being written. By doing so, traffic with addresses that do not belong to the monitored regions is ignored, as are benign modifications in which a whitelisted value is written. As mentioned in the previous section, VTMU registers are configurable via the driver we implemented. The addresses of the monitored regions and corresponding whitelists can be passed to VTMU at any time, so the operation of VTMU can be controlled even during runtime. In addition, the monitoring capacity, such as the total number of regions monitored simultaneously or the length of the whitelist, can

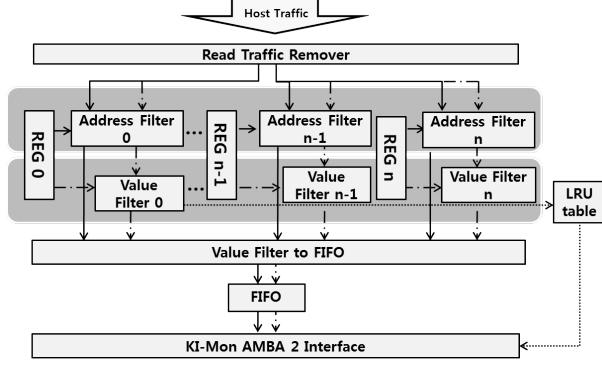


Figure 3: VTMU Internal Architecture Overview

be adjusted easily. More specifically, one can increase the number of registers or simply place multiple VTMU units in KI-Mon.

The operation of VTMU consists of three stages: bus traffic snooping, address filtering, and value filtering. The first stage of VTMU operations, bus traffic snooping, is implemented based on a shared bus architecture that conforms to the AMBA 2 protocol. Modules attached to the AMBA 2 AHB protocol bus are categorized into masters and slaves. Masters are active modules that access slave modules as needed, whereas slaves are passive modules that respond to the requests of masters. In our implementation, the processor and DMA module are master modules, and the memory controller (MCTRL), serial port (UART), and VTMU are slave modules. The gray box in Figure 2 shows the bus architecture of the monitored system and the KI-Mon hardware platform. Also, the connections of VTMU on the KI-Mon hardware platform are shown. *MuxM* is a multiplexer unit that passes only one master's traffic to a slave. *MuxM* is controlled by hardware logics called *arbiters* and *decoders*. These modules decide which master utilizes the bus at each clock cycle. That is, only one master can utilize the bus at each clock cycle, and all slaves receive the same traffic from the master at each time. With this hardware principle, we designed the bus traffic snooping stage of VTMU to acquire all memory traffic from the monitored system by duplicating the output signals of *MuxM*. The type of the traffic – whether the traffic indicates a write operation or not – is checked with a simple comparator, so that this stage only passes write-traffic to the address filtering stage. The value filtering process is the last stage of VTMU operations. The value filter is an extension of the address filter in terms of the hardware structure. While the address filter has 8 sets of 2 address registers that store the starting and ending addresses of the monitored regions, the value filter has 8 sets of 6 registers. This is because the 6 whitelist values correspond to each of the 8 monitored regions.

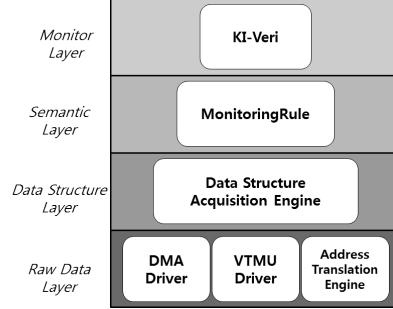


Figure 4: KI-Mon API

The FIFO buffer stores the output of the filter until that output is fetched by KI-Mon. Although a larger FIFO would be more robust against bursty traffic, a buffer length of 16 was sufficient for our current prototype and experiment settings. The tag registers keep track of whitelist values that match the occurred traffic. The register is set once traffic hits the registers. With this feature, KI-Mon can replace the values in the whitelist registers as needed with the recently used values. For instance, KI-Mon keeps the recently used values in the whitelist registers and replaces those that have not recently been used. The traffic that has passed through the second stage is fed into the value filters. The value of the traffic, or the value being written to the monitored regions, is compared with the values stored in the whitelist registers. If the traffic matches—meaning that this traffic indicates benign changes—it is discarded; if the traffic does not match, such bus traffic is stored in the FIFO buffer unit. Finally, a HAW event is generated and triggers KI-Veri to acquire the address and value pair, generated from the FIFO buffer. The overall view of VTMU's internal structure is illustrated in Figure 3.

3.2 KI-Mon Software Platform Prototype

KI-Veri, which is the main operator of the software platform, is positioned at the *monitor layer* to coordinate the monitoring rules, the API, and interactions with the hardware components. The *semantic layer* implements MonitoringRule, which defines the monitored regions, whitelists, and corresponding actions. The *data structure layer* adds abstractions to access the monitored system's raw data contents, so that the raw data is parsed into appropriate types and structures. Lastly, the *raw data layer* contains the low-level drivers for the hardware platform, which directly interacts with the monitored host system's memory interface. KI-Mon API consists of 913 lines of C code.

Upon the occurrence of an event, KI-Veri searches the VTMU registers to find the MonitoringRule instance for which the registers are reserved. Then, KI-Veri executes

the HAW-event handler of the MonitoringRule instance to verify which action needs to be invoked for the HAW-event.

As shown in Figure 5, KI-Veri retrieves the pointer to the MonitoringRule that is responsible for the HAW-event. The HAW-event handler of this MonitoringRule determines the action that needs to be taken for the given *addr* and *value* pair. The pair contains the address, where the modification has occurred and the value of the modification.

The class MonitoringRule is implemented as an object-oriented C structure. It is designed to serve as a template for writing a kernel integrity monitoring rule on KI-Mon’s event-triggered mechanism. The class includes critical regions, corresponding whitelists, an initializer function, and the action functions. Figure 6 is a pseudo code definition of the class MonitoringRule.

The *CriticalRegion* data structure defines the starting and ending address of the monitored region as well as the whitelist for the region. The *initMonitoringRule* can contain initialization procedures such as acquiring of the addresses of the monitored data structures, which addresses will be stored in the *criticalRegion* variable. The *onHawEvent* defines the action to be taken upon the arrival of HAW-events from the hardware layer. If the MonitoringRule was of a HAW-based Verification template – all write attempts to the monitored regions are considered malicious if they are not in the whitelist – the function can simply declare that an attack was detected. For the MonitoringRules, which were written for a Callback-based Semantic Verification template, *onHawEvent* can call *inspectIntegrity* passing arguments as needed. Then, the *inspectIntegrity* function verifies the modification reported via HAW-event with memory snapshots collected from the monitored system. Similarly, *traceDataStructures* can be called if *onHawEvent* sees that the HAW-event generated signifies change in the location or size of the monitored structure.

```
onHawEventFromVTMU(addr,value) {
    monitoringRule = getMonitoringRuleFor(addr);
    requiredAction = \
    monitoringRule->HawEventHandler(addr,value);
    if(requiredAction == INSPECT_NEEDED) {
        monitoringRule->inspectIntegrity(argArray);
    }
    else if(requiredAction == RAISE_ALERT) {
        monitoringRule->traceDataStructures(argArray);
    }
    else {
        //Other requiredAction can be here
    }
}
```

Figure 5: KI-Veri’s Main Routine

```
typedef struct MonitoringRuleType {
    CriticalRegion criticalRegion;
    void initMonitoringRule();
    int (*onHawEvent)(addr,value);
    int (*inspectIntegrity)(argArray);
    int (*traceDataStructures)();
}MonitoringRule;
```

Figure 6: Class MonitoringRule

The functions and macros defined in the data structure layer can be used as building blocks for implementing the action functions in MonitoringRules. The *Data Structure Acquisition Engine* is the actual implementation of the layer. Memory snapshots extracted from the monitored system’s memory are raw memory contents. Since KI-Mon or any other external hardware monitor does not have OS-managed metadata of the monitored data structures, additional parsing and constructing of a meaningful data structure out of the raw data is essential.

The Raw Data Layer consists of the low-level hardware drivers that provide core functionalities for the upper layers. The *VTMU Driver* manages the memory value verification units, which count up to 16 in our current implementation. Each unit consists of 6 registers: the first two registers store the starting and ending addresses of the interval to be monitored. The rest of the registers store the whitelisted values referenced by the comparators. It should be noted that the VTMU driver only engages in the configuration of the hardware. That means, the memory bus traffic monitoring can be effortlessly done in the hardware layer thus it is not necessary for the driver to be running during the monitoring. VTMU notifies the software stack of an event when a write event to the monitored regions is detected. The *DMA Driver* makes DMA requests to the monitored system memory to acquire memory snapshots. The functionality of the driver is rather straightforward: given an address and size of a snapshot, it fetches the region from the monitored system memory. The aforementioned Data Structure Acquisition Engine adds usability to the snapshot-taking capability of the DMA module. The *Address Translation Engine* translates the virtual addresses of the monitored system into a physical address. The Address Translation Engine implements a virtual to physical address translation process of the monitored system in KI-Mon. The Address Translation Engine performs page table walks by fetching the corresponding entries of the page table in the monitored system’s memory.

3.3 KI-Mon MonitoringRule Examples

In order to illustrate the monitoring capabilities of KI-Mon and the programmability of its API, we developed two MonitoringRule examples against the two real-world rootkit attacks, ported to operate on the Linux kernel running on our prototype, where the VFS hooking attack from *Adore-NG* is an example of an attack on kernel control-flow components and the *LKM hiding attack* from *EnyeLKM* is a kernel data component manipulation attack.

The two examples that we choose, represent real-world rootkit attacks on control-flow and data components. We analyzed the open source real-world rootkits [39, 16, 27, 33, 1] and referenced works that analyzed the behaviors of well-known rootkits [42, 35, 22, 19]. Table 1 summarizes some of the attacks on kernel mutable objects identified from the rootkits. These well-known rootkits manipulate both the control-flow and the data components. It is noticeable that the VFS hooking attack and its variants, which manipulates the control-flow components of Linux Virtual File System including the *proc* file system (VFS) [24, 14], are popular for being deployed to hide files, processes, and network connections. Also, the LKM hiding was a common behavior among the analyzed rootkits. The attack manipulates a *module->list* structure to hide an entry in the *Loadable Kernel Module (LKM)* list. The rootkits utilize LKMs as a means to inject kernel-level code into the victimized kernel, and they launch the LKM hiding attack once their malicious code is loaded in the kernel memory space.

One of the two MonitoringRules we implemented is built using the HAW-based verification template to detect the VFS hooking attack. The other MonitoringRule is built using the Callback-based Semantic Verification template to demonstrate the detection of the LKM hiding attack. The rest of this subsection provides the two attack examples and our MonitoringRules in detail.

VFS Hooking Attack: The Virtual File System (VFS) [24, 14] provides an abstraction to accessing file systems in the Linux kernel; all file access is made through VFS in the modern Linux kernel. The kernel maintains a unique *inode* data structure for each file, which includes a *fops* data structure that stores pointers to the VFS operation functions such as open, close, read, write, and so forth. Various critical information about the kernel, such as the network connections and the system logs, are stored in the form of a file and are queried via the VFS interface. Rootkits are capable of directly manipulating the functionalities of VFS. More specifically, they can hook the VFS operation functions of the *fops* data structure in a file to manipulate the contents read from it. Examples of malicious exploitation of VFS include hiding network connections or running processes,

Table 1: Examples of Attacks on Mutable Kernel Objects

Rootkit Name	Target Object Type	Object Type
Adore-NG 0.41	inode->i_ops	Control-flow component
	task_struct-> {flags,uid,...}	Data component
	module->list	Data component
Knark 2.4.3	proc_dir_entry	Control-flow component
	task_struct-> flags	Data component
	module->list	Data component
Kis 0.9	proc_dir_entry	Control-flow component
	tcp4_seq_fops	Control-flow component
	module->list	Data component
EnyeLKM 1.3	module->list	Data component

associated with the attacker. In Linux, */proc* [24] contains important files that maintain system information. By hooking the VFS data structure that corresponds to */proc*, the adversary can deceive administrative tools that rely on */proc* for retrieving system information.

VFS MonitoringRule: The implemented VFS MonitoringRule applies the HAW-based Verification method to detect VFS hooking attacks on */proc* in the Linux filesystem. We observe that the VFS operation function pointers in the *fops* data structure store the addresses of the legitimate filesystem functions. For instance, the VFS function pointers of the data structure of a file in a *ext3* filesystem, point to *ext3* operations in the kernel static region. In the same way, the *fops* data structure of a file in an *NTFS* file system includes pointers to *NTFS* operations. Using this property, we apply HAW-based Verification to detect this particular attack. The procedural flow of the monitor is as follows: First, we trace the exact location of the *fops* data structure using the DMA module and Address Translation Engine. Next, we set the function pointers as critical regions of the MonitoringRule, and the location of the operation functions of the known file systems – such as *ext3*, *ext2*, and *NTFS* – as the whitelist. With these settings, VTMU notifies the *onHawEvent* function of the MonitoringRule, which will subsequently provide notification of this likely malicious

event.

LKM Hiding Attack: Many rootkits take advantage of the Linux kernel’s support of LKM. Initially designed to support extending of the kernel code during runtime without modifying and recompiling the entire kernel, LKMs often serve as a means to inject malicious code into the highest privilege level in a system. Moreover, adversaries often manipulate the linked list data structure that maintains the list of loaded LKMs in order to conceal malicious LKM loaded in the kernel. The following code line frequently appears in rootkits that are injected via LKMs:

```
list_del_init(&__this_module.list);
```

The kernel function `list_del_init` removes the given entry from the list in which it belongs. The developers of rootkits insert the code into the `module_init` function, so that the malicious LKM will be removed from the linked list upon its load. If the snapshot is not taken immediately, this attack cannot be detected because it removes itself from the linked list as soon as it gets loaded.

LKM MonitoringRule: LKM MonitoringRule exemplifies the Callback-based Semantic Verification template used in KI-Mon. By setting the `next` pointer of the LKM linked list head as the critical region of the MonitoringRule, KI-Mon gets notified of the insertion of a new LKM as well as the address of the newly inserted `module` structure. When a new LKM is inserted, the `on_HawEvent` function of the MonitoringRule is triggered, and it requests the DMA module to obtain a snapshot of the new module’s code region and the hash accelerator to hash the contents of the region.

The rest of the procedure to verify if the new LKM is hidden from the list is as follows. First, the monitor waits for 30 milliseconds. Note that the wait time before this check is arbitrary. However, many rootkit LKMs include codes that hide the LKMs in the initialization function [39, 16, 27, 33]. Second, the linked list is traversed with the Data Structure Acquisition Engine to check if the inserted LKM is still in the list. Third, if the LKM is not found in the list, we walk the page table using the Address Translation Engine to verify that the virtual to physical address mapping that correspond to the LKM’s code region has been deleted. The Linux kernel frees the memory regions of the LKM upon its removal. Therefore, the absence of the page table mapping to the region once occupied by the LKM indicates that the LKM was normally removed. In case mapping does exist, the last step of the procedure is executed. Recall that the monitor took a hash of the LKM’s code region: we compare this hash against the hash of the current contents of the physical memory. If the two hashes match, this indicates that the LKM that was not found in the linked list iteration, is not properly freed from the memory. In other words,

the inconsistency between the LKM linked list and the memory contents reveals the LKM hiding attack.

A page table consistency check is used to avoid the hash comparison of the memory contents, which requires additional processing time and memory bandwidth. The Linux kernel allocates the memory space for LKMs using `vmalloc` and de-allocates with `vfree`. The `vmalloc` function allocates a physically non-contiguous region of the requested size. That is, the allocated region is not necessarily contiguous in the physical memory, but is mapped to contiguous virtual addresses. Such non-linear mapping in the page table is deleted as the region is freed, using the `vfree` function. Therefore, the fact that the mapping is deleted in the page table assures that the LKM object is freed in the memory.

Even when page table mapping exists, it does not necessarily mean that a hidden LKM attack has occurred because the region that had been allocated for the LKM was possibly freed already and reallocated for another data object. Thus, a hash comparison of the region is necessary to verify the contents of the region. The kernel constantly allocates and de-allocates memory blocks from the non-contiguous memory regions for `vmalloc` requests. Therefore, it is likely that the freed region that used to hold a data structure object will soon be allocated for new one.

The consistency check is performed once, 30 milliseconds after the detection of a new LKM. This MonitoringRule for the LKM hiding attack, is effective against known LKM hiding technique, deployed in many real-world rootkits. However, it is possible that rootkits evade the single fixed-timed check by delaying the execution of LKM hiding using a timer. To cope with such evasions, we can simply adjust the MonitoringRule to schedule multiple random-interval checks for each occurrence of an LKM loading. For instance, we let the time of first check in seconds t_1 at the interval $[0,5]$, the t_2 at $[5,20]$, and so forth. By setting the lower bound of the random interval of t_n sufficiently long, we render the hiding attack ineffective; the longer the attacker has to wait, the effectiveness of the attack substantially diminishes.

4 Evaluation

In this section, we explain the experiments conducted to prove the effectiveness of the event-triggered mechanism employed in KI-Mon. The VFS MonitoringRule and LKM MonitoringRule were implemented as explained in the previous section. Both successfully detected the example rootkit attacks. In this section, we discuss the implications of the experiment with respect to evaluating the design objectives *O2.3: Minimal overhead on monitored region* and *O2.4: Efficient monitoring processor usage*, which are defined in Section 2.

In addition to the experiments that will be presented and discussed in this section, we conducted an experiment on VTMU whitelist register replacement scheme for large whitelists. While the replacement scheme improves the scalability aspect of KI-Mon, it is rather supplemental to the main experiments. Therefore the experiment is not discussed in this section, but included in the Appendix section.

In the first experiment, we measured the performance overheads, inflicted on the monitored host system by KI-Mon and by a snapshot-only monitor using the LKM MonitoringRule example. Using the same example, KI-Mon’s efficiency, in terms of the CPU usage of the monitoring processor, is presented in the second experiment. The third experiment, which is performed using the LKM MonitoringRule example, compares the detection rate of KI-Mon’s event-triggered mechanism with that of the snapshot-only monitor against frequently recurring LKM hiding attacks.

One desirable requirement for an external kernel integrity monitor is to minimize the performance overhead imposed on the target system. Taking exhaustive memory snapshots would incur a memory bus contention, which in turn would be a major cause of performance degradation of the monitored system. KI-Mon minimizes the performance degradation by applying efficient event-triggered monitoring based on the VTMU hardware module. The snapshot-only version of the VFS monitor was implemented for this experiment. In addition to the monitoring of the inode data structure of /proc, the monitor also performs hash checking on the static regions of the kernel. This corresponds to the default MonitoringRule, which thwarts all modifications to the static regions, in KI-Mon. Here, two benchmarks are used, STREAMBENCH [25], and RAMSPEED [20] to measure the impact on the memory bandwidth performance of the monitored system. These two open-source benchmark tools were ported to our platform with minor modifications: we replaced the floating-point tests with integer tests because the processor on our current prototype does not support floating-point instructions. In addition, we modified the total size of the memory used for the benchmark because the monitored system only has 64 MB of RAM.

Figure 7 shows the average of 10 trials of the measurement using the two benchmark tools. The snapshot-only monitor inevitably incurs performance overhead that is directly proportional to the frequency of the snapshot taking. In order to monitor more dynamic data structures in the dynamic regions of a kernel, the frequency needs to be increased accordingly. This is, however, an inefficient approach to the monitoring of the dynamic regions. KI-Mon implements an event-triggered monitoring mechanism that overcomes this inherent limitation

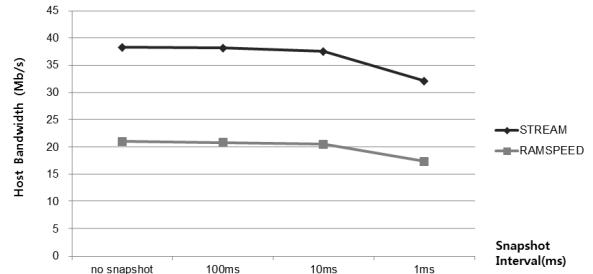


Figure 7: Performance Impact of Snapshots on Monitored System (Avg. of 10 trials): The performance overhead caused by snapshot-only monitor increases as the snapshot interval shortens. When the snapshot interval falls below 1ms, the memory bandwidth of the monitored system drops more than 20%.

of the snapshot-only monitor for an efficient form of dynamic region monitoring. The detection of modifications in KI-Mon does not operate on a periodic basis; VTMU filters memory modification events and trigger the software platform only when an event requires further verification.

4.1 Monitor Processor’s CPU Usage

Efficient usage of the CPU and memory bandwidth is another beneficial aspect for a hardware-based external monitor, such that the monitor can be implemented even with less powerful hardware components. We inserted checkpoints in the software components of KI-Mon and the snapshot-only monitor to analyze the CPU usage of the two monitoring mechanisms. We used the LKM hiding attack example to illustrate the difference in CPU usage between KI-Mon and the snapshot-only monitor.

Figure 8 shows the execution timeline of the two monitoring schemes. The *clock()* function, which is from the standard Linux library, was placed at the beginning and in the end of each functions to record processor times. The snapshot-only monitor repeats the snapshot-based polling before eventually capturing the existence of a newly inserted LKM, whereas KI-Mon stays idle until a HAW-event is received from VTMU. The snapshot-only monitor keeps the external monitor’s CPU active with the snapshot polling until the occurrence of an event.

Each block represents functions that are executed by the LKM MonitoringRule upon the insertion of an LKM by KI-Mon and the snapshot-only monitor. Note that the functions executed after the detection of the events are the same for both monitors. Each snapshot used in the polling takes 400 microseconds of CPU time to read 16 bytes of the LKM linked list head. The *getLKMHash()* took 5600 microseconds for 280 bytes to take a snapshot

of the code section of the LKM. The *checkLKM()* spent 2000 microseconds of CPU time to iterate the LKM linked list of 6 entries to find the newly inserted module. Because it found that the newly inserted module is missing in the list, it took another 1750 microseconds of CPU time to look up the page table entry of the LKM address. The *compareHash()* is finally executed and took 5600 microseconds to take a snapshot of the region that is supposedly the code section of the hidden LKM to confirm that the LKM is indeed hidden. Thus, a total of 14950 microseconds of CPU time were used to verify the event. KI-Mon only uses a total of 14950 microseconds of CPU time for the example, whereas the snapshot-only monitor uses additional CPU time for snapshot polling. Although only a part of the snapshot polling is shown in Figure 8, it should be noted that the polling is constantly running to consume CPU time.

In addition, this particular trial represents a case in which the snapshot-only monitor detects the LKM insertion event; the snapshot-only monitor does not always detect the event. Discussion of the detection rates will be presented later in this section.

While Figure 8 shows the state of the CPU, Figure 9 compares CPU usage rates between the snapshot-only monitor and KI-Mon. The CPU cycles consumed were calculated from the processor times that we obtained for 8. Before the occurrence of the attack, the snapshot-only monitor shows a steady usage over 10^6 cycles per second while KI-Mon does not consume any CPU cycles. At 18 seconds from the origin, an LKM hiding attack was launched using the rootkit sample and both monitoring mechanisms detected the modification and executed the verification procedures, which consume CPU

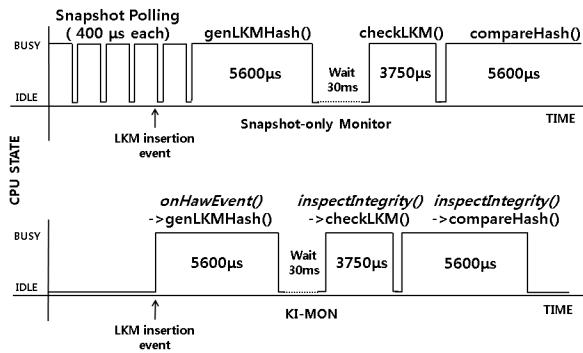


Figure 8: CPU State during Operation of KI-Mon and Snapshot-only Monitor: X-axis represents the time elapsed in microseconds, and Y-axis represents the CPU state as either *busy* or *idle*. The labels in each blocks are the names of the functions being executed during that time.

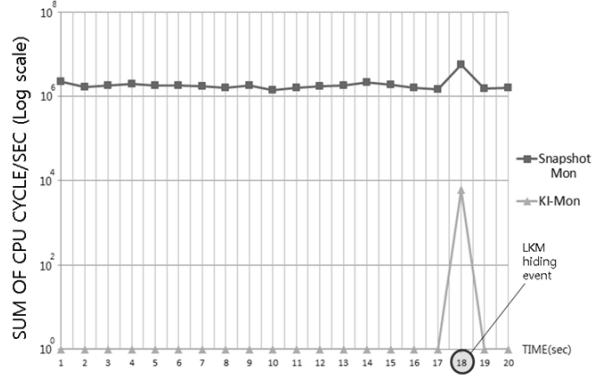


Figure 9: CPU Cycles Consumed in Operation of KI-Mon and Snapshot-only monitor: X-axis represents the time elapsed in seconds, and Y-axis represents the sum of CPU cycles of the external monitor used in log-scale. The vent at 18th second is the LKM hiding attack. Snapshot-only monitor constantly consumes CPU cycles whereas KI-Mon stays idle until an event is occurred.

cycles. The snapshot-only monitor consumes additional CPU cycles to verify the event on top of the periodic polling, whereas KI-Mon consumes only the required number of cycles for verification.

The fundamental difference in the monitoring mechanisms is shown in this experiment. For the snapshot-only monitor to detect an event that occurs with a time interval of t seconds with a snapshot-polling frequency of f hz, a total number of snapshots n is calculated as $t * f$. The times of occurrences of modification events on the monitored data structures are often unpredictable. For instance, connecting a new USB device to a Linux machine might trigger the loading of a corresponding driver LKM. Even for such unpredictable rare events, however, the snapshot-only monitor has no choice but to keep taking snapshots for possible events. Moreover, the frequency of the snapshots may need to be increased to keep up with frequently-changing objects, and this increases the number of snapshots used for polling.

KI-Mon does not consume CPU cycles until an event triggers its operation, whereas the snapshot-only monitor continuously consumes a significant number of CPU cycles until an event is captured. KI-Mon overcomes the inefficiency of the snapshot-only model with its event-triggered mechanism. VTMU replaces the snapshot polling with bus traffic without consuming any CPU cycles because it snoops the bus traffic for modification events. Also, not all events need to be inspected in KI-Mon's mechanism since VTMU filters known legitimate changes with HAW.

Table 2: Detection rate against 100 trials of recurring LKM hiding attack

<i>1khz</i> Snapshot	Max-frequency Snapshot (over <i>10khz</i>)	KI-Mon
4% detected	70% detected	100% detected

4.2 Detection Rate Against Recurring Attacks

The detection rates against frequent and recurring modifications were measured using an LKM hiding attack. As explained in the previous section, many real-world rootkits [1] hide themselves from the LKM linked list when they load. Therefore, the head of the linked list changes for a short period of time, then reverts to the original value. We tested the detection rate for 100 occurrences of such an attack with KI-Mon and with the snapshot-only monitor using *1khz* and *10khz*, the maximum frequency.

Table 2 shows the results of this experiment. The snapshot-only monitor only detected 4% of the attacks with a frequency of *1khz*, and 70%, with a maximum frequency that is over *10khz*. On the other hand, KI-Mon detected all occurrences of attacks. As shown in this experiment, the snapshot-only monitor cannot reliably detect all modifications even with full-throttle snapshot polling. However, KI-Mon maintains a *continuous view* on mutable kernel object with its event-triggered monitoring mechanism. That is, VTMU’s bus traffic monitoring enables tracing of the history of the modifications made to the monitored region. This indicates that KI-Mon is capable of keeping a history of all modifications of the monitored region.

There are cases in which the history of modifications can be used for validation of integrity. This means that the fact that value x was written to the region becomes a trigger for the integrity verification condition y . To be more concrete with the LKM hiding example, KI-Mon detects all LKM insertion events, and then performs an integrity validation for each one of those events. On the other hand, the snapshot-only monitor only detects 70% of the LKM insertions, with 30% of the events were not even given an attempt for verification. The experiment shows the inherent difference in the monitoring mechanisms and proves why KI-Mon is more suitable for monitoring of the dynamic regions of the kernel.

5 Related Work

KI-Mon is an external hardware-based platform that enables event-triggered kernel integrity monitoring. Monitoring rules can be implemented using the KI-Mon API

to monitor mutable kernel objects with invariants. In order to discuss the novelty of our work, we introduce previous works about hardware-based integrity monitoring, monitoring of mutable kernel objects in general, and event-triggered monitoring. We also briefly discuss works that adopt the concept of an independent auditor, and VMM self-protection.

5.1 Hardware-based Kernel/VMM Integrity Monitoring

Before VMM became a popular platform on which to build kernel integrity monitors, several hardware-based operating system kernel monitors were proposed. Zhang et al. [43] was one of the first to propose the concept of integrity monitoring with a coprocessor. Petroni et al. [29] presented Copilot, an external hardware-based kernel integrity monitor based on memory snapshot inspection for static kernel regions.

When virtualization technology emerged, many VMM-based approaches to kernel integrity monitoring were also introduced. A majority of works in kernel integrity monitoring were implemented on VMMs due to the ease of development. However, the expansion of VMMs in both code size and complexity, as well as the attention of researchers and attackers, propelled the discovery of vulnerabilities in VMMs themselves [5, 4, 2, 3]. As a consequence, works that strived to secure the integrity of VMMs with the assistance of hardware support were presented to address the issue [10, 40]. An alternative approach was to implement minimalistic VMMs, so that static analysis could be applied to the minimized attack surface to mitigate vulnerability [37, 23, 36].

HyperSafe [41] took a different approach. This work proposed a self-protection scheme to ensure the integrity of the static region and control flow of VMMs. Azab et al. proposed HyperSentry [10], a VMM-integrity monitor framework in which the root-of-trust is a hardware component (Intel SMM). Recently, in line with Copilot [29], Moon et al. presented Vigilare [26], which introduces the concept of snoop-based monitoring for static immutable regions of operating system kernels using SoC hardware.

5.2 Event-triggered Monitoring

Works that deploy event-triggered monitoring have been presented, following the existing snapshot-based monitoring schemes. Payne et al. [28] presented Lares, which provides a VMM-based platform to add hooks to the monitored system for monitoring; however, their work lacks monitoring schemes that use the proposed technique. KernelGuard [34] and OSck [19], mentioned

in previous section, used the event-triggered monitoring scheme in their works. KernelGuard, by hooking the VMM hypercall, achieved an event-triggered method to map and monitor dynamic regions of the kernel. In addition, OSck adopted both snapshot-based and event-triggered schemes, and used event-triggered schemes to monitor static regions of the kernel.

Even though previous works have dealt with the monitoring of kernel dynamic regions with event-triggered monitoring, they are all designed on VMM-based platforms. On the other hand, KI-Mon implements an event-triggered monitoring scheme as well as having a hardware-based platform on which the monitoring scheme operates. VMM-based event-triggered techniques such as hypercalls or page fault handler hooking are limited to VMM-based platforms.

Vigilare was the first external hardware-based system to introduce event-triggered monitoring with its bus snooping [26]. However, its snooper module was only capable of detecting the occurrence of write traffic on a fixed immutable region. It could not extract a newly updated value from a modification event, nor could it trigger any further verification processing with the event. Thus, Vigilare’s definition of an event is rather primitive and was only sufficient for monitoring a fixed immutable region in the kernel. In order to monitor mutable kernel objects with invariants, KI-Mon refines event generation from bus traffic monitoring by extracting an address and value pair for each event; its hardware-assisted whitelisting scheme eliminates unnecessary event generation for repeated benign updates. Also, its callback-based semantic verification scheme enables monitoring of mutable kernel objects with semantic invariants.

5.3 Monitoring Dynamic Regions of Kernel

Early works in integrity monitoring of operating system kernels have focused on the integrity of static regions. Since monitoring static regions is rather straightforward, many kernel integrity monitors apply similar techniques such as hash checking [29]. Unlike that for static regions, monitoring of dynamic regions of kernels has inherent challenges. As studies have progressed in VMM-based and hardware-based integrity monitoring, numerous works on the monitoring of kernel dynamic regions have been presented [6, 31, 34, 13, 30, 41, 15].

The contents of the dynamic regions of kernels can be mainly put into two categories: control-flow related data and non-control-flow related data. Monitoring the linkages of control-flow related data, which is also known as Control-Flow Integrity (CFI), was introduced by Abadi et al. [6]. Petroni and Hicks [31] defined State-Based Control Flow Integrity (SBCFI) of Linux kernels. This

system is an approximation of CFI. They implemented a monitor that checks the SBCFI of the Linux kernel on a VMM-based platform. Rhee et al. proposed KernelGuard [34] to watch dynamic data of a Linux kernel on a VMM-based platform. Carbone et al. proposed KOP [15], which aimed to map dynamic kernel data from a memory dump of the monitored system. More recently, Hofmann et al. presented OSck [19], which implemented existing monitoring schemes comprehensively with the addition of self-created rootkit attacks and detection mechanisms for monitoring kernel dynamic regions on a VMM-based platform.

KI-Mon focuses on providing an event-triggered mechanism as an architectural foundation for monitoring mutable kernel objects with invariants. Although KI-Mon’s main objective is not to monitor the dynamic regions of a kernel as a whole, the architecture of KI-Mon and its API leaves room for extensions that may cover more mutable objects in the dynamic regions of the kernel.

6 Limitations and Future Work

KI-Mon is a novel hardware-based platform of event-triggered monitoring. Its concepts are shown through experiments with a prototype. Nevertheless, development of a new platform that incorporates both hardware and software components is a rather formidable task. The current prototype of KI-Mon is not at its full maturity. We describe the limitations of the current prototype in this section.

The current prototype has a total of eight address registers for the snooper module. Depending on the required monitoring coverage for KI-Mon, tens or even hundreds of MonitoringRules might run concurrently, which in turn may require a large number of address registers. Design constraints such as hardware cost and chip area would possibly limit the number of registers that can be equipped. For this reason, we plan to explore the possibility of improving the snooper module to utilize a dedicated memory space in addition to the provided registers. On the other hand, we can modify the host kernel’s memory allocation mechanism if the source code of the kernel is provided. More specifically, the kernel can be modified to allocate the monitored data structure of the same types in a contiguous physical memory space so that less number of registers are required for efficient enforcement of MonitoringRules.

We also consider a quantitative estimation of the requirements for KI-Mon’s processing power as future work. We used the same processor for the monitored host and KI-Mon for the prototype. When the monitored host operates at much faster clock speed compared to that of our prototype, the processing power requirements for

KI-Mon needs to be investigated. While it is fairly uncomplicated to design a snooper module that operates at the bus clock speed of the host, the processing power requirements for KI-Mon depend on several other factors such as the required number of MonitoringRules and the computation complexity of each rule. The snooper module is designed to drop incoming HAW-events when its queue is full, hence the optimum combination of the size of the queue and processor speed of KI-Mon needs to be explored.

This paper focuses on illustrating the capability of the KI-Mon platform to efficiently enforce kernel invariants with a principle of event-triggered monitoring. Although the generation of invariants on mutable kernel objects was not discussed as it would exceed the scope of this paper, automation of kernel invariant extraction is another avenue in kernel integrity monitoring. Existing works in the topic aim to infer and enforce *invariants* for each data structure type used in the operating system kernel [12, 30]. Developing or adapting such tools, as well as creating an API extension that can automatically build monitoring rules for KI-Mon based on extracted invariants, will be essential improvements for KI-Mon in terms of applicability.

We discharge a few classes of attacks that are beyond the scope of this paper. Attacks only tampering with processor registers or caches are not considered in this work. Although it might be theoretically possible to devise a rootkit that can reside only in registers and caches, it would be practically impossible to leave no footprint in the memory or in the system bus. Such hypothetical rootkits are not within the scope of this paper. Bahram et al. [11] explain that the existing virtual machine introspection tools are vulnerable to *DKSM* attacks. Just like these VMM-based introspection tools, KI-Mon is also vulnerable to such types of attack that exploit the semantic gap between the monitor and the monitored host system. Difficulties with semantic gaps are an innate weakness of external monitors. To overcome the issue, one possible extension [11, 38] would be the planting of an in-host agent that can interact with KI-Mon. However, it is also notable that KI-Mon is resilient to TLB poisoning attacks. This is because, unlike VMM-based monitors, KI-Mon does not depend on the TLB cache. Instead, KI-Mon walks the host page tables to perform virtual to physical address translation. The KI-Mon processor is independent of the monitored host system, so it cannot use the host processor’s TLB cache.

In addition, we assume that the caches on the host follow a write-through policy, and that the update traffic to registers will always appear on the bus. Today’s processors have a more than 2 level memory hierarchy. The level 2 or higher caches usually use a write-back policy to replace their cache contents. Therefore, if memory traf-

fic is monitored from outside these caches, much of the memory access history would be lost. However, many modern processors have a write-through policy for their level 1 caches [21, 8]. In our hardware design, we connect VTMU right below the L1 write-through cache so that KI-Mon can monitor the whole memory access history of the host processor in a timely manner. This design is viable for some architectures such as ARM Cortex, which do not integrate an L2 cache inside the processor core, but rather only include the L1 cache while providing an interface to the L2 cache that can be assembled later into an SoC along with other hardware components like VTMU.

7 Conclusion

In this paper, we have presented KI-Mon, an external hardware-based monitoring platform that operates on an event-triggered mechanism based on a VTMU hardware unit. Unlike the existing external hardware-based approaches, KI-Mon is an event-triggered verification mechanism, designed to monitor the integrity of dynamic regions of kernels.

We built the KI-Mon prototype on an FPGA-based development board and evaluated the possibility of monitoring dynamic data structures using LKM attack and VFS attack examples. KI-Mon is designed to operate independently of the monitored host system; thus, its operation remains unaffected even when the host is compromised by a rootkit. The hardware platform monitors the host bus traffic and generates events, assisted by its whitelisting capability of filtering benign updates, so that the monitor will not be triggered by common benign updates. This HAW-generated event triggers the software platform to execute verification routines. Also, the KI-Mon API has been developed to support the programmability of the monitoring rules that takes advantage of this event-triggered verification scheme.

Our experiments have showed that KI-Mon consumes significantly fewer CPU cycles due to its event-triggered mechanism because it eliminates the need of constant snapshot-based polling of the monitored region. We have also showed that even at the maximum frequency, the snapshot-only monitor missed 30% of LKM hiding attacks, while KI-Mon was able to detect 100% of the attacks. Overall, KI-Mon lays an architectural foundation for an event-triggered kernel monitoring mechanism on an external hardware-based monitor.

8 Acknowledgments

We would like to thank our shepherd Niels Provos and the anonymous reviewers for insightful com-

ments and suggestions. This research was supported by MOTIE(The Minister of Trade, Industry and Energy), Korea, under the BrainScoutingProgram(HB609-12-3002) by the NIPA(National IT Promotion Agency).

This research is also based on work supported by the Engineering Research Center of Excellence Program of Korea Ministry of Science, ICT & Future Planning(MSIP) (NRF-2008-0062609), and the Center for Integrated Smart Sensors funded by the Ministry of Education, Science and Technology as Global Frontier Project (CISS-20126054193).

References

- [1] <http://packetstormsecurity.com/UNIX/penetration/rootkits>. Last accessed Sep 4, 2012.
- [2] Vmware: Vulnerability statistics. <http://www.cvedetails.com/vendor/252/Vmware.html>. Last accessed April 4, 2012.
- [3] Vulnerability report: Vmware esx server 3.x. <http://seunia.com/advisories/product/10757>. Last accessed April 4, 2012.
- [4] Vulnerability report: Xen 3.x. <http://seunia.com/advisories/product/15863>. Last accessed April 4, 2012.
- [5] Xen: Security vulnerabilities. http://www.cvedetails.com/vulnerability-list/vendor_id-6276/XEN.html. Last accessed April 4, 2012.
- [6] ABADI, M., BUDIU, M., ERLINGSSON, U., AND LIGATTI, J. Control-flow integrity. In *Proceedings of the 12th ACM conference on Computer and communications security* (New York, NY, USA, 2005), CCS '05, ACM, pp. 340–353.
- [7] AEROFLEX GAISLE. *GRLIB IP Core User's Manual*, January 2012.
- [8] ARM. *Cortex-A Series Programmers Guide*, January 2011.
- [9] ARM LIMITED. *AMBATM Specification*, May 1999.
- [10] AZAB, A. M., NING, P., WANG, Z., JIANG, X., ZHANG, X., AND SKALSKY, N. C. Hypersentry: enabling stealthy in-context measurement of hypervisor integrity. In *Proceedings of the 17th ACM conference on Computer and communications security* (New York, NY, USA, 2010), CCS '10, ACM, pp. 38–49.
- [11] BAHRAM, S., JIANG, X., WANG, Z., GRACE, M., LI, J., SRINIVASAN, D., RHEE, J., AND XU, D. Dksm: Subverting virtual machine introspection for fun and profit. In *Reliable Distributed Systems, 2010 29th IEEE Symposium on* (31 2010-nov. 3 2010), pp. 82 –91.
- [12] BALIGA, A., GANAPATHY, V., AND IFTODE, L. Automatic inference and enforcement of kernel data structure invariants. In *Proceedings of the 24th Annual Computer Security Applications Conference* (2008), ACSAC '08.
- [13] BALIGA, A., GANAPATHY, V., AND IFTODE, L. Detecting kernel-level rootkits using data structure invariants. *Dependable and Secure Computing, IEEE Transactions on* 8, 5 (sept.-oct. 2011), 670 –684.
- [14] BOVET, D. P., AND CESATI, M. *Understanding the Linux Kernel*, 2 ed. O'Reilly and Associates, Dec. 2002.
- [15] CARBONE, M., CUI, W., LU, L., LEE, W., PEINADO, M., AND JIANG, X. Mapping kernel objects to enable systematic integrity checking. In *Proceedings of the 16th ACM Conference on Computer and Communications Security* (2009), CCS '09, ACM.
- [16] CYBERWINDS. knark-2.4.3.tgz. <http://packetstormsecurity.com/files/24853/knark-2.4.3.tgz.html>. Last accessed Sep 4, 2012.
- [17] DINABURG, A., ROYAL, P., SHARIF, M., AND LEE, W. Ether: malware analysis via hardware virtualization extensions. In *Proceedings of the 15th ACM conference on Computer and communications security* (New York, NY, USA, 2008), CCS '08, ACM, pp. 51–62.
- [18] HELLSTRÖM, D. *SnapGear Linux for LEON*. Gaisler Research, November 2008.
- [19] HOFMANN, O. S., DUNN, A. M., KIM, S., ROY, I., AND WITCHEL, E. Ensuring operating system kernel integrity with osck. In *Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems* (New York, NY, USA, 2011), ASPLOS '11, ACM, pp. 279–290.
- [20] HOLLANDER, R. M. Ramspeed, a cache and memory benchmarking tool. <http://www.alasir.com/ramspeed/>. Last accessed April 30, 2012.
- [21] INTEL. *Intel 64 and IA-32 Architectures Software Developers Manual*, Aug 2012.
- [22] JUNGHWAN RHEE, D. X. Livedm: Temporal mapping of dynamic kernel memory for dynamic kernel malware analysis and debugging. Tech. rep., 2 2010.
- [23] KANEDA, K. Tiny virtual machine monitor. <http://www.yiss.s.u-tokyo.ac.jp/~kaneda/tvmm/>.
- [24] LOVE, R. *Linux Kernel Development*, 3 ed. Addison Wesley, Nov. 2010.
- [25] MCCALPIN, J. D. Memory bandwidth and machine balance in current high performance computers. *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter* (Dec. 1995), 19–25.
- [26] MOON, H., LEE, H., LEE, J., KIM, K., PAEK, Y., AND KANG, B. B. Vigilare: toward snoop-based kernel integrity monitor. In *Proceedings of the 2012 ACM conference on Computer and communications security* (New York, NY, USA, 2012), CCS '12, ACM, pp. 28–37.
- [27] OPTYX. Kis 0.9. <http://packetstormsecurity.com/files/25029/kis-0.9.tar.gz.html>. Last accessed Sep 4, 2012.
- [28] PAYNE, B. D., CARBONE, M., SHARIF, M., AND LEE, W. Lares: An architecture for secure active monitoring using virtualization. In *Proceedings of the 2008 IEEE Symposium on Security and Privacy* (Washington, DC, USA, 2008), SP '08, IEEE Computer Society, pp. 233–247.
- [29] PETRONI, JR., N. L., FRASER, T., MOLINA, J., AND ARBAUGH, W. A. Copilot - a coprocessor-based kernel runtime integrity monitor. In *Proceedings of the 13th conference on USENIX Security Symposium - Volume 13* (Berkeley, CA, USA, 2004), SSYM'04, USENIX Association, pp. 13–13.
- [30] PETRONI, JR., N. L., FRASER, T., WALTERS, A., AND ARBAUGH, W. A. An architecture for specification-based detection of semantic integrity violations in kernel dynamic data. In *Proceedings of the 15th conference on USENIX Security Symposium - Volume 15* (Berkeley, CA, USA, 2006), USENIX-SS'06, USENIX Association.
- [31] PETRONI, JR., N. L., AND HICKS, M. Automated detection of persistent kernel control-flow attacks. In *Proceedings of the 14th ACM conference on Computer and communications security* (New York, NY, USA, 2007), CCS '07, ACM, pp. 103–115.
- [32] RAFAL WOJTCZUK, JOANNA RUTKOWSKA, A. T. Xen Owning trilogy. <http://invisiblenthingslab.com/it1/Resources.html>, 2008.

- [33] RAISE. Enye lkm rookit modified for ubuntu 8.04. <http://packetstormsecurity.com/files/75184/Enye-LKM-Rookit-Modified-For-Ubuntu-8.04.html>. Last accessed Sep 4, 2012.
- [34] RHEE, J., RILEY, R., XU, D., AND JIANG, X. Defeating dynamic data kernel rootkit attacks via vmm-based guest-transparent monitoring. In *Availability, Reliability and Security, 2009. ARES '09. International Conference on* (march 2009), pp. 74–81.
- [35] RHEE, J., RILEY, R., XU, D., AND JIANG, X. Kernel malware analysis with un-tampered and temporal views of dynamic kernel memory. In *Proceedings of the 13th international conference on Recent advances in intrusion detection* (Berlin, Heidelberg, 2010), RAID'10, Springer-Verlag, pp. 178–197.
- [36] RUSSELL, R. Lguest: The simple x86 hypervisor. <http://lguest.ozlabs.org/>. Last accessed April 31, 2012.
- [37] SESHADRI, A., LUK, M., QU, N., AND PERRIG, A. Secvisor: a tiny hypervisor to provide lifetime kernel code integrity for commodity oses. In *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles* (New York, NY, USA, 2007), SOSP '07, ACM, pp. 335–350.
- [38] SHARIF, M. I., LEE, W., CUI, W., AND LANZI, A. Secure in-vm monitoring using hardware virtualization. In *Proceedings of the 16th ACM conference on Computer and communications security* (New York, NY, USA, 2009), CCS '09, ACM, pp. 477–487.
- [39] TESO, S. adore-ng-0.41.tgz. <http://packetstormsecurity.com/files/32843/adore-ng-0.41.tgz.html>. Last accessed Sep 4, 2012.
- [40] WANG, J., STAVROU, A., AND GHOSH, A. Hypercheck: A hardware-assisted integrity monitor. In *Recent Advances in Intrusion Detection*, S. Jha, R. Sommer, and C. Kreibich, Eds., vol. 6307 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 2010, pp. 158–177. 10.1007/978-3-642-15512-3-9.
- [41] WANG, Z., AND JIANG, X. Hypersafe: A lightweight approach to provide lifetime hypervisor control-flow integrity. In *Proceedings of the 31st IEEE Symposium on Security and Privacy* (2010).
- [42] WANG, Z., JIANG, X., CUI, W., AND WANG, X. Countering persistent kernel rootkits through systematic hook discovery. In *Proceedings of the 11th international symposium on Recent Advances in Intrusion Detection* (Berlin, Heidelberg, 2008), RAID '08, Springer-Verlag, pp. 21–38.
- [43] ZHANG, X., VAN DOORN, L., JAEGER, T., PEREZ, R., AND SAILER, R. Secure coprocessor-based intrusion detection. In *Proceedings of the 10th workshop on ACM SIGOPS European workshop* (New York, NY, USA, 2002), EW 10, ACM, pp. 239–242.

A Appendix

A.1 VTMU Replacement Algorithm for Large Whitelists

In order to utilize KI-Mon’s memory space as an additional storage for whitelist values. We preliminarily implemented an approximation of the LRU replacement scheme, which swaps between the values in the registers and those in memory. The tag registers is set when the value written to the monitored region matches the value in a whitelist register, all tag registers are cleared when all the tag registers are set. KI-Mon compares the update value with the whitelist values in the registers as well

as those in the memory. When a match has occurred with the one in the memory, KI-Mon swaps the matched whitelist value with a value in a register whose tag value is 0.

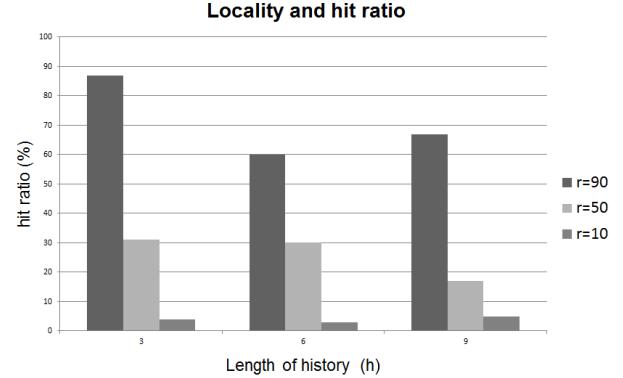


Figure 10: Whitelist LRU test results: X-axis of graph indicates parameter h , length of history locality, and the legend indicates the parameter r , the rate of locality. Y-axis of the graph shows the hit ratio.

We evaluated the replacement scheme with an experiment in which synthesized bus traffic was used as the input. The characteristic of the synthesized bus traffic is modeled with two parameters: length of locality h and rate of locality r . We implemented a traffic generator that writes a value to the monitored the region of VTMU, so that the traffic will trigger the operation of the replacement scheme. The traffic generator chooses a value out of the 100 whitelist values, which consist of h local values and $100 - h$ non-local values. Among these values, we choose a local number out of the h local number with a probability of r , and presumably a non-local number from the $100 - h$ non-local pool with a probability of $1 - r$. Note that higher r or lower h would generate a more local model in this setting.

Figure 10 shows the results of the experiment. We see that for traffic patterns with less locality, which were generated with higher h or lower r , the hit ratio is lower. This means that our replace scheme is less effectively utilized for this particular traffic pattern. For cases with high locality, however, the hit ratio is higher than 50% and tops out at 90%. Note that the number of whitelist registers is much smaller than the whitelist, which has 100 entries. This means that the approximate preliminary LRU scheme helps KI-Mon deal with large whitelists in situations in which where the pattern of benign updates on the monitored regions are local. For every miss, KI-Veri needs to manually check if the modification is legitimate using the whitelist values that are stored in KI-Mon’s main memory. This procedure takes a minimal number of CPU cycles; nevertheless, it could burden the CPU in cases of bursty traffic. While the snapshot-only model consumes CPU cycles for comparing any modified value with the whitelist values for every detection of modifications, KI-Mon only needs to perform a comparison with a probability of $1 - \text{hitratio}$.

WHYPER: Towards Automating Risk Assessment of Mobile Applications

Rahul Pandita, Xusheng Xiao, Wei Yang, William Enck, Tao Xie

North Carolina State University, Raleigh, NC, USA

{rpandit, xxiao2, wei.yang}@ncsu.edu {enck, xie}@csc.ncsu.edu

Abstract

Application markets such as Apple’s App Store and Google’s Play Store have played an important role in the popularity of smartphones and mobile devices. However, keeping malware out of application markets is an ongoing challenge. While recent work has developed various techniques to determine what applications do, no work has provided a technical approach to answer, *what do users expect?* In this paper, we present the first step in addressing this challenge. Specifically, we focus on permissions for a given application and examine whether the application description provides any indication for why the application needs a permission. We present WHYPER, a framework using Natural Language Processing (NLP) techniques to identify sentences that describe the need for a given permission in an application description. WHYPER achieves an average precision of 82.8%, and an average recall of 81.5% for three permissions (address book, calendar, and record audio) that protect frequently-used security and privacy sensitive resources. These results demonstrate great promise in using NLP techniques to bridge the semantic gap between user expectations and application functionality, further aiding the risk assessment of mobile applications.

1 Introduction

Application markets such as Apple’s App Store and Google’s Play Store have become the *de facto* mechanism of delivering software to consumer smartphones and mobile devices. Markets have enabled a vibrant software ecosystem that benefits both consumers and developers. Markets provide a central location for users to discover, purchase, download, and install software with only a few clicks within on-device market interfaces. Simultaneously, they also provide a mechanism for developers to advertise, sell, and distribute their applications. Unfortunately, these characteristics also provide an easy

distribution mechanism for developers with malicious intent to distribute malware.

To address market-security issues, the two predominant smartphone platforms (Apple and Google) use starkly contrasting approaches. On one hand, Apple forces all applications submitted to its App Store to undergo some level of manual inspection and analysis before they are published. This manual intervention allows an Apple employee to read an application’s description and determine whether the different information and resources used by the application are appropriate. On the other hand, Google performs no such checking before publishing an application. While Bouncer [1] provides static and dynamic malware analysis of published applications, Google primarily relies on permissions for security. Application developers must request permissions to access security and privacy sensitive information and resources. This permission list is presented to the user at the time of installation with the implicit assumption that the user is able to determine whether the listed permissions are appropriate.

However, it is non-trivial to classify an application as malicious, privacy infringing, or benign. Previous work has looked at permissions [2–5], code [6–10], and runtime behavior [11–13]. However, underlying all of this work is a caveat: *what does the user expect?* Clearly, an application such as a *GPS Tracker* is expected to record and send the phone’s geographic location to the network; an application such as a *Phone-Call Recorder* is expected to record audio during a phone call; and an application such as *One-Click Root* is expected to exploit a privilege-escalation vulnerability. Other cases are more subtle. The Apple and Google approaches fundamentally differ in who determines whether an application’s permission, code, or runtime behavior is appropriate. For Apple, it is an employee; for Google, it is the end user.

We are motivated by the vision of bridging the semantic gap between what the user expects an application to do and what it actually does. This work is a first step in

this direction. Specifically, we focus on permissions and ask the question, *does the application description provide any indication for the application’s use of a permission?* Clearly, this hypothesis will work better for some permissions than others. For example, permissions that protect a user-understandable resource such as the address book, calendar, or microphone should be discussed in the application description. However, other low-level system permissions such as accessing network state and controlling vibration are not likely to be mentioned. We note that while this work primarily focuses on permissions in the Android platform and relieving the strain on end users, it is equally applicable to other platforms (e.g., Apple) by aiding the employee performing manual inspection.

With this vision, in this paper, we present WHYPER, a framework that uses Natural Language Processing (NLP) techniques to determine *why* an application uses a *permission*. WHYPER takes as input an application’s description from the market and a semantic model of a permission, and determines which sentence (if any) in the description indicates the use of the permission. Furthermore, we show that for some permissions, the permission semantic model can be automatically generated from platform API documents. We evaluate WHYPER against three popularly-used permissions (address book, calendar, and record audio) and a dataset of 581 popular applications. These three frequently-used permissions protect security and privacy sensitive resources. Our results demonstrate that WHYPER effectively identifies the sentences that describe needs of permissions with an average precision of 82.8% and an average recall of 81.5%. We further investigate the sources of inaccuracies and discuss techniques of improvement.

This paper makes the following main contributions:

- We propose the use of NLP techniques to help bridge the semantic gap between what mobile applications do and what users expect them to do. To the best of our knowledge, this work is the first attempt to automate this inference.
- We evaluate our framework on 581 popular Android application descriptions containing nearly 10,000 natural-language sentences. Our evaluation demonstrates substantial improvement over a basic keyword-based searching.
- We provide a publicly available prototype implementation of our approach on the project website [14].

WHYPER is merely the first step in bridging the semantic gap of user expectations. There are many ways in which we see this work developing. Application descriptions are only one form of input. We foresee pos-

sibility in also incorporating the application name, user reviews, and potentially even screen-shots. Furthermore, permissions could potentially be replaced with specific API calls, or even the results of dynamic analysis. We also see great potential in developing automatic or partially manual techniques of creating and fine-tuning permission semantic models.

Finally, this work dovetails nicely with recent discourse concerning the appropriateness of Android permissions to protect user security [15–18]. The overwhelming evidence indicates that most users do not understand what permissions mean, even if they are inclined to look at the permission list [18]. On the other hand, permission lists provide a necessary foundation for security. Markets cannot simultaneously cater to the security and privacy requirements of all users [19], and permission lists allow researchers and expert users to become “whistle blowers” for security and privacy concerns [11]. In fact, a recent comparison [20] of the Android and iOS versions of applications showed that iOS applications overwhelmingly more frequently use privacy-sensitive APIs. Tools such as WHYPER can help raise awareness of security and privacy problems and lower the sophistication required for concerned users to take control of their devices.

The remainder of this paper proceeds as follows. Section 2 presents the overview of the WHYPER framework along with background on NLP techniques used in this work. Section 3 presents our framework and implementation. Section 4 presents evaluation of our framework. Section 6 discusses related work. Finally, Section 7 concludes.

2 WHYPER Overview

We next present a brief overview of the WHYPER framework. The name WHYPER itself is a word-play on phrase *why permissions*. We envision WHYPER to operate between the application market and end users, either as a part of the application market or a standalone system as shown in Figure 1.

The primary goal of the WHYPER framework is to bridge the semantic gap of user expectations by determining *why* an application requires a permission. In particular, we use application descriptions to get this information. Thus, the WHYPER framework operates between the application market and end users. Furthermore, our framework could also serve to help developers with the feedback to improve their applications, as shown by the dotted arrows between developers and WHYPER in Figure 1.

A straightforward way of realizing the WHYPER framework is to perform a keyword-based search on application descriptions to annotate sentences describing

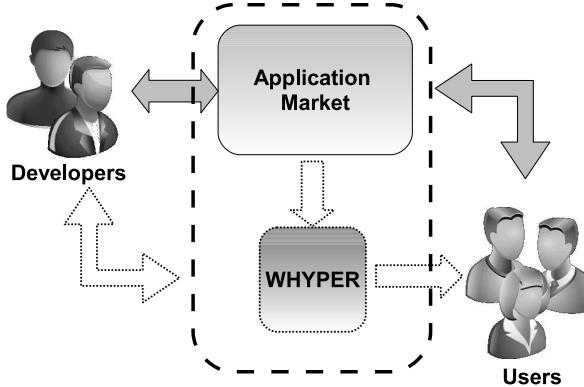


Figure 1: Overview of WHYPER

sensitive operations pertinent to a permission. However, we demonstrate in our evaluation that such an approach is limited by producing many false positives. We propose Natural Language Processing (NLP) as a means to alleviate the shortcomings of keyword-based searching. In particular, we address the following limitations of keyword-based searching:

- 1. Confounding Effects.** Certain keywords such as “contact” have a confounding meaning. For instance, ‘... displays user contacts, ...’ vs ‘... contact me at abc@xyz.com’. The first sentence fragment refers to a sensitive operation while the second fragment does not. However, both fragments include the keyword “contact”.

To address this limitation, we propose NLP as a means to infer semantics such as whether the word refers to a resource or a generic action.

- 2. Semantic Inference.** Sentences often describe a sensitive operation such as *reading contacts* without actually referring to keyword “contact”. For instance, “share... with your friends via email, sms”. The sentence fragment describes the need for *reading contacts*; however the “contact” keyword is not used.

To address this limitation, we propose to use API documents as a source of semantic information for identifying actions and resources related to a sensitive operation.

To the best of our knowledge, ours is the first framework in this direction. We next present the key NLP techniques used in this work.

2.1 NLP Preliminaries

Although well suited for human communication, it is very difficult to convert natural language into unambiguous specifications that can be processed and understood by computers. With recent research advances in the area of NLP, existing NLP techniques have been shown to be fairly accurate in highlighting grammatical structure of a natural language sentence. We next briefly introduce the key NLP techniques used in this work.

Parts Of Speech (POS) Tagging [21, 22]. Also known as ‘word tagging’, ‘grammatical tagging’ and ‘word-sense disambiguation’, these techniques aim to identify the part of speech (such as nouns and verbs) a particular word in a sentence belongs to. Current state-of-the-art approaches have been shown to achieve 97% [23] accuracy in classifying POS tags for well-written news articles.

Phrase and Clause Parsing. Also known as chunking, this technique further enhances the syntax of a sentence by dividing it into a constituent set of words (or phrases) that logically belong together (such as a Noun Phrase and Verb Phrase). Current state-of-the-art approaches can achieve around 90% [23] accuracy in classifying phrases and clauses over well-written news articles.

Typed Dependencies [24,25]. The Stanford-typed dependencies representation provides a hierarchical semantic structure for a sentence using dependencies with precise definitions of what each dependency means.

Named Entity Recognition [26]. Also known as ‘entity identification’ and ‘entity extraction’, these techniques are a subtask of information extraction that aims to classify words in a sentence into predefined categories such as names, quantities, and expressions of time.

We next describe the threat model that we considered while designing our WHYPER framework.

2.2 Use Cases and Threat Model

WHYPER is an enabling technology for a number of use cases. In its simplest form, WHYPER could enable an enhanced user experience for installing applications. For example, the market interface could highlight the sentences that correspond to a specific permission, or raise warnings when it cannot find any sentence for a permission. WHYPER could also be used by market providers to help force developers to disclose functionality to users. In its primitive form, market providers could use WHYPER to ensure permission requests have justifications in the description. More advanced versions of WHYPER could also incorporate the results of static and dynamic application analysis to ensure more semantically appropriate justifications. Such requirements could be placed on all new applications, or iteratively applied to existing applications by automatically emailing developers of applications with unjustified permissions. Alternatively, market providers and security researchers could

use WHYPER to help triage markets [5] for dangerous and privacy infringing applications. Finally, WHYPER could be used in concert with existing crowd-sourcing techniques [27] designed to assess user expectations of application functionality. All of these use cases have unique threat models.

For the purposes of this paper, we consider the generic use scenario where a human is notified by WHYPER if specific permissions requested by an application are not justified by the application’s textual description. WHYPER is primarily designed to help identify privacy infringements in relatively benign applications. However, WHYPER can also help highlight malware that attempts to sneak past consumers by adding additional permissions (e.g., to send premium-rate SMS messages). Clearly, a developer can lie when writing the application’s description. WHYPER does not attempt to detect such lies. Instead, we assume that statements describing unexpected functionality will appear out-of-place for consumers reading an application description before installing it. We note that malware may still hide malicious functionality (e.g., eavesdropping) within an application designed to use the corresponding permission (e.g., an application to take voice notes). However, false claims in an application’s description can provide justification for removal from the market or potentially even criminal prosecution. WHYPER does, however, provide defense against developers that simply include a list of keywords in the application description (e.g., to aid search rankings).

Fundamentally, WHYPER identifies if there is a possible implementation need for a permission based on the application’s description. In a platform such as Android, there are many ways to accomplish the same implementation goals. Some implementation options require permissions, while others do not. For example, an application can make a call that starts Android’s address book application to allow the user to select a contact (requiring no permission), or it can access the address book directly (requiring a permission). From WHYPER’s perspective, it only matters that privileged functionality may work expectedly when using the application, and that functionality is disclosed in the application’s description. Other techniques (e.g., Stowaway [28]) can be used to determine if an application’s code actually requires a permission.

3 WHYPER Design

We next present our framework for annotating the sentences that describe the needs for permissions in application descriptions. Figure 2 gives an overview of our framework. Our framework consists of five components: a preprocessor, an NLP Parser, an intermediate-

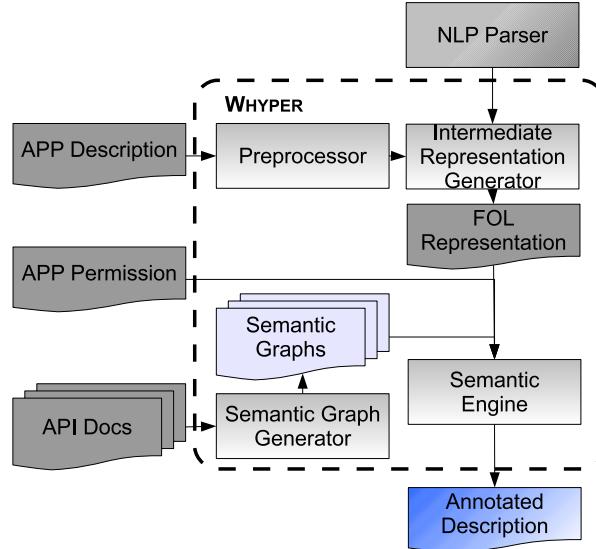


Figure 2: Overview of WHYPER framework

representation generator, a semantic engine (SE), and an analyzer.

The pre-processor accepts application descriptions and preprocesses the sentences in the descriptions, such as annotating sentence boundaries and reducing lexical tokens. The intermediate-representation generator accepts the pre-processed sentences and parses them using an NLP parser. The parsed sentences are then transformed into the first-order-logic (FOL) representation. SE accepts the FOL representation of a sentence and annotates the sentence based on the semantic graphs of permissions. Our semantic graphs are derived by analyzing Android API documents. We next describe each component in detail.

3.1 Preprocessor

The preprocessor accepts natural-language application descriptions and preprocesses the sentences, to be further analyzed by the intermediate-representation generator. The preprocessor annotates sentence boundaries, and reduces the number of lexical tokens using semantic information. The reduction of lexical tokens greatly increases the accuracy of the analysis in the subsequent components of our framework. In particular, the preprocessor performs following preprocessing tasks:

1. Period Handling. In simplistic English, the character period (‘.’) marks the end of a sentence. However, there are other legal usages of the period such as: (1) decimal (periods between numbers), (2) ellipsis (three continuous periods ‘...’), (3) shorthand notations (“Mr.”, “Dr.”, “e.g.”). While these are legal usages, they hinder detection of sentence boundaries, thus forcing the sub-

sequent components to return incorrect or imprecise results.

We pre-process the sentences by annotating these usages for accurate detection of sentence boundaries. We achieve so by looking up known shorthand words from WordNet [29] and detecting decimals, which are also the period character, by using regular expressions. From an implementation perspective, we have maintained a static lookup table of shorthand words observed in WordNet.

2. Sentence Boundaries. Furthermore, there are instances where an enumeration list is used to describe functionality, such as “*The app provides the following functionality: a) abc..., b) xyz...* ”. While easy for a human to understand the meaning, it is difficult from a machine to find appropriate boundaries.

We leverage the structural (positional) information: (1) placements of tabs, (2) bullet points (numbers, characters, roman numerals, and symbols), and (3) delimiters such as “:” to detect appropriate boundaries. We further improve the boundary detection using the following patterns we observe in application descriptions:

- We remove the leading and trailing “*” and ‘-’ characters in a sentence.
- We consider the following characters as sentence separators: ‘-’, ‘-’, ‘ø’, ‘§’, ‘†’, ‘◊’, ‘◊’, ‘♣’, ‘♥’, ‘♠’ ... A comprehensive list can be found on the project website [14].
- For an enumeration sentence that contains at least one enumeration phrase (longer than 5 words), we break down the sentence to short sentences for each enumerated item.

3. Named Entity Handling. Sometimes a sequence of words correspond to the name of entities that have a specific meaning collectively. For instance, consider the phrases “*Pandora internet radio*”, “*Google maps*”, which are the names of applications. Further resolution of these phrases using grammatical syntax is unnecessary and would not bring forth any semantic value. Thus, we identify such phrases and annotate them as single lexical units. We achieve so by maintaining a static lookup table.

4. Abbreviation Handling. Natural-language sentences often consist of abbreviations mixed with text. This can result in subsequent components to incorrectly parse a sentence. We find such instances and annotate them as a single entity. For example, text followed by abbreviations such as “*Instant Message (IM)*” is treated as single lexical unit. Detecting such abbreviations is achieved by using the common structure of abbreviations and encoding such structures into regular expressions. Typically, regular expressions provide a reasonable approximation for handling abbreviations.

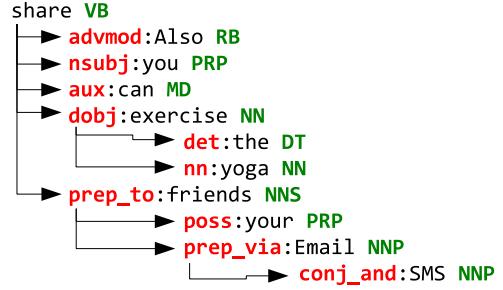


Figure 3: Sentence annotated with Stanford dependencies

3.2 NLP Parser

The NLP parser accepts the pre-processed documents and annotates every sentence within each document using standard NLP techniques. From an implementation perspective, we chose the Stanford Parser [30]. However, this component can be implemented using any other existing NLP libraries or frameworks:

1. **Named Entity Recognition:** NLP parser identifies the named entities in the document and annotates them. Additionally, these entities are further added to the lookup table, so that the preprocessor use the entities for processing subsequent sentences.
2. **Stanford-Typed Dependencies:** [24, 25] NLP parser further annotates the sentences with Stanford-typed dependencies. Stanford-typed dependencies is a simple description of the grammatical relationships in a sentence, and targeted towards extraction of textual relationships. In particular, we use standford-typed dependencies as an input to our intermediate-representation generator.

Next we use an example to illustrate the annotations added by the NLP Parser. Consider the example sentence “*Also you can share the yoga exercise to your friends via Email and SMS.*”, that indirectly refers to the READ_CONTACTS permission. Figure 3 shows the sentence annotated with Stanford-typed dependencies. The words in red are the names of dependencies connecting the actual words of the sentence (in black). Each word is followed by the Part-Of-Speech (POS) tag of the word (in green). For more details on Stanford-typed dependencies and POS tags, please refer to [24, 25].

3.3 Intermediate-Representation Generator

The intermediate-representation generator accepts the annotated documents and builds a relational represen-

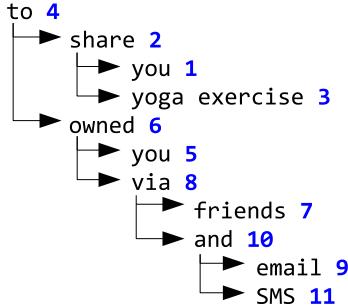


Figure 4: First-order logic representation of annotated sentence in Figure 3

tation of the document. We define our representation as a tree structure that is essentially a First-Order-Logic (FOL) expression. Recent research has shown the adequacy of using FOL for NLP related analysis tasks [31–33]. In our representation, every node in the tree except for the leaf nodes is a predicate node. The leaf nodes represent the entities. The children of the predicate nodes are the participating entities in the relationship represented by the predicate. The first or the only child of a predicate node is the governing entity and the second child is the dependent entity. Together the governing entity, predicate and the dependent entity node form a tuple.

We implemented our intermediate-representation generator based on the principle shallow parsing [34] techniques. A typical shallow parser attempts to parse a sentence based on the function of POS tags. However, we implemented our parser as a function of Stanford-typed dependencies [22, 24, 25, 30]. We chose Stanford-typed dependencies for parsing over POS tags because Stanford-typed dependencies annotate the grammatical relationships between words in a sentence, thus provide more semantic information than POS tags that merely highlight the syntax of a sentence.

In particular, our intermediate-representation generator is implemented as a series of cascading finite state machines (FSM). Earlier research [31, 33–36] has shown the effectiveness and efficiency of using FSM in linguistics analysis such as morphological lookup, POS tagging, phrase parsing, and lexical lookup. We wrote semantic templates for each of the typed dependencies provided by the Stanford Parser.

Table 1 shows a few of these semantic templates. Column “Dependency” lists the name of the Stanford-typed dependency, Column “Example” lists an example sentence containing the dependency, and Column “Description” describes the formulation of tuple from the dependency. All of these semantic templates are publicly available on our project website [14]. Figure 4 shows the FOL representation of the sentence in Figure 3. For ease of understanding, read the words in the order of

the numbers following them. For instance, “you” \leftarrow “share” \rightarrow “yoga exercise” forms one tuple. Notice the additional predicate “owned” annotated 6 in the Figure 4, does not appear in actual sentence. The additional predicate “owned” is inserted when our intermediate-representation generator encounters the possession modifier relation (annotated “poss” in Figure 3).

The FOL representation helps us effectively deal with the problem of *confounding effects* of keywords as described in Section 2. In particular, the FOL assists in distinguishing between a resource that would be a leaf node and an action that would be a predicate node in the intermediate representation of a sentence. The generated FOL representation of the sentence is then provided as an input to the semantic engine for further processing.

3.4 Semantic Engine (SE)

The Semantic Engine (SE) accepts the FOL representation of a sentence and based on the semantic graphs of Android permissions annotates a sentence if it matches the criteria. A semantic graph is basically a semantic representation of the resources which are governed by a permission. For instance, the READ_CONTACTS permission governs the resource “CONTACTS” in Android system.

Figure 5 shows the semantic graph for the permission READ_CONTACTS. A semantic graph primarily constitutes of subordinate resources of a permission (represented in rectangular boxes) and a set of available actions on the resource itself (represented in curved boxes). Section 3.5 elaborates on how we build such graphs systematically.

Our SE accepts the semantic graph pertaining to a permission and annotates a sentence based on the algorithm shown in Algorithm 1. The Algorithm accepts the FOL representation of a sentence rep , the semantic graph associated with the resource of a permission g and a boolean value $recursion$ that governs the recursion. The algorithm outputs a boolean value $isPStmt$, which is `true` if the statement describes the permission associated with a semantic graph (g), otherwise `false`.

Our algorithm systematically explores the FOL representation of the sentence to determine if a sentence describes the need for a permission. First, our algorithm attempts to locate the occurrence of associated resource name within the leaf node of the FOL representation of the sentence (Line 3). The method `findLeafContaining(name)` explores the FOL representation to find a leaf node that contains term `name`. Furthermore, we use WordNet and Lemmatisation [37] to deal with synonyms of a word in question to find appropriate matches. Once a leaf node is found, we systematically traverse the tree from the leaf node to the root,

Table 1: Semantic Templates for Stanford Typed Dependencies

S. No.	Dependency	Example	Description
1	conj	“Send via SMS and email.” conj_and (email, SMS)	Governor (SMS) and dependent (email) are connected by a relationship of conjunction type(and)
2	iobj	“This App provides you with beautiful wallpapers.” iobj (provides, you)	Governor (you) is treated as dependent entity of relationship resolved by parsing the dependent’s (provides) typed dependencies
3	nsubj	“This is a scrollable widget.” nsubj (widget, This)	Governor (This) is treated as governing entity of relationship resolved by parsing the dependent’s (widget) typed dependencies

matching all parent predicates as well as immediate child predicates [Lines 5-16].

Our algorithm matches each of the traversed predicate with the actions associated with the resource defined in semantic graph. Similar to matching entities, we also employ WordNet and Lemmatisation [37] to deal with synonyms to find appropriate matches. If a match is found, then the value `isPStmt` is set to `true`, indicating that the statement describes a permission.

In case no match is found, our algorithms recursively search all the associated subordinate resources in the semantic graph of current resource. A subordinate resource may further have its own subordinate resources. Currently, our algorithm considers only immediate subordinate resources of a resource to limit the false positives.

In the context of the FOL representation shown in Figure 4, we invoke Algorithm 1 with the semantic graph shown in Figure 5. Our algorithm attempts to find a leaf node containing term “CONTACT” or some of its synonym. Since the FOL representation does not contain such a leaf node, algorithm calls itself with semantic graphs of subordinate resources (Line 17-25), namely ‘NUMBER’, ‘EMAIL’, ‘LOCATION’, ‘BIRTHDAY’, ‘ANNIVERSARY’.

The subsequent invocation will find the leaf-node “email” (annotated 9 in Figure 4). Our algorithm then explores the preceding predicates and finds predicate “share” (annotated 2 in Figure 4). The Algorithm matches the word “share” with action “send” (using Lemmatisation and WordNet similarity), one of the actions available in the semantic graph of resource ‘EMAIL’ and returns `true`. Thus, the sentence is appropriately identified as describing the need for permission READ_CONTACT.

3.5 Semantic-Graph Generator

A key aspect of our proposed framework is the employment of a semantic graph of a permission to perform deep semantic inference of sentences. In particular, we propose to initially infer such graphs from API documents.

Algorithm 1 Sentence_Annotator

```

Input: K_Graph g, FOL.rep rep, Boolean recursion
Output: Boolean isPStmt
1: Boolean isPStmt = false
2: String r_name = g.resourceName
3: FOL.rep r' = rep.findLeafContaining(r_name)
4: List actionList = g.actionList
5: while (r'.hasParent) do
6:   if actionList.contains(r'.parent.predicate) then
7:     isPStmt = true
8:     break
9:   else
10:    if actionList.contains(r'.leftSibling.predicate) then
11:      isPStmt = true
12:      break
13:    end if
14:  end if
15:  r' = r'.parent
16: end while
17: if ((NOT(isPStmt)) AND recursion) then
18:   List resourceList = g.resourceList
19:   for all (Resource res in resourceList) do
20:     isPStmt = Sentence_Annotator(getKGraph(res), rep, false)
21:     if isPStmt then
22:       break
23:     end if
24:   end for
25: end if
26: return isPStmt

```

For third-party applications in mobile devices, the relatively limited resources (memory and computation power compared to desktops and servers) encourage development of thin clients. *The key insight to leverage API documents is that mobile applications are predominantly thin clients, and actions and resources provided by API documents can cover most of the functionality performed by these thin clients.*

Manually creating a semantic graph is prohibitively time consuming and may be error prone. We thus came up with a systematic methodology to infer such semantic graphs from API documents that can potentially be automated. First, we leverage Au et al.’s work [38] to find the API document of the class/interface pertaining to a particular permission. Second, we identify the corresponding resource associated with the permission from the API class name. For instance, we identify ‘CONTACTS’ and ‘ADDRESS BOOK’ from the

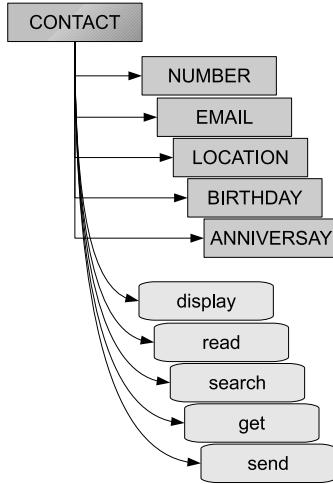


Figure 5: Semantic Graph for the READ_CONTACT permission

`ContactsContract.Contacts`¹ class that is associated with READ_CONTACT permission. Third, we systematically inspect the member variables and member methods to identify actions and subordinate resources.

From the name of member variables, we extract noun phrases and then investigate their types for deciding whether these noun phrases describe resources. For instance, one of member variables of `ContactsContract.Contacts` class leads us to its member variable “email” (whose type is `ContactsContract.CommonDataKinds.Email`). From this variable, we extract noun phrase “EMAIL” and classify the phrase as a resource.

From the name of an Android API public method (describing a possible action on a resource), we extract both noun phrases and their related verb phrases. The noun phrases are used as resources and the verb phrases are used as the associated actions. For instance, `ContactsContract.Contacts` defines operations Insert, Update, Delete, and so on. We consider those operations as individual actions associated with ‘CONTACTS’ resource.

The process is iteratively applied to the individual subordinate resources that are discovered for an API. For instance, “EMAIL” is identified as a subordinate resource in “CONTACT” resource. Figure 5 shows a sub-graph of graph for READ_CONTACT permission.

4 Evaluation

We now present the evaluation of WHYPER. Given an application, the WHYPER framework bridges the semantic

gap between user expectations and the permissions it requests. It does this by identifying in the application description the sentences that describe the need for a given permission. We refer to these sentences as *permission sentences*. To evaluate the effectiveness of WHYPER, we compare the permission sentences identified by WHYPER to a manual annotation of all sentences in the application descriptions. This comparison provides a quantitative assessment of the effectiveness of WHYPER. Specifically, we seek to answer the following research questions:

- **RQ1:** What are the precision, recall and F-Score of WHYPER in identifying permission sentences (i.e., sentences that describe need for a permission)?
- **RQ2:** How effective WHYPER is in identifying permission sentences, compared to keyword-based searching ?

4.1 Subjects

We evaluated WHYPER using a snapshot of popular application descriptions. This snapshot was downloaded in January 2012 and contained the top 500 free applications in each category of the Google Play Store (16,001 total unique applications). We then identified the applications that contained specific permissions of interest.

For our evaluation, we consider the READ_CONTACTS, READ_CALENDAR, and RECORD_AUDIO permissions. We chose these permissions because they protect tangible resources that users understand and have significant enough security and privacy implications that developers should provide justification in the application’s description. We found that 2327 applications had at least one of these three permissions. Since our evaluation requires manual effort to classify each sentence in the application description, we further reduced this set of applications by randomly choosing 200 applications for each permission. This resulted in a total of 600 unique applications for these permissions. The set was further reduced by only considering applications that had an English description). Overall, we analysed 581 application descriptions, which contained 9,953 sentences (as parsed by WHYPER).

4.2 Evaluation Setup

We first manually annotated the sentences in the application descriptions. We had three authors independently annotate sentences in our corpus, ensuring that each sentence was annotated by at least two authors. The individual annotations were then discussed by all three authors to reach to a consensus. In our evaluation, we annotated

¹<http://developer.android.com/reference/android/provider/ContactsContract.Contacts.html>

Table 2: Statistics of Subject permissions

Permission	#N	#S	S_p
READ_CONTACTS	190	3379	235
READ_CALENDAR	191	2752	283
RECORD_AUDIO	200	3822	245
TOTAL	581	9953	763

#N: Number of applications that requests the permission; #S: Total number of sentences in the application descriptions; S_p : Number of sentences manually identified as permission sentences.

a sentence as a permission sentence if at least two authors agreed that the sentence described the need for a permission. Otherwise we annotated the sentence as a permission-irrelevant sentence.

We applied WHYPER on the application descriptions and manually measured the number of true positives (TP), false positives (FP), true negatives (TN) and false negatives (FN) produced by WHYPER as follows:

1. TP : A sentence that WHYPER correctly identifies as a permission sentence.
2. FP : A sentence that WHYPER incorrectly identifies as a permission sentence.
3. TN : A sentence that WHYPER correctly identifies as not a permission sentence.
4. FN : A sentence that WHYPER incorrectly identifies as not a permission sentence.

Table 2 shows the statistics of the subjects used in the evaluations of WHYPER. Column “Permission” lists the names of the permissions. Column “#N” lists the number of applications that requests the permissions used in our evaluations. Column “#S” lists the total number of sentences in application descriptions. Finally, Column “ S_p ” lists the number of sentences that are manually identified as permission sentences by authors. We used this manual identification (Column “ S_p ”) to quantify the effectiveness of WHYPER in identifying permission sentences by answering RQ1. The results of Column “ S_p ” is also used to compare WHYPER with keyword-based searching to answer RQ2 described next.

For RQ2, we applied keyword-based searching on the same subjects. We consider a word as a keyword in the context of a permission if it is a synonym of the word in the permission. To minimize manual efforts, we used words present in `Manifest.permission` class from Android API. Table 4 shows the keywords used in our evaluation. We then measured the number of true positives (TP'), false positives (FP'), true negatives (TN'), and false negatives (FN') produced by the keyword-based searching as follows:

1. TP' :- A sentence that is a permission sentence and contains the keywords.
2. FP' :- A sentence that is not a permission sentence but contains the keywords.
3. TN' :- A sentence that is not a permission sentence and does not contain the keywords.
4. FN' :- A sentence that is a permission sentence but does not contain the keywords.

In statistical classification [39], *Precision* is defined as the ratio of the number of true positives to the total number of items reported to be true, and *Recall* is defined as the ratio of the number of true positives to the total number of items that are true. *F-score* is defined as the weighted harmonic mean of Precision and Recall. *Accuracy* is defined as the ratio of sum of true positives and true negatives to the total number of items. Higher values of precision, recall, F-Score, and accuracy indicate higher quality of the permission sentences inferred using WHYPER. Based on the total number of TPs, FPs, TNs, and FNs, we computed the precision, recall, F-score, and accuracy of WHYPER in identifying permission sentences as follows:

$$Precision = \frac{TP}{TP + FP}$$

$$Recall = \frac{TP}{TP + FN}$$

$$F\text{-score} = \frac{2 \times Precision \times Recall}{Precision + Recall}$$

$$Accuracy = \frac{TP + TN}{TP + FP + TN + FN}$$

4.3 Results

We next describe our evaluation results to demonstrate the effectiveness of WHYPER in identifying contract sentences.

4.3.1 RQ1: Effectiveness in identifying permission sentences

In this section, we quantify the effectiveness of WHYPER in identifying permission sentences by answering RQ1. Table 3 shows the effectiveness of WHYPER in identifying permission sentences. Column “Permission” lists the names of the permissions. Column “ S_I ” lists the number of sentences identified by WHYPER as permission sentences. Columns “TP”, “FP”, “TN”, and “FN” represent the number of true positives, false positives, true negatives, and false negatives, respectively. Columns “P(%)”, “R(%)”, “ $F_S(%)$ ”, and “Acc(%)” list percentage values of precision, recall, F-score, and accuracy respectively. Our results show that, out of 9,953 sentences, WHYPER effectively identifies permission sentences with the average precision, recall, F-score, and

Table 3: Evaluation results

Permission	S_I	TP	FP	FN	TN	P (%)	R (%)	F_S (%)	Acc (%)
READ_CONTACTS	204	186	18	49	2930	91.2	79.1	84.7	97.9
READ_CALENDAR	288	241	47	42	2422	83.7	85.1	84.4	96.8
RECORD_AUDIO	259	195	64	50	3470	75.9	79.7	77.4	97.0
TOTAL	751	622	129	141	9061	82.8*	81.5*	82.2*	97.3*

* Column average; S_I : Number of sentences identified by WHYPER as permission sentences; TP: Total number of True Positives; FP: Total number of False Positives; FN: Total number of False Negatives; TN: Total number of True Negatives; P: Precision; R: Recall; F_S : F-Score; and Acc: Accuracy

accuracy of 82.8%, 81.5%, 82.2%, and 97.3%, respectively.

We also observed that out of 581 applications whose descriptions we used in our experiments, there were only 86 applications that contained at least one false negative statement that were annotated by WHYPER. Similarly, among 581 applications whose descriptions we used in our experiments, there were only 109 applications that contained at least one false positive statement that were annotated by WHYPER.

We next present an example to illustrate how WHYPER incorrectly identifies a sentence as a permission sentence (producing false positives). False positives are particularly undesirable in the context of our problem domain, because they can mislead the users of WHYPER into believing that a description actually describes the need for a permission. Furthermore, an overwhelming number of false positives may result in user fatigue, and thus devalue the usefulness of WHYPER.

Consider the sentence “*You can now turn recordings into ringtones.*”. The sentence describes the application functionality that allows users to create ringtones from previously recorded sounds. However, the described functionality does not require the permission to record audio. WHYPER identifies this sentence as a permission sentence. WHYPER correctly identifies the word *recordings* as a resource associated with the record audio permission. Furthermore, our intermediate representation also correctly constructs that action *turn* is being performed on the resource *recordings*. However, our semantic engine incorrectly matches the action *turn* with the action *start*. The later being a valid semantic action that is permitted by the API on the resource *recording*. In particular, the general purpose WordNet-based Similarity Metric [37] shows 93.3% similarity. We observed that a majority of false positives resulted from incorrect matching of semantic actions against a resources. Such instances can be addressed by using domain-specific dictionaries for synonym analysis.

Another major source of FPs is the incorrect parsing of sentences by the underlying NLP infrastructure. For instance, consider the sentence “*MyLink Advanced provides full synchronization of all Microsoft Outlook*

emails (inbox, sent, outbox and drafts), contacts, calendar, tasks and notes with all Android phones via USB.”. The sentence describes the users calendar will be synchronized. However, the underlying Stanford parser [30] is not able to accurately annotate the dependencies. Our intermediate-representation generator uses shallow parsing that is a function of these dependencies. An inaccurate dependency annotation causes an incorrect construction of intermediate representation and eventually causes an incorrect classification. Such instances will be addressed with the advancement in underlying NLP infrastructure. Overall, a significant number of false positives will be reduced by as the current NLP research advances the underlying NLP infrastructure.

We next present an example to illustrate how WHYPER fails to identify a valid permission sentence (false negatives). Consider the sentence “*Blow into the mic to extinguish the flame like a real candle*”. The sentence describes a semantic action of blowing into the microphone. The noise created by blowing will be captured by the microphone, thus implying the need for record audio permission. WHYPER correctly identifies the word *mic* as a resource associated with the record audio permission. Furthermore, our intermediate representation also correctly shows that the action *blow into* is performed on the resource *mic*. However, from API documents, there is no way for WHYPER framework to infer the knowledge that blowing into microphone semantically implies recording of the sound. Thus, WHYPER fails to identify the sentence as a permission sentence. We can reduce a significant number of false negatives by constructing better semantic graphs.

Similar to reasons for false positives, a major source of false negatives is the incorrect parsing of sentences by the underlying NLP infrastructure. For instance, consider the sentence “*Pregnancy calendar is an application that, not only allows, after entering date of last period menstrual, to calculate the presumed (or estimated) date of birth; but, offering prospects to show, week to week, all appointments which must to undergo every mother, ad a rule, for a correct and healthy pregnancy.*”² The sen-

²Note that the incorrect grammar, punctuation, and spacing are a

tence describes that the users calendar will be used to display weekly appointments. However, the length and complexity in terms of number of clauses causes the underlying Stanford parser [30] to inaccurately annotate the dependencies, which eventually results into incorrect classification.

We also observed that in a few cases, the process followed to identify sentence boundaries resulted in extremely long and possibly incorrect sentences. Consider a sentence that our preprocessor did not identify as a permission sentence for READ_CALENDAR permission:

Daily Brief “How does my day look like today” “Any meetings today” “My reminders” “Add reminder” Essentials Email, Text, Voice dial, Maps, Directions, Web Search “Email John Subject Hello Message Looking forward to meeting with you tomorrow” “Text Lisa Message I will be home in an hour” “Map of Chicago downtown” “Navigate to Millenium Park” “Web Search Green Bean Casserole” “Open Calculator” “Open Alarm Clock” “Launch Phone book” Personal Health Planner ... “How many days until Christmas” Travel Planner “Show airline directory” “Find hotels” “Rent a car” “Check flight status” “Currency converter” Cluze Car Mode Access Daily Brief, Personal Radio, Search, Maps, Directions etc..

A few fragments (sub-sentences) of this incorrectly marked sentence describe the need for read calender permission (“...*My reminder* ... *Add reminder* ...”). However, inaccurate identification of sentence boundaries causes the underlying NLP infrastructure produces a incorrect annotation. Such incorrect annotation is propagated to subsequent phases of WHYPER, ultimately resulting in inaccurate identification of a permission sentence. Overall, as the current research in the field of NLP advances the underlying NLP infrastructure, a significant number of false negatives will be reduced.

4.3.2 RQ2: Comparison to keyword-based searching

In this section, we answer RQ2 by comparing WHYPER to a keyword-based searching approach in identifying permission sentences. As described in Section 4.2, we manually measured the number of permission sentences in the application descriptions. Furthermore, we also manually computed the precision (P), recall (R), f-score (F_S), and accuracy (Acc) of WHYPER as well as precision (P'), recall (R'), f-score (F'_S), and accuracy (Acc') of keyword-based searching in identifying permission sentences. We then calculated the improvement in using WHYPER against keyword-based searching as $\Delta P = P - P'$, $\Delta R = R - R'$, $\Delta F_S = F_S - F'_S$, and $\Delta Acc = Acc - Acc'$. Higher values of ΔP , ΔR , ΔF_S , and ΔAcc are indicative of better performance of WHYPER against keyword-based search.

Table 5 shows the comparison of WHYPER in identifying permission sentences to keyword-based searching

reproduction of the original description.

Table 4: Keywords for Permissions

S. No	Permission	Keywords
1	READ_CONTACTS	contact, data, number, name, email
2	READ_CALENDAR	calendar, event, date, month, day, year
3	RECORD_AUDIO	record, audio, voice, capture, microphone

Table 5: Comparison with keyword-based search

Permission	$\Delta P\%$	$\Delta R\%$	$\Delta F_S\%$	$\Delta Acc\%$
READ_CONTACTS	50.4	1.3	31.2	7.3
READ_CALENDAR	39.3	1.5	26.4	9.2
RECORD_AUDIO	36.9	-6.6	24.3	6.8
Average	41.6	-1.2	27.2	7.7

approach. Columns “ ΔP ”, “ ΔR ”, “ ΔF_S ”, and “ ΔAcc ” list percentage values of increase in the precision, recall, f-scores, and accuracy respectively. Our results show that, in comparison to keyword-based searching, WHYPER effectively identifies permission sentences with the average increase in precision, F-score, and accuracy of 41.6%, 27.2%, and 7.7% respectively. We indeed observed a decrease in average recall by 1.2%, primarily due to poor performance of WHYPER for RECORD_AUDIO permission.

However, it is interesting to note that there is a substantial increase in precision (average 46.0%) in comparison to keyword-based searching. This increase is attributed to a large false positive rate of keyword-based searching. In particular, for descriptions related to READ_CONTACTS permission, WHYPER resulted in only 18 false positives compared to 265 false positives resulted by keyword-based search. Similarly, for descriptions related to RECORD_AUDIO, WHYPER resulted in 64 false positives while keyword-based searching produces 338 false positives.

We next present illustrative examples of how WHYPER performs better than keyword-based search in context of false positives. One major source of false positives in keyword-based search is confounding effects of certain keywords such as *contact*. Consider the sentence “*contact me if there is a bad translation or you'd like your language added!*”. The sentence describes that developer is open to feedback about his application. A keyword-based searching incorrectly identifies this sentence as a permission sentence for READ_CONTACTS permission. However, the word *contact* here refers to an action rather than a resource. In contrast, WHYPER correctly identifies the word *contact* as an action applicable to pronoun *me*. Our framework thus correctly classifies the sentences as a permission-irrelevant sentence.

Consider another sentence “*To learn more, please visit our Checkmark Calendar web site: calendar.greenbeansoft.com*” as an instance of confounding effect of keywords. The sentence is incorrectly identified as a permission sentence for READ_CALENDAR permission because it contains keyword *calendar*. In contrast, WHYPER correctly identifies “Checkmark Calendar” as a named entity rather than resource *calendar*. Our framework thus correctly identifies the sentences as not a permission sentence.

Another common source of false positives in keyword-based searching is lack of semantic context around a keyword. For instance, consider the sentence “*That’s what this app brings to you in addition to learning numbers!*”. A keyword-based search classifies this sentence as an permission sentence because it contains the keyword *number*, which is one of the keywords for READ_CONTACTS permission as listed in Table 4. However, the sentence is actually describing the generic numbers rather than phone numbers. Similar to keyword-based search, our framework also identifies word *number* as a candidate match for subordinating resource *number* in READ_CONTACTS permission (Figure 5). However, the identified semantic action on candidate resource *number* for this sentence is *learning*. Since *learning* is not an applicable action to *phone number* resource in our semantic graphs, WHYPER correctly classifies the sentences as not a permission sentence.

The final category, where WHYPER performed better than keyword-based search, is due to the use of synonyms. For instance, *address book* is a synonym for *contact* resource. Similarly *mic* is synonym for *microphone* resource. Our framework, leverages this synonym information in identifying the resources in a sentence. Synonyms could potentially be used to augment the list of keywords in keyword-based search. However, given that keyword-based search already suffers from a very high false positive rate, we believe synonym augmentation to keywords would further worsen the problem.

We next present discussions on why WHYPER caused a decline in recall in comparison to keyword-based search. We do observe a small increase in recall for READ_CONTACTS (1.3%) and READ_CALENDAR (1.5%) permission related sentences, indicating that WHYPER performs slightly better than keyword-based search. However, WHYPER performs particularly worse in RECORD_AUDIO permission related descriptions, which results in overall decrease in the recall compared to keyword-based search.

One of the reasons for such decline in the recall is an outcome of the false negatives produced by our framework. As described in Section 4.3.1 incorrect identification of sentence boundaries and limitations of underlying NLP infrastructure caused a significant number of false

negatives in WHYPER. Thus, improvement in these areas will significantly decrease the false negative rate of WHYPER and in turn, make the existing gap negligible.

Another cause of false negatives in our approach is inability to infer knowledge for some ‘resource’ - ‘semantic action’ pairs, for instance, ‘microphone’ - ‘blow into’. We further observed, that with a small ***manual effort in augmenting semantic graphs*** for a permission, we could significantly bring down the false negatives of our approach. For instance, after a precursory observation of false negative sentences for RECORD_AUDIO permission manually, we augmented the semantic graphs with just two resource-action pairs (1. microphone-*blow into* and 2. call-*record*). We then applied WHYPER with the augmented semantic graph on READ_CONTACTS permission sentences.

The outcome increased ΔR value from -6.6% to 0.6% for RECORD_AUDIO permission and an average increase of 1.1% in ΔR across all three permissions, without affecting values for ΔP . We refrained from including such modifications for reporting the results in Table 5 to stay true to our proposed framework. In the future, we plan to investigate techniques to construct better semantic graphs for permissions, such as mining user comments and forums.

4.4 Summary

In summary, we demonstrate that WHYPER effectively identifies permission sentences with the average precision, recall, F-score, and accuracy of 80.1%, 78.6%, 79.3%, and 97.3% respectively. Furthermore, we also show that WHYPER performs better than keyword-based search with an average increase in precision of 40% with a relatively small decrease in average recall (1.2%). We also provide discussion that such gap in recall can be alleviated by improving the underlying NLP infrastructure and a little manual effort. We next present discussions on threats to validity.

4.5 Threats to Validity

Threats to external validity primarily include the degree to which the subject permissions used in our evaluations were representative permissions. To minimize the threat, we used permissions that guard against the resources that can be subjected to privacy and security sensitive operations. The threat can be further reduced by evaluating WHYPER on more permissions. Another threat to external validity is the representativeness of the description sentences we used in our experiments. To minimize the threat we randomly selected actual Android application descriptions from a snapshot of the meta-data of

16001 applications from Google Play store (dated January 2012).

Threats to internal validity include the correctness of our implementation in inferring mapping between natural language description sentences and application permissions. To reduce the threat, we manually inspected all the sentences annotated by our system. Furthermore, we ensured that the results were individually verified and agreed upon by at least two authors. The results of our experiments are publicly available on the project website [14].

5 Discussions and Future work

Our framework currently only takes into account application descriptions and Android API documents to highlight permission sentences. Thus, our framework can semi-formally enumerate the uses of a permission. This information can be leveraged in the future to enhance searching for desired applications.

Furthermore, the outputs from our framework could be used in conjunction with program analysis techniques to facilitate effective code reuse. For instance, our framework outputs the reasons of why a permission is needed for an application. These reasons are usually the functionalities provided by the application. In future work, we plan to locate the code fragments that implement the described functionalities. Such mapping can be used as indexes to existing code searching approaches [40, 41] to facilitate effective reuse.

Modular Applications: In our evaluations, we encountered cases where a description referring to another application where the permission sentences were described. For instance, consider the following description sentence “*Navigation2GO is an application to easily launch Google Maps Navigation.*”.

The description states that the current application will launch another application “Google Maps Navigation”, and thus requires the permissions required by that application. Currently, our framework does not deal with such cases. We plan to implement deeper semantic analysis of description sentences to identify such permission sentences.

Generalization to Other Permissions: WHYPER is designed to identify the textual justification for permissions that protect “user understandable” information and resources. That is, the permission must protect an information source or resource that is in the domain of knowledge of general smartphone users, as opposed to a low-level API only known to developers. The permissions studied in this paper (i.e., address book, calendar, microphone) fall within this domain. Based on our studies, we expect similar permissions, such as those that protect SMS interfaces and data stores, the ability to make and

receive phone calls, read call logs and browser history, operate and administer Bluetooth and NFC, and access and use phone accounts will have similar success with WHYPER.

Due to current developer trends and practices, there is class of permissions that we expect will raise alarms for many applications when evaluated with WHYPER. Recent work [7, 11] has shown that many applications leak geographic location and phone identifiers without the users knowledge. We recommend that deployments of WHYPER first focus on other permissions to better gauge and account for developer response. Once general deployment experience with WHYPER has been gained, these more contentious permissions should be tackled. We believe that adding justification for access to geographic location and phone identifiers in the application’s textual description will benefit users. For example, if an application uses location for ads or analytics, the developer should state this in the application description.

Finally, there are some permissions that are implicitly used by applications and therefore will have poor results with WHYPER. In particular, we do not expect the Internet permission to work well with WHYPER. Nearly all smartphone applications access the Internet, and we expect attempts to build a semantic graph for the Internet permission will be largely ineffective.

Results Presentation: A potential after-effect of using WHYPER on existing application descriptions might be more verbose application descriptions. One can argue that it would lead to additional burden on end users to read a lengthy description. However, such additional description provides an opportunity for the users to make informed decisions instead of making assumptions. In future work, we plan to implement and evaluate interfaces to present users with information in a more efficient way, countering user fatigue in case of lengthy descriptions. For example, we may consider using icons in different colors to represent permissions with and without explanation.

6 Related Work

Our proposed framework touches quiet a few research areas such as mining Mobile Application Stores, NLP on software engineering artifacts, program comprehension and software verification. We next discuss relevant work pertinent to our proposed framework in these areas.

Harman et al. [42], first used mining approaches on the application description, pricing, and customer ratings in Blackberry App Store. In particular, they use light-weight text analytics to extract feature information, which is then combined with pricing and customer rating to analyze applications’ business aspects. In contrast, WHYPER uses deep semantic analysis of sentences

in application descriptions, which can be used as a complimentary approach to their feature extraction.

The use of NLP techniques is not new in the software engineering domain. NLP has been used previously in requirements engineering [31, 32, 43]. NLP has even been used to assess the usability of API docs [44].

There are existing approaches [33, 45–47] that apply either NLP or a mix of NLP and machine learning algorithms to infer specifications from the natural-language elements such as code comments, API descriptions, and formal requirements documents. The semi-formal structure of natural-language documents used in these approaches facilitate the application of NLP techniques required for these problem areas. Furthermore, these approaches often rely on some meta-information from source code as well. In contrast, our proposed framework targets a relatively more unconstrained natural-language text and is independent of the source code of the application under analysis.

With respect to program comprehension there are existing techniques that assist in building domain-specific ontologies [48]. Furthermore, there are existing approaches [49, 50] that automatically infer natural-language documentation from source code. These approaches would immensely help in comprehension of the functionality of a mobile application. However, the inherent dependency on source code to generate such documents poses a problem in cases, where source code is not available. In contrast, WHYPER relies on application description and API documents that are readily available artifact for mobile applications.

In the realm of mobile software verification, there is existing work on permissions [2–5, 15], code [6–10], and runtime behavior [11–13] to detect malicious applications. In particular, Zhou et al. [3] propose an approach that leverages the permission information in the manifest of the applications as a criteria to filter malicious applications. They further employed static analysis to identify the malicious applications by forming behavioral patterns in terms of sequences of API calls of known malicious applications. They also propose the use of heuristics-based dynamic analysis to detect previously unknown applications. Furthermore, Enck et al. [11] also use dynamic analysis techniques (dynamic taint analysis) to detect misuse of users private information.

These previously described techniques are primarily targeted towards finding malicious applications in mobile applications. However, these approaches do little for bridging the semantic gap between what the user expects an application to do and what it actually does. In contrast, WHYPER is the first step targeted towards bridging this gap. Furthermore, WHYPER can be used in conjunction with these approaches for an improved experience while interacting with mobile ecosystem.

In addition, Felt et al. [28] apply automated testing techniques to find permission required to invoke each method in the Android 2.2 API. They use this information to detect over-privilege in Android Applications, by generating a maximum set of permissions required by an applications and comparing them to the permissions requested by the application. Although it is important from a developer perspective to minimize the set of permissions requested, the information does not empower an end user to decide what the requested permissions are being used for. In contrast, WHYPER leverages some of the results provided by Felt et al. [28] to highlight the sentences describing the need for a permission, in turn enabling end users to make informed decision while installing and application

Finally, Lin et al. [27] introduce a new model for privacy, namely *privacy as expectations* for mobile applications. In particular, they use crowd-sourcing as means to capture users expectations of sensitive resources used by a mobile applications. We believe the sentences highlighted by WHYPER, can be used as supporting evidence to formulate such user expectations at the first place.

7 Conclusion

In this paper, we have presented WHYPER, a framework that uses Natural Language Processing (NLP) techniques to determine why an application uses a permission. We evaluated our prototype implementation of WHYPER on real-world application descriptions that involve three permissions (address book, calendar, and record audio). These are frequently-used permissions that protect privacy and security sensitive resources. Our evaluation results show that WHYPER achieves an average precision of 82.8%, and an average recall of 81.5% for three permissions. In summary, our results demonstrate great promise in using NLP techniques to bridge the semantic gap of user expectations to aid the risk assessment of mobile applications.

8 Acknowledgments

This work was supported in part by an NSA Science of Security Lablet grant at North Carolina State University, NSF grants CCF-0845272, CCF-0915400, CNS-0958235, CNS-1160603, CNS-1222680, and CNS-1253346. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the funding agencies. We would also like to thank the conference reviewers and shepherds for their feedback in finalizing this paper.

References

- [1] H. Lockheimer, “Android and security,” Google Mobile Blog, Feb. 2012, <http://googlemobile.blogspot.com/2012/02/android-and-security.html>.
- [2] A. P. Felt, M. Finifter, E. Chin, S. Hanna, and D. Wagner, “A survey of mobile malware in the wild,” in *Proc. 1st ACM SPSM Workshop*, 2011, pp. 3–14.
- [3] 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 *Proc. of 19th NDSS*, 2012.
- [4] H. Peng, C. Gates, B. Sarma, N. Li, Y. Qi, R. Potharaju, C. Nita-Rotaru, and I. Molloy, “Using probabilistic generative models for ranking risks of Android Apps,” in *Proc. of 19th ACM CCS*, 2012, pp. 241–252.
- [5] S. Chakradeo, B. Reaves, P. Traynor, and W. Enck, “MAST: Triage for market-scale mobile malware analysis,” in *Proc. 6th of ACM WiSec*, 2013, pp. 13–24.
- [6] M. Egele, C. Kruegel, E. Kirda, and G. Vigna, “PiOS: Detecting privacy leaks in iOS applications.”
- [7] W. Enck, D. Ochteau, P. McDaniel, and S. Chaudhuri, “A study of Android application security,” in *Proc. 20th USENIX Security Symposium*, 2011, p. 21.
- [8] Y. Zhou and X. Jiang, “Dissecting Android malware: Characterization and evolution,” in *Proc. of IEEE Symposium on Security and Privacy*, 2012, pp. 95–109.
- [9] M. Grace, Y. Zhou, Q. Zhang, S. Zou, and X. Jiang, “RiskRanker: Scalable and accurate zero-day Android malware detection,” in *Proc. of 10th MobiSys*, 2012, pp. 281–294.
- [10] C. Gibler, J. Crussell, J. Erickson, and H. Chen, “AndroidLeaks: Automatically detecting potential privacy leaks in Android applications on a large scale,” in *Proc. of 5th TRUST*, 2012, pp. 291–307.
- [11] W. Enck, P. Gilbert, B. Chun, L. Cox, J. Jung, P. McDaniel, and A. Sheth, “TaintDroid: an information-flow tracking system for realtime privacy monitoring on smartphones,” in *Proc. of 9th USENIX OSDI*, 2010, pp. 1–6.
- [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 *Proc. 18th ACM CCS*, 2011, pp. 639–652.
- [13] L. K. Yan and H. Yin, “DroidScope: Seamlessly reconstructing the OS and Dalvik semantic views for dynamic Android malware analysis,” in *Proc. of 21st USENIX Security Symposium*, 2012, p. 29.
- [14] “Whyper,” <https://sites.google.com/site/whypermission/>.
- [15] W. Enck, M. Ongtang, and P. McDaniel, “On lightweight mobile phone application certification,” in *Proc. of 16th ACM CCS*, 2009, pp. 235–245.
- [16] D. Barrera, H. G. Kayacik, P. C. van Oorshot, and A. Somayaji, “A methodology for empirical analysis of permission-based security models and its application to Android,” in *Proc. of 7th ACM CCD*, 2010, pp. 73–84.
- [17] A. P. Felt, K. Greenwood, and D. Wagner, “The effectiveness of application permissions,” in *Proc. of 2nd USENIX WebApps*, 2011, pp. 7–7.
- [18] A. P. Felt, E. Ha, S. Egelman, A. Haney, E. Chin, and D. Wagner, “Android permissions: User attention, comprehension and behavior,” in *Proc. of 8th SOUPS*, 2012, p. 3.
- [19] P. McDaniel and W. Enck, “Not so great expectations: Why application markets haven’t failed security,” *IEEE Security & Privacy Magazine*, vol. 8, no. 5, pp. 76–78, 2010.
- [20] J. Han, Q. Yan, D. Gao, J. Zhou, and R. Deng, “Comparing mobile privacy protection through cross-platform applications,” in *Proc. of 20th NDSS*, 2013.
- [21] K. Toutanova, D. Klein, C. D. Manning, and Y. Singer, “Feature-rich part-of-speech tagging with a cyclic dependency network,” in *Proc. HLT-NAAACL*, 2003, pp. 252–259.
- [22] D. Klein and D. Manning, Christopher, “Fast exact inference with a factored model for natural language parsing,” in *Proc. 15th NIPS*, 2003, pp. 3 – 10.
- [23] “The Stanford Natural Language Processing Group,” 1999, <http://nlp.stanford.edu/>.
- [24] M. C. de Marneffe, B. MacCartney, and C. D. Manning, “Generating typed dependency parses from phrase structure parses,” in *Proc. 5th LREC*, 2006, pp. 449–454.

- [25] M. C. de Marneffe and C. D. Manning, “The stanford typed dependencies representation,” in *Proc. Workshop COLING*, 2008, pp. 1–8.
- [26] J. R. Finkel, T. Grenager, and C. Manning., “Incorporating non-local information into information extraction systems by gibbs sampling,” in *Proc. 43nd ACL*, 2005, pp. 363–370.
- [27] J. Lin, N. Sadeh, S. Amini, J. Lindqvist, J. I. Hong, and J. Zhang, “Expectation and purpose: understanding users’ mental models of mobile App privacy through crowdsourcing,” in *Proc. 14th ACM Ubicomp*, 2012, pp. 501–510.
- [28] A. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner, “Android permissions demystified,” in *Proc. of 18th ACM CCS*, 2011, pp. 627–638.
- [29] Fellbaum et al., *WordNet: an electronic lexical database*. Cambridge, Mass: MIT Press, 1998.
- [30] D. Klein and C. D. Manning, “Accurate unlexicalized parsing.” in *Proc. 41st ACL*, 2003, pp. 423–430.
- [31] A. Sinha, A. M. Paradkar, P. Kumanan, and B. Boguraev, “A linguistic analysis engine for natural language use case description and its application to dependability analysis in industrial use cases.” in *Proc. 39th DSN*, 2009, pp. 327–336.
- [32] A. Sinha, S. M. SuttonJr., and A. Paradkar, “Text2Test: Automated inspection of natural language use cases,” in *Proc. 3rd ICST*, 2010, pp. 155–164.
- [33] R. Pandita, X. Xiao, H. Zhong, T. Xie, S. Oney, and A. Paradkar, “Inferring method specifications from natural language API descriptions,” in *Proc. 34th ICSE*, 2012, pp. 815–825.
- [34] B. K. Boguraev, “Towards finite-state analysis of lexical cohesion.” in *Proc. 3rd FSMNLP*, 2000.
- [35] M. Stickel and M. Tyson, *FASTUS: A Cascaded Finite-state Transducer for Extracting Information from Natural-language Text*. MIT Press, 1997.
- [36] G. Gregory, *Light Parsing as Finite State Filtering*. Cambridge University Press, 1999.
- [37] Q. Do, D. Roth, M. Sammons, Y. Tu, and V. Vydiswaran, “Robust, light-weight approaches to compute lexical similarity,” University of Illinois, Computer Science Research and Technical Reports, 2009.
- [38] K. W. Y. Au, Y. F. Zhou, Z. Huang, and D. Lie, “PScout: analyzing the Android permission specification,” in *Proc. 19th CCS*, 2012, pp. 217–228.
- [39] D. Olson, *Advanced Data Mining Techniques*. Springer Verlag, 2008.
- [40] S. Thummalapenta and T. Xie, “PARSEWeb: A programmer assistant for reusing open source code on the web,” in *Proc. 22nd ASE*, 2007, pp. 204–213.
- [41] S. P. Reiss, “Semantics-based code search,” in *Proc. 31st ICSE*, 2009, pp. 243–253.
- [42] M. Harman, Y. Jia, and Y. Zhang, “App store mining and analysis: MSR for app stores,” in *Proc. 9th IEEE MSR*, 2012, pp. 108–111.
- [43] V. Gervasi and D. Zowghi, “Reasoning about inconsistencies in natural language requirements,” *ACM Transactions Software Engineering Methodologies*, vol. 14, pp. 277–330, 2005.
- [44] U. Dekel and J. D. Herbsleb, “Improving API documentation usability with knowledge pushing,” in *Proc. 31st ICSE*, 2009, pp. 320–330.
- [45] L. Tan, D. Yuan, G. Krishna, and Y. Zhou, “/*iComment: Bugs or bad comments?*/,” in *Proc. 21st SOSP*, 2007, pp. 145–158.
- [46] H. Zhong, L. Zhang, T. Xie, and H. Mei, “Infering resource specifications from natural language API documentation,” in *Proc. 24th ASE*, November 2009, pp. 307–318.
- [47] X. Xiao, A. Paradkar, S. Thummalapenta, and T. Xie, “Automated extraction of security policies from natural-language software documents,” in *Proc. 20th FSE*, 2012, pp. 12:1–12:11.
- [48] H. Zhou, F. Chen, and H. Yang, “Developing application specific ontology for program comprehension by combining domain ontology with code ontology,” in *Proc. 8th QSIC*, 2008, pp. 225 –234.
- [49] G. Sridhara, L. Pollock, and K. Vijay-Shanker, “Generating parameter comments and integrating with method summaries,” in *Proc. 19th ICPC*, 2011, pp. 71–80.
- [50] P. Robillard, “Schematic pseudocode for program constructs and its computer automation by schema-code,” *Comm. of the ACM*, vol. 29, no. 11, pp. 1072–1089, 1986.

Effective Inter-Component Communication Mapping in Android with *Epicc*: An Essential Step Towards Holistic Security Analysis

Damien Octeau¹, Patrick McDaniel¹, Somesh Jha², Alexandre Bartel³, Eric Bodden⁴, Jacques Klein³, and Yves Le Traon³

¹*Department of Computer Science and Engineering, Pennsylvania State University*

²*Computer Sciences Department, University of Wisconsin,*

³*Interdisciplinary Centre for Security, Reliability and Trust, University of Luxembourg*

⁴*EC SPRIDE, Technische Universität Darmstadt*

{octeau,mcdaniel}@cse.psu.edu, jha@cs.wisc.edu, {alexandre.bartel,jacques.klein,yves.lettraon}@uni.lu, eric.bodden@ec-spride.de

Abstract

Many threats present in smartphones are the result of interactions between application components, not just artifacts of single components. However, current techniques for identifying inter-application communication are ad hoc and do not scale to large numbers of applications. In this paper, we reduce the discovery of inter-component communication (ICC) in smartphones to an instance of the Interprocedural Distributive Environment (IDE) problem, and develop a sound static analysis technique targeted to the Android platform. We apply this analysis to 1,200 applications selected from the Play store and characterize the locations and substance of their ICC. Experiments show that full specifications for ICC can be identified for over 93% of ICC locations for the applications studied. Further the analysis scales well; analysis of each application took on average 113 seconds to complete. Epicc, the resulting tool, finds ICC vulnerabilities with far fewer false positives than the next best tool. In this way, we develop a scalable vehicle to extend current security analysis to entire collections of applications as well as the interfaces they export.

1 Introduction

The rapid rise of smartphone has led to new applications and modes of communication [1]. The scale of the new software markets is breathtaking; Google’s Play Store has served billions of application downloads [31] within a few years. Such advances have also come with a dark side. Users are subjected to privacy violations [11, 12] and malicious behaviors [33] from the very applications they have come to depend on. Unfortunately, for many reasons, application markets cannot provide security assurances on the applications they serve [26], and previous attempts at doing so have had limited success [27].

Past analyses of Android applications [12, 14, 15, 17, 19, 36] have largely focused on analyzing application

components in isolation. Recent works have attempted to expose and analyze the interfaces provided by components to interact [6, 12], but have done so in ad hoc and imprecise ways. Conversely, this paper attempts to formally recast Inter-Component Communication (ICC) analysis to infer the locations and substance of all inter- and intra-application communication available for a target environment. This approach provides a high-fidelity means to study how components interact, which is a necessary step for a comprehensive security analysis. For example, our analysis can also be used to perform information flow analysis between application components and to identify new types of attacks, such as application collusion [5, 8], where two applications work together to compromise the privacy of the user. In general, most vulnerability analysis techniques for Android need to analyze ICC, and thus can benefit from our analysis.

Android application components interact through ICC objects – mainly *Intents*. Components can also communicate across applications, allowing developers to reuse functionality. The proposed approach identifies a *specification* for every ICC source and sink. This includes the location of the ICC entry point or exit point, the ICC Intent action, data type and category, as well as the ICC Intent key/value types and the target component name. Note that where ICC values are not fixed we infer all possible ICC values, thereby building a complete specification of the possible ways ICC can be used. The specifications are recorded in a database in flows detected by matching compatible specifications. The structure of the specifications ensures that ICC matching is efficient.

We make the following contributions in this work:

- We show how to reduce the analysis of Intent ICC to an Interprocedural Distributive Environment (IDE) problem. Such a problem can be solved efficiently using existing algorithms [32].
- We develop Epicc, a working analysis tool built on top of an existing IDE framework [3] within the Soot [34] suite, which we have made available

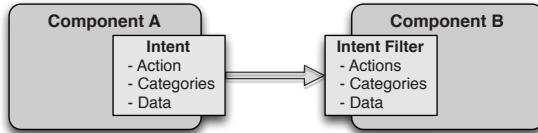


Figure 1: Implicit Intent ICC

at <http://siis.cse.psu.edu/epicc/>.

- We perform a study of ICC vulnerabilities and compare it to ComDroid [6], the current state-of-the-art. Our ICC vulnerability detection shows significantly increased precision, with ComDroid flagging 32% more code locations. While we use our tool to perform a study of some ICC vulnerabilities, our analysis can be used to address a wider variety of ICC-related vulnerabilities.
- We perform a study of ICC in 1,200 representative applications from the free section of the Google Play Store. We found that the majority of specifications were relatively narrow, most ICC objects having a single possible type. Also, key/value pairs are widely used to communicate data over ICC. Lastly, our analysis scales well, with an average analysis time of 113 seconds per application.

1.1 Android ICC

Android applications are developed in Java and compiled to a platform-specific Dalvik bytecode, and are composed of four types of components:

- An *Activity* represents a user screen. The user interface is defined through Activities.
- A *Service* allows developers to specify processing that should take place in the background.
- A *Content Provider* allows sharing of structured data within and across applications.
- A *Broadcast Receiver* is a component that receives broadcast communication objects, called *Intents*.

Intents are the primary vehicle for ICC. For example, a developer might want to start a component to display the user’s current location on a map. She can create an Intent containing the user’s location and send it to a component that renders the map. Developers can specify an Intent’s target component (or target components) in two ways, (a) explicitly, by specifying the target’s application package and class name, and (b) implicitly by setting the Intent’s *action*, *category* or *data* fields.

In order for a component to be able to receive implicit Intents, *Intent Filters* have to be specified for it in the application’s manifest file. Illustrated in Figure 1, Intent Filters describe the action, category or data fields of the Intents that should be delivered by the operating system to a given application component.

ICC can occur both within a single application and between different applications. In order for a component to

be accessible to other applications, its *exported* attribute has to be set to *true* in the manifest file. If the *exported* attribute of a component is not defined, the OS makes the component available to other applications if an Intent Filter has been declared for it.

Intents can carry extra data in the form of key-value mappings. This data is contained in a Bundle object associated with the Intent. Intents can also carry data in the form of URIs with context-specific references to external resources or data.

Developers can restrict access to components using permissions. Permissions are generally declared in the manifest file. A component protected by a permission can only be addressed by applications that have obtained that permission. Permission requests by applications are granted by users at install time and enforced by the OS at runtime.

2 Android ICC Analysis

As highlighted above, the goal of the analysis presented in this paper is to infer specifications for each ICC source and sink in the targeted applications. These specifications detail the type, form, and data associated with the communication. We consider communication with Content Providers to be out of scope. Our analysis has the following goals:

Soundness - The analysis should generate *all* specifications for ICC that may appear at runtime. Informally, we want to guarantee that no ICC will go undetected. Our analysis was designed to be sound under the assumption that the applications use no reflection or native calls, and that the components’ life cycle is modeled completely.

Precision - The previous goal implies that some generated ICC specifications may not happen at runtime (“false positives”). Precision means that we want to limit the number of cases where two components are detected as connected, even though they are not in practice. Our analysis currently does not handle URIs¹. Since the data contained in Intents in the form of URIs is used to match Intents to target components, not using URIs as a matching criterion potentially implies more false positives. Other possible sources of imprecision include the points-to and string analyses. We empirically demonstrate analysis precision in Section 6.1.

Note that, since we do not handle URIs yet, this implies that Content Providers are out of the scope of this paper and will be handled in future work.

¹Extending the analysis to include URIs is a straightforward exercise using the same approaches defined in the following sections. We have a working prototype and defer reporting on it to future work.

```

1| private OnClickListener mMyListener =
2|     new OnClickListener() {
3|         public void onClick(View v) {
4|             Intent intent = new Intent();
5|             intent.setAction("a.b.ACTION");
6|             intent.addCategory("a.b.CATEGORY");
7|             startActivity(intent); } };

```

Figure 2: Example of implicit Intent communication

2.1 Applications

Although Android applications are developed in Java, existing Java analyses cannot handle the Android-specific ICC mechanisms. The analysis presented in this paper deals with ICC and can be used as the basis for numerous important analyses, for example:

Finding ICC vulnerabilities - Android ICC APIs are complex to use, which causes developers to commonly leave their applications vulnerable [6, 12]. Examples of ICC vulnerabilities include sending an Intent that may be intercepted by a malicious component, or exposing components to be launched by a malicious Intent. The first application of our work is in finding these vulnerabilities. We present a study of ICC vulnerabilities in Section 6.4.

Finding attacks on ICC vulnerabilities - Our analysis can go beyond ICC vulnerability detection and can be used for a holistic attack detection process. For each app, we compute entry points and exit points and systematically match them with entry and exit points of previously processed applications. Therefore, our analysis can detect applications that may exploit a given vulnerability.

Inter-component information flow analysis - We compute which data sent at an exit point can potentially be used at a receiving entry point. An information flow analysis using our ICC analysis find flows between a source in a component and a sink in a different component (possibly in a different application).

In the case where the source and sink components belong to different applications, we can detect cases of *application collusion* [5, 8]. The unique communication primitives in Android allow for a new attack model for malicious or privacy-violating application developers. Two or more applications can work together to leak private information and go undetected. For example, application A can request access to GPS location information, while application B requests access to the network. Permissions requested by each application do not seem suspicious, therefore a user might download both applications. However, in practice it is possible for A and B to work together to leak GPS location data to the network. It is almost impossible for users to anticipate this kind of breach of privacy. However, statically detecting this attack is a simple application of our ICC analysis, whereas the current state-of-the-art requires dynamic analysis and modification of the Android platform [5].

```

1| public void onClick(View v) {
2|     Intent i = new Intent();
3|     i.putExtra("Balance", this.mBalance);
4|     if (this.mCondition) {
5|         i.setClassName("a.b",
6|                         "a.b.MyClass");
7|     } else {
8|         i.setAction("a.b.ACTION");
9|         i.addCategory("a.b.CATEGORY");
10|        i = modifyIntent(i);
11|    }
12|    startActivity(i); }
13| public Intent modifyIntent(Intent in) {
14|     Intent intent = new Intent(in);
15|     intent.setAction("a.b.NEW_ACTION");
16|     intent.addCategory("a.b.NEW_CATEGORY");
17|     return intent; }

```

Figure 3: Intent communication: running example

2.2 Examples

Figure 2 shows a representative example of ICC programming. It defines a field that is a click listener. When activated by a click on an element, it creates Intent *intent* and sets its action and category. Finally, the *startActivity()* call takes *intent* as an argument. It causes the OS to find an activity that accepts Intents with the given action and category. When such an activity is found, it is started by the OS. If several activities meeting the action and category requirements are found, the user is asked which activity should be started.

This first example is trivial. Let us now consider the more complex example from Figure 3, which will be used throughout this paper. Let us assume that this piece of code is in a banking application. First, Intent *intent* containing private data is created. Then, if condition *this.mCondition* is true, *intent* is made explicit by targeting a specific class. Otherwise, it is made implicit. Next, an activity is started using *startActivity()*. Note that we have made the implicit Intent branch contrived to demonstrate how function calls are handled. In this example, the safe branch is the one in which *intent* targets a specific component. The other one may leak data, since it might be intercepted by a malicious Activity. We want to be able to detect that possible information leak. In other words, we want to infer the two possible Intent values at *startActivity()*. In particular, knowing the implicit value would allow us to find which applications can intercept it and to detect possible eavesdropping.

3 Connecting Application Components: Overview

Our analysis aims at connecting components, both within single applications and between different applications. For each input application \mathcal{A} , it outputs the following:

1. A list of entry points for \mathcal{A} that may be called by com-

- ponents in \mathcal{A} or in other applications.
2. A list of exit points for \mathcal{A} where \mathcal{A} may send an Intent to another component. That component can be in \mathcal{A} or in a different application. The value of Intents at each exit point is precisely determined, which allows us to accurately determine possible targets.
 3. A list of links between \mathcal{A} 's own components and between \mathcal{A} 's components and other applications' components. These links are computed using 1. and 2. as well as all the previously analyzed applications.

Let us consider the example in Figure 3, which is part of our example banking application. The `startActivity(i)` instruction is an exit point for the application. Our analysis outputs the value of i at this instruction as well as all the possible targets. These targets can be components of our banking application itself or components of previously analyzed applications.

Figure 4 shows an overview of our component matching process. It can be divided into three main functions:

- Finding target components that can be started by other components (i.e. “entry points”) and identifying criteria for a target to be activated.
- Finding characteristics of exit points, i.e. what kind of targets can be activated at these program points.
- Matching exit points with possible targets.

Given an application, we start by parsing its manifest file to extract package information, permissions used and a list of components² and associated intent filters (1). These components are the potential targets of ICC. We match these possible entry points with the pool of already computed exit points (2). We then add the newly computed entry points to our database of entry points (3). This database and the exit points database grow as we analyze more applications. Then we proceed with the string analysis, which identifies key API method arguments such as action strings or component names (4). Next, the main Interprocedural Distributive Environment (IDE) analysis precisely computes the values of Intent used at ICC API calls (5). It also compute the values of Intent Filters that select Intents received by dynamically registered Broadcast Receivers. These exit points are matched with entry points from the existing pool of entry points (6). The newly computed exit points are stored in the exit point database to allow for later matching (7). The values associated with dynamically registered Broadcast Receivers are used for matching with exit points in the database (8). Finally, these values are stored in the entry point database (9).

One of the inputs to our analysis is a set of class files. These classes are in Java bytecode format, since our analysis is built on top of Soot [34], an existing Java analysis framework. Android application code is distributed in

a platform-specific Dalvik bytecode format that is optimized for resource-constrained devices, such as smartphones and tablets. Therefore, we use Dare [29], an existing tool that efficiently and accurately retarget Dalvik bytecode to Java bytecode. While other tools such as dex2jar³ and ded [28] are available, Dare is currently the only formally defined one and other tools’ output is sometimes not reliable.

The manifest parsing step is trivial and we use a simple string analysis (see Section 6). Also, the matching process matches exit points with entry points. It can be made efficient if properly organized in a database. Thus, we focus our description on the main IDE analysis.

It is important to distinguish between what is computed by the string analysis and by the IDE analysis. In the example from Figure 2, the string analysis computes the values of the arguments to the API calls `setAction()` and `addCategory()`. The IDE analysis, on the other hand, uses the results from the string analysis along with a model of the Android ICC API to determine the value of the Intent. In particular, in Figure 2, it determines that, at the call to `startActivity()`, Intent $intent$ has action `a.b.ACTION` and category `a.b.CATEGORY`. In Figure 3, the IDE analysis tells us that i has two possible values at the call to `startActivity()` and determines exactly what the two possible values are.

Reducing the Intent ICC problem to an IDE problem [32] has important advantages. Our analysis is scalable (see Section 6). Further, it is a precise analysis, in the sense that it generates few false positives (links between two components which may not communicate in reality). Thus, security analyses using our ICC analysis will not be plagued by ICC-related false positives. This precision is due to the fact that the IDE framework is flow-sensitive, inter-procedural and context-sensitive.

The flow-sensitivity means that we can distinguish Intent values between different program points. In the example from Figure 3, if Intent i was used for ICC right before the call to `modifyIntent()`, we would accurately capture that this value is different from the one at `startActivity()`. The context-sensitivity means that the analysis of the call to `modifyIntent()` is sensitive to the method’s calling context. If `modifyIntent()` is called at another location with a different argument $i2$, the analysis will precisely distinguish between the values returned by the two calls. Otherwise, in a context-insensitive analysis, the return value would summarize all possible values given all contexts in which `modifyIntent()` is called in the program. The value of i computed by a context-insensitive analysis would be influenced by the value of $i2$, which is not the case in reality. That would be significantly less precise, resulting in more false positives.

²Broadcast Receivers can be registered either statically in the manifest file or dynamically using the `registerReceiver()` methods.

³Available at <http://code.google.com/p/dex2jar/>.

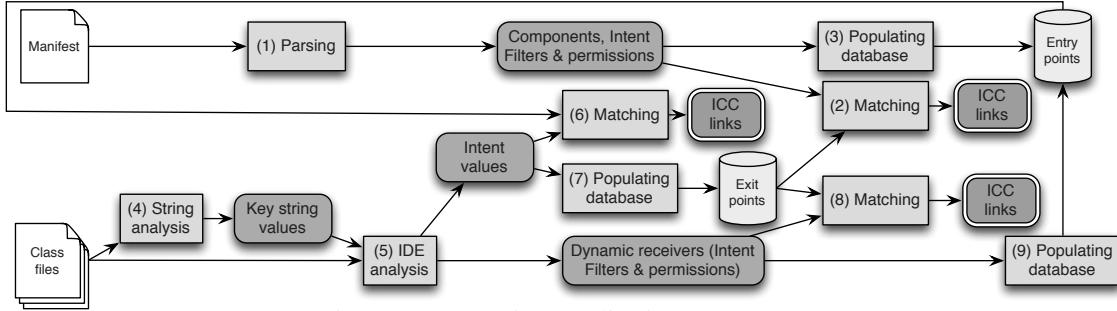


Figure 4: Connecting Application Components

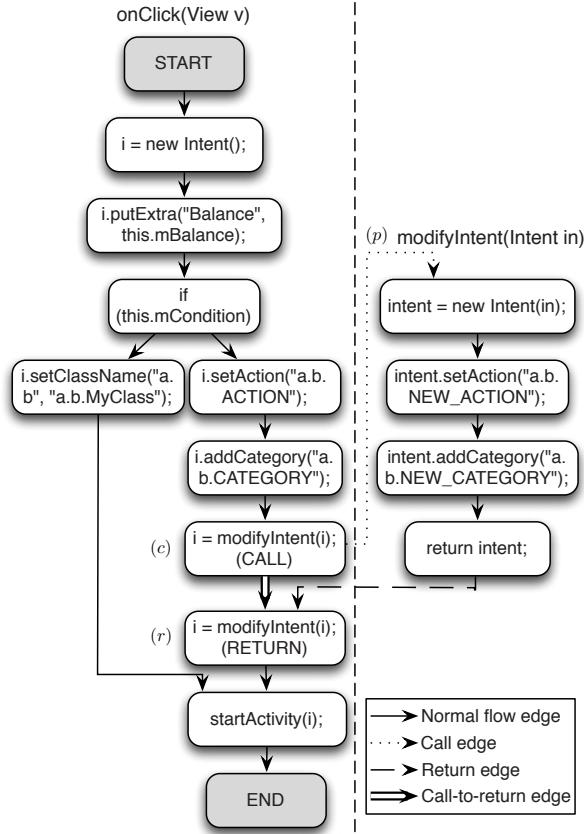


Figure 5: Supergraph G^* for the program from Figure 3

4 The IDE Framework: Background

The main part of our analysis is based on the IDE framework [32]. In this section, we summarize the main ideas and notations of the IDE framework. A complete description is available in [32]. The IDE framework solves a class of interprocedural data flow analysis problems. In these problems, an environment contains information at each program point. For each program idiom, environment transformers are defined and modify the environment according to semantics. The solution to this class of problems can be found efficiently.

4.1 Supergraphs

A program is represented using a *supergraph* G^* . G^* is composed of the control flow graphs of the procedures in the program. Each procedure call site is represented by two nodes, one call node representing control right before the callee is entered and one return-site node to which control flows right after exiting the callee. Figure 5 shows the supergraph of the program in Figure 3.

The nodes of a supergraph are program statements. There are four kinds of edges between these nodes. Given a call to procedure (p) with call node (c) and return-site (r) , three kinds of edges are used to model the effects of the procedure call on the environment:

- A call edge between (c) and the first statement of (p) .
- A return edge between the last statement of (p) and (r) .
- A call-to-return edge between (c) and (r) .

All other edges in the supergraph are normal intraprocedural flow edges. Informally, the call edge transfers symbols and associated values from the calling method to the callee when a symbol of interest is a procedure argument. The return edge transfers information from the return value of the callee to the environment in the calling procedure. Finally, the call-to-return edge propagates data flow information that is not affected by the callee, “in parallel” to the procedure call (e.g., local variables).

4.2 Environment transformers

Let D be a finite set of symbols (e.g., program variables). D contains at least a symbol Λ that represents the absence of a data flow fact. Let $L = (V, \sqcup)$ be a join semilattice with bottom element \perp , where V is a set of values⁴. An *environment* e is a function from D to L . The set of environments from D to L is denoted by $Env(D, L)$.

Operator \sqcup is defined over $Env(D, L)$ as a natural extension of \sqcup in semilattice L : for $e_1, e_2 \in Env(D, L)$, $e_1 \sqcup e_2$ is such that, for all $d \in D$, $(e_1 \sqcup e_2)(d) = e_1(d) \sqcup e_2(d)$.

An *environment transformer* is a function from $Env(D, L)$ to $Env(D, L)$. The algorithms from [32]

⁴A join semilattice is a partially ordered set in which any two elements have a least upper bound.

Constructor $b = \text{new Bundle}()$	Adding int key-value pair $b.putInt("MyInt", mInt)$
$\lambda e.e[b \mapsto \perp]$	$\lambda e.e[b \mapsto \beta_{(\{\text{MyInt}\}, \emptyset, \perp)}^b(e(b))]$
$\begin{array}{ccc} \Lambda & b & d \\ \downarrow & \searrow \lambda B.B & \downarrow \\ \Lambda & b & d \end{array}$	$\begin{array}{ccc} \Lambda & b & d \\ \downarrow & \downarrow \lambda B.B & \downarrow \\ \Lambda & b & \lambda B.\beta_{(\{\text{MyInt}\}, \emptyset, \perp)}^b(B) & d \\ & & \downarrow & \downarrow \lambda B.B \end{array}$
Copy constructor $b = \text{new Bundle}(d)$	Clearing extra data keys $d.clear()$
$\lambda e.e[b \mapsto e(d)]$	$\lambda e.e[d \mapsto \beta_{(\emptyset, \emptyset, 1, \perp)}^b(e(d))]$
$\begin{array}{ccc} \Lambda & b & d \\ \downarrow & \nearrow \lambda B.B & \downarrow \\ \Lambda & b & d \end{array}$	$\begin{array}{ccc} \Lambda & b & d \\ \downarrow & \downarrow \lambda B.B & \downarrow \\ \Lambda & b & \lambda B.\beta_{(\emptyset, \emptyset, 1, \perp)}^b(B) & d \\ & & \downarrow & \downarrow \end{array}$

Figure 6: Pointwise environment transformers for common Bundle operations

require that the environment transformers be *distributive*. An environment transformer t is said to be distributive if for all $e_1, e_2, \dots \in Env(D, L)$, and $d \in D$, $(t(\sqcup_i e_i))(d) = (\sqcup_i t(e_i))(d)$. It is denoted by $t : Env(D, L) \longrightarrow_d Env(D, L)$. Environment transformers have a pointwise representation. We show an example on Figure 6. Given environment $e \in Env(D, L)$, transformer $\lambda e.e$ is the *identity*, which preserves the value of e . Given symbol $b \in D$ and value $B \in L$, $\lambda e.e[b \mapsto B]$ transforms e to an environment where all values are the same as in e , except that symbol b is associated with value B . The functions from L to L (represented next to each arrow in Figure 6) are called *micro-functions*.

The environment transformer for the copy constructor call $b = \text{new Bundle}(d)$ is $\lambda e.e[b \mapsto e(d)]$. It means that the value associated with b after the instruction is the same as d 's value before the instruction. In the pointwise representation, this is symbolized by an arrow between d and b with an identity function next to it.

We are trying to determine the value associated with each symbol at program points of interest, which is done by solving an Interprocedural Distributive Environment (IDE) problem. An instance IDE problem is defined as a tuple (G^*, D, L, M) , where:

- $G^* = (N^*, E^*)$ is the supergraph of the application being studied.
- D is the set of symbols of interest.
- L is a join semilattice (V, \sqcup) with least element \perp .
- M assigns distributive environment transformers to the edges of G^* , i.e. $M : E^* \longrightarrow (Env(D, L) \longrightarrow_d Env(D, L))$.

```

1 public ComponentName
2   makeComponentName() {
3     ComponentName c;
4     if (this.mCondition) {
5       c = new ComponentName("c.d",
6         "a.b.MyClass");
7     } else {
8       c = new ComponentName("c.d",
9         "a.b.MySecondClass"); }
10    return c; }

11 public Bundle makeBundle(Bundle b) {
12   Bundle bundle = new Bundle();
13   bundle.putString("FirstName",
14     this.mFirstName);
15   bundle.putAll(b);
16   bundle.remove("Surname");
17   return bundle; }

18 public void onClick(View v) {
19   Intent intent = new Intent();
20   intent.setComponent(makeComponentName());
21   Bundle b = new Bundle();
22   b.putString("Surname", this.mSurname);
23   intent.putExtras(makeBundle(b));
24   registerMyReceiver();
25   startActivity(intent); }

26 public void registerMyReceiver() {
27   IntentFilter f = new IntentFilter();
28   f.addAction("a.b.ACTION");
29   f.addCategory("a.b.CATEGORY");
30   registerReceiver(new MyReceiver(),
31   f, "a.b.PERMISSION", null); }

```

Figure 7: ICC objects example

Under certain conditions on the representation of micro-functions, an IDE problem can be solved in time $O(ED^3)$ [32]. For example, micro-functions should be applied in constant time. In the model we present in Section 5, we relax some of these constraints but find that the problem can still be solved efficiently in the average case. When the problem is solved, we know the value associated with each symbol at important program points.

5 Reducing Intent ICC to an IDE problem

To solve the Intent ICC problem, we need to model four different kinds of objects. First, ComponentName objects contain a package name and a class name. They can be used by explicit Intents. For example, in method *makeComponentName()* of Figure 7, a ComponentName object can take two different values depending on which branch is executed. In the first branch, it refers to class *a.b.MyClass* from application package *c.d*. In the second one, it refers to class *a.b.MySecondClass*. We want to know the possible return values of *makeComponentName()*.

Second, Bundle objects store data as key-value mappings. Method *makeBundle()* of Figure 7 creates a Bundle and modifies its value. We need to find the possible return values of *makeBundle()*.

Third, Intent objects are the main ICC communica-

tion objects. They contain all the data that is used to start other components. In method `onClick()` of Figure 7, the target class of `intent` is set using the return value of `makeComponentName()`. Its extra data is set to the return value of `makeBundle()`. Finally, a new Activity is started using the newly created Intent. We need to determine the value of `intent` at the `startActivity(intent)` instruction.

Fourth, IntentFilter objects are used for dynamic Broadcast Receivers. In `registerMyReceiver()` on Figure 7, an action and a category are added to IntentFilter f . Then a Broadcast Receiver of type `MyReceiver` (which we assume to be defined) is registered using method `registerReceiver()`. It receives Intents that have action `a.b.ACTION` and category `a.b.CATEGORY` and that originate from applications with permission `a.b.PERMISSION`. We want to determine the arguments to the `registerReceiver()` call. That is, we want to know that f contains action `a.b.ACTION` and category `a.b.CATEGORY`. We also want to know that the type of the Broadcast Receiver is `MyReceiver`.

In this section, we use the notations from Sagiv *et al.* [32] summarized in Section 4. We assume that string method arguments are available. We describe the string analysis used in our implementation in Section 6.

5.1 ComponentName Model

In this section, we introduce the model we use for ComponentName objects. We introduce the notion of a branch ComponentName value. It represents the value that a ComponentName object can take on a single branch, given a single possible string argument value for each method setting the ComponentName’s package and class names, and in the absence of aliasing.

Definition 1. A branch ComponentName value is a tuple $c = (p, k)$, where p is a package name and k is a class name.

In method `makeComponentName()` of Figure 7, two branch ComponentName values are constructed:

$$(c.d, a.b.MyClass) \quad (1)$$

and

$$(c.d, a.b.MySecondClass) \quad (2)$$

The next definition introduces ComponentName values, which represent the possibly multiple values that a ComponentName can have at a program point. A ComponentName can take several values in different cases:

- After traversing different branches, as in method `makeComponentName()` of Figure 7.
- When a string argument can have several values at a method call.
- When an object reference is a possible alias of another local reference or an object field.

- When an object reference is a possible array element.

In the last two cases, in order to account for the possibility of a false positive in the alias analysis, we keep track of two branch ComponentName values. One considers the influence of the call on the possible alias and the other one does not.

Definition 2. A ComponentName value C is a set of branch ComponentName values: $C = \{c_1, c_2, \dots, c_m\}$. The set of ComponentName values is denoted as V_c . We define $\perp = \emptyset$ and \top as the ComponentName value that is the set of all possible branch ComponentName values in the program. The operators \cup and \subseteq are defined as traditional set union and comparison operators: for $C_1, C_2 \in V_c$, $C_1 \subseteq C_2$ iff $C_1 \cup C_2 = C_2$. $L_c = (V_c, \cup)$ is a join semilattice.

Note that given the definitions of \perp and \top as specific sets, \cup and \subseteq naturally apply to them. For example, for all $C \in V_c$, $\top \cup C = \top$.

In method `makeComponentName()` from Figure 7, the value of c at the return statement is

$$\{(c.d, a.b.MyClass), (c.d, a.b.MySecondClass)\}. \quad (3)$$

It simply combines the values of c created in the two branches, given by Equations (1) and (2).

We define transformers from V_c to V_c that represent the influence of a statement or a sequence of statements on a ComponentName value. A pointwise branch ComponentName transformer represents the influence of a single branch, whereas a pointwise ComponentName transformer represents the influence of possibly multiple branches.

Definition 3. A pointwise branch ComponentName transformer is a function $\delta_{(\pi, \chi)}^c : V_c \rightarrow V_c$, where π is a package name and χ is a class name. It is such that, for each $C \in V_c$,

$$\delta_{(\pi, \chi)}^c(C) = \{(\pi, \chi)\}$$

Note that $\delta_{(\pi, \chi)}^c(C)$ is independent of C , because API methods for ComponentName objects systematically replace existing values for package and class names. In the example from Figure 7, the pointwise branch ComponentName transformer corresponding to the first branch is

$$\delta_{(c.d, a.b.MyClass)}^c, \quad (4)$$

and the one for the second branch is

$$\delta_{(c.d, a.b.MySecondClass)}^c. \quad (5)$$

Definition 4. A pointwise ComponentName transformer is a function $\delta_{\{(\pi_1, \chi_1), \dots, (\pi_n, \chi_n)\}}^c : V_c \rightarrow V_c$ such that, for each $C \in V_c$,

$$\delta_{\{(\pi_1, \chi_1), \dots, (\pi_n, \chi_n)\}}^c(C) = \{(\pi_1, \chi_1), \dots, (\pi_n, \chi_n)\}$$

A pointwise ComponentName transformer summa-

rizes the effect of multiple branches (or a single branch with multiple possible string arguments, or with possible aliasing) on a ComponentName value. That is, given the value C of a ComponentName right after statement s_i and given transformer $\delta_{\{(\pi_1, \chi_1), \dots, (\pi_n, \chi_n)\}}^c$ that summarizes the influence of statements s_{i+1}, \dots, s_k on C , $\delta_{\{(\pi_1, \chi_1), \dots, (\pi_n, \chi_n)\}}^c(C)$ represents all the possible values of C right after s_k . In method `makeComponentName()` of Figure 7, the pointwise ComponentName transformer that models the two branches is

$$\delta_{\{(c.d.a.b.MyClass), (c.d.a.b.MySecondClass)\}}^c. \quad (6)$$

It combines the transformers given by Equations (4) and (5). In order to understand how this transformer is applied in practice, we should mention that the algorithm to solve IDE problems initially sets values to \perp [32]. Therefore, in method `makeComponentName()`, the value associated with c is initially $\perp = \emptyset$. Using Definition 4, we can easily see that if we apply the transformer given by Equation (6), we get the value given by Equation (3). This confirms that the transformer models the influence of the two branches:

$$\begin{aligned} \delta_{\{(c.d.a.b.MyClass), (c.d.a.b.MySecondClass)\}}^c(\perp) \\ = \{(c.d.a.b.MyClass), \\ (c.d.a.b.MySecondClass)\} \end{aligned}$$

5.2 Bundle Model

The model of Bundle objects is defined similarly to the model of ComponentName objects. An additional difficulty is introduced. The data in a Bundle can be modified by adding the data in another Bundle to it, as shown in method `makeBundle()` of Figure 7. In this example, the data in Bundle b is added to the data in Bundle $bundle$. Bundle $bundle$ is later modified by removing the key-value pair with key `Surname`. The issue is that when the data flow problem is being tackled, the value of b is not known. Therefore, the influence of the call to `remove("Surname")` is not known: if a key-value pair with key `Surname` is part of b , then the call removes it from $bundle$. Otherwise, it has no influence.

Our approach to deal with this object composition problem is to perform two successive analyses. In Analysis I, we use placeholders for Bundles such as b in instruction `bundle.putAll(b)`. We also record all subsequent method calls affecting $bundle$. After the problem is solved, b 's key-value pairs at the `putAll(b)` method call are known, as well as the subsequent method calls. We then perform Analysis II, in which b 's key-value pairs are added to $bundle$'s. The influence of the subsequent method call is precisely evaluated and finally the value of $bundle$ at the return statement can be known.

5.2.1 Analysis I

In the first analysis, we consider intermediate values that contain “placeholders” for Bundle values that are not known when the problem is being solved.

Definition 5. An intermediate branch Bundle value is a tuple $b_i = (E, O)$, where:

- E is a set of keys describing extra data.
- O is a tuple of two types of elements. O contains references to particular Bundle symbols at instructions where `putAll()` calls occur. O also contains functions from V_b^i to V_b^i , where V_b^i is the set of intermediate Bundle values defined below. These functions represent a sequence of method calls affecting a Bundle.

The difference with previous definitions is the introduction of O , which models calls to `putAll()` as well as subsequent calls affecting the same Bundle. In method `makeBundle()` of Figure 7, at the return statement, the intermediate branch Bundle value associated with $bundle$ is (E, O) , where

$$E = \{\text{FirstName}\} \quad (7)$$

$$O = ((b, \text{bundle.putAll}(b)), \beta_{(\emptyset, \{\text{Surname}\}, 0, ())}^b) \quad (8)$$

In O , $(b, \text{bundle.putAll}(b))$ is a reference to variable b at instruction `bundle.putAll(b)`. $\beta_{(\emptyset, \{\text{Surname}\}, 0, ())}^b$ models the `remove()` method call. It is defined below.

We just defined intermediate branch Bundle values. As we did before, we need to consider multiple branches and related issues (e.g., several possible string values):

Definition 6. An intermediate Bundle value B_i is a set of intermediate branch Bundle values: $B_i = \{b_{i_1}, \dots, b_{i_m}\}$. The set of intermediate Bundle values is V_b^i . We define $\perp = \emptyset$ and \top as the intermediate Bundle value that is the set of all possible intermediate branch Bundle values in the program. We define \subseteq and \cup as natural set comparison and union operators. They are such that, for $B_{i_1}, B_{i_2} \in V_b^i$, $B_{i_1} \subseteq B_{i_2}$ iff $B_{i_1} \cup B_{i_2} = B_{i_2}$. $L_b^i = (V_b^i, \cup)$ is a join semilattice.

In method `makeBundle()` from Figure 7, since there is only a single branch, the intermediate Bundle value associated with $bundle$ at the return statement is $\{(E, O)\}$, where E and O are given by Equations (7) and (8).

Pointwise transformers are defined from V_b^i to V_b^i . Similarly to the ComponentName model, we first introduce pointwise branch Bundle transformers before defining pointwise Bundle transformers. In the definitions below, we use the \ notation for set difference, and \cup is naturally extended to tuples such that $(a_1, \dots, a_k) \cup (a_{k+1}, \dots, a_l) = (a_1, \dots, a_k, a_{k+1}, \dots, a_l)$.

Definition 7. A pointwise branch Bundle transformer is a function $\beta_{(\eta^+, \eta^-, cl, \Theta)}^b : V_b^i \rightarrow V_b^i$, where:

- η^+ is a set of string keys describing extra data. It models calls to `putExtra()` methods.

- η^- is a set of string keys describing removed extra data. It represents the influence of calls to the `removeExtra()` method.
- cl takes value 1 if the Bundle data has been cleared with the `clear()` method and 0 otherwise.
- Θ is a tuple of two types of elements. It contains references to particular Bundle symbols at instructions where `putAll()` calls occur. It also contains functions from V_b^i to V_b^i . These functions represent a sequence of method calls affecting a Bundle.

It is such that

$$\beta_{(\eta^+, \eta^-, cl, \Theta)}^b(\perp) = \{(\eta^+ \setminus \eta^-, \Theta)\}$$

and, for $B_i = \{(E_1, O_1), \dots, (E_m, O_m)\}$ ($B_i \neq \perp$),

$$\beta_{(\eta^+, \eta^-, cl, \Theta)}^b(B_i) = \{(E'_1, O'_1), \dots, (E'_m, O'_m)\}$$

where, for each j from 1 to m :

$$E'_j = \begin{cases} \eta^+ \setminus \eta^- & \text{if } cl = 1 \\ (E_j \cup \eta^+) \setminus \eta^- & \text{if } cl = 0 \text{ and } O_j = \emptyset \\ E_j & \text{otherwise} \end{cases}$$

$$O'_j = \begin{cases} \Theta & \text{if } cl = 1 \text{ or } O_j = \emptyset \\ O_j \cup (\beta_{(\eta^+, \eta^-, 0, \Theta)}^b) & \text{otherwise} \end{cases}$$

The definition of E'_j accounts for several possible cases:

- If the Bundle data has been cleared (i.e., $cl = 1$), then we discard any data contained in E_j . This leads to value $\eta^+ \setminus \eta^-$ for E'_j : we only keep the values η^+ that were added to the Bundle data and remove the values η^- that were removed from it.
- If the Bundle has not been cleared, then there are two possible cases: either no reference to another Bundle has been previously recorded (i.e., $O_j = \emptyset$), or such a reference has been recorded to model a call to `putAll()`. In the first case, we simply take the union of the original set E_j and the set η^+ of added values, and subtract the set η^- of removed values. This explain the $(E_j \cup \eta^+) \setminus \eta^-$ value. In the second case, a call to `putAll()` has been detected, which means that any further method call adding or removing data has to be added to set O_j instead of E_j . Therefore in this case $E'_j = E_j$.

The definition of O'_j considers several cases:

- If the Bundle data has been cleared, then the previous value of O_j is irrelevant and we set $O'_j = \Theta$. Also, if O_j is empty, then we can also just set O'_j to Θ (which may or may not be empty).
- Otherwise, the Bundle data has not been cleared ($cl = 0$) and a call to `putAll()` has been detected ($O_j \neq \emptyset$). Then it means that the current function models method calls that happened after a call to `putAll()`. Therefore we need to record $\beta_{(\eta^+, \eta^-, 0, \Theta)}^b$ in O'_j , which explains the definition $O'_j = O_j \cup (\beta_{(\eta^+, \eta^-, 0, \Theta)}^b)$.

For example, the pointwise branch Bundle transformer that models the influence of the method `makeBundle()` from Figure 7 is $\beta_{(\eta^+, \emptyset, 0, \Theta)}^b$, where

$$\eta^+ = \{\text{FirstName}\} \quad (9)$$

$$\Theta = \left((b, \text{bundle.putAll}(b)), \beta_{(\emptyset, \{\text{Surname}\}, 0, ()})^b \right) \quad (10)$$

Pointwise branch Bundle transformers model the influence of a single branch. In order to account for multiple branches or issues such as possible aliasing false positive, we define pointwise Bundle transformers.

Definition 8. A pointwise Bundle transformer is a function

$$\beta_{\{(\eta_1^+, \eta_1^-, cl_1, \Theta_1), \dots, (\eta_n^+, \eta_n^-, cl_n, \Theta_n)\}}^b : V_b^i \rightarrow V_b^i$$

such that, for each $B_i \in V_b^i$,

$$\beta_{\{(\eta_1^+, \eta_1^-, cl_1, \Theta_1), \dots, (\eta_n^+, \eta_n^-, cl_n, \Theta_n)\}}^b(B_i) = \beta_{(\eta_1^+, \eta_1^-, cl_1, \Theta_1)}^b(B_i) \cup \dots \cup \beta_{(\eta_n^+, \eta_n^-, cl_n, \Theta_n)}^b(B_i)$$

For example, method `makeBundle()` from Figure 7 only has a single branch, thus the pointwise Bundle transformer that models it is simply $\beta_{\{(\eta^+, \emptyset, 0, \Theta)\}}^b$, where η^+ and Θ are given in Equations (9) and (10). As we did for the ComponentName value example, we can confirm using Definitions 7 and 8 that $\beta_{\{(\eta^+, \emptyset, 0, \Theta)\}}^b(\perp) = \{(E, O)\}$, where E and O are given by Equations (7) and (8).

5.2.2 Analysis II

After Analysis I has been performed, the values of the Bundles used in placeholders in intermediate Bundle values are known. Ultimately, we want to obtain branch Bundle values and finally Bundle values:

Definition 9. A branch Bundle value b is a set E of string keys describing extra data.

Definition 10. A Bundle value B is a set of branch Bundle values: $B = \{b_1, \dots, b_m\}$.

Since the values of the referenced Bundles are known, we can integrate them into the Bundle values referring to them. Then the influence of the subsequent method calls that have been recorded can precisely be known.

Let us consider the example of `makeBundle()` from Figure 7. After Analysis I has been performed, we know that the intermediate value of `bundle` at the return statement is $\{(E, O)\}$, where

$$E = \{\text{FirstName}\}$$

$$O = \left((b, \text{bundle.putAll}(b)), \beta_{(\emptyset, \{\text{Surname}\}, 0, ())}^b \right)$$

We consider all elements of O in order. As the first element of O is $(b, \text{bundle.putAll}(b))$, we integrate b 's value into `bundle`. From Analysis I, we know

that the value of b at instruction `bundle.putAll(b)` is $\{\{\text{Surname}\}, \emptyset\}$. Thus, E becomes $\{\text{FirstName}, \text{Surname}\}$. The next element of O is $\beta_{(\emptyset, \{\text{Surname}\}, 0, ()^*)}^b$. This means that we have to remove key `Surname` from E . The final value of E is therefore $\{\text{FirstName}\}$. Thus, the `Bundle` value associated with `bundle` at the return statement is $\{\{\text{FirstName}\}\}$.

Note that the referenced `Bundle` can also make references to other `Bundles`. In that case, we perform the resolution for the referenced `Bundles` first. There can be an arbitrary number of levels of indirection. Analysis II is iterated until a fix-point is reached.

5.3 Intent and IntentFilter Models

The Intent model is defined similarly to the `Bundle` model, which includes object composition. In method `onClick()` of Figure 7, the target of Intent `intent` is set using a `ComponentName` object and its extra data is set with a `Bundle`. Because of this object composition, finding the Intent value also involves two analyses similar to the ones performed for `Bundles`. First, intermediate Intent values with placeholders for referenced `ComponentName` and `Bundle` objects are found. Second, the referenced objects' values are integrated into `intent`'s value.

Similarly to the `Bundle` model, we define intermediate branch Intent values and intermediate Intent values. The set of intermediate Intent values is V_i^i and we define a lattice $L_i^i = (V_i^i, \cup)$ as we did for L_b^i . We also define pointwise branch Intent transformers and pointwise Intent transformers. For example, in method `onClick()` of Figure 7, the final intermediate value for `intent` simply has placeholders for a `ComponentName` and a `Bundle` value. Other fields, such as action and categories, are empty. The `ComponentName` and `Bundle` values are computed using the models presented in Sections 5.1 and 5.2. Finally, we define branch Intent values and Intent values, which are output by the second analysis. The final value for `intent` after the second analysis precisely contains the two possible targets (`a.b.MyClass` and `a.b.MySecondClass` in package `c.d`) and extra data key `FirstName`. For conciseness, and given the strong similarities with the `Bundle` model, we do not include a full description of the Intent model here.

In order to analyze dynamic Broadcast Receivers, we model `IntentFilter` objects. Modeling `IntentFilters` is similar to modeling `Intents`, except that `IntentFilters` do not involve object composition. That is because `IntentFilters` do not have methods taking other `IntentFilters` as argument, except for a copy constructor. Thus, their analysis is simpler and involves a single step. Similarly to what we did for other ICC models, we define branch `IntentFilter` values, `IntentFilter` values, pointwise branch `IntentFilter` transformers and pointwise `IntentFilter` trans-

formers. In particular, we define lattice $L_f = (V_f, \cup)$, where V_f is the set of `IntentFilter` values. In method `onClick()` from Figure 7, the final value of f contains action `a.b.ACTION` and category `a.b.CATEGORY`. Given the similarity of the `IntentFilter` model with previous models, we do not include a complete description.

5.4 Casting as an IDE Problem

These definitions allow us to define environment transformers for our problem. Given environment $e \in Env(D, L)$, environment transformer $\lambda e.e$ is the *identity*, which does not change the value of e . Given Intent i and Intent value I , $\lambda e.e[i \mapsto I]$ transforms e to an environment where all values are the same as in e , except that Intent i is associated with value I .

We define an environment transformer for each API method call. Each of these environment transformers uses the pointwise environment transformers defined in Sections 5.1, 5.2 and 5.3. It precisely describes the influence of a method call on the value associated with each of the symbols in D .

Figure 6 shows some environment transformers and their pointwise representation. The first one is a constructor invocation, which sets the value corresponding to b to \perp . The second one adds an integer to the key-value pairs in `Bundle` b 's extra data, which is represented by environment transformer

$$\lambda e.e[b \mapsto \beta_{(\{\text{MyInt}\}, \emptyset, 0, ())}^b(e(b))].$$

It means that the environment stays the same, except that the value associated with b becomes $\beta_{(\{\text{MyInt}\}, \emptyset, 0, ())}^b(e(b))$, with $e(b)$ being the value previously associated with b in environment e . The pointwise transformer for b is

$$\beta_{(\{\text{MyInt}\}, \emptyset, 0, ())}^b,$$

which we denote by

$$\lambda B.\beta_{(\{\text{MyInt}\}, \emptyset, 0, ())}^B(B)$$

on Figure 6 for consistency with the other pointwise transformers. It simply adds key `MyInt` to the set of data keys. The next transformer is for a copy constructor, where the value associated with d is assigned to the value associated with b . The last transformer clears the data keys associated with d .

Trivially, these environment transformers are distributive. Therefore, the following proposition holds.

Proposition 1. Let G^* be the supergraph of an Android application. Let D_c , D_b , D_i and D_f be the sets of `ComponentName`, `Bundle` and `Intent` variables, respectively, to which we add the special symbol Λ^5 . Let L_c , L_b^i , L_i^i and L_f be the lattices defined above.

⁵Recall from Section 4.2 that Λ symbolizes the absence of a data flow fact.

Let M_c , M_b , M_i and M_f be the corresponding assignments of distributive environment transformers. Then (G^*, D_c, L_c, M_c) , (G^*, D_b, L_b^i, M_b^i) , (G^*, D_b, L_i^i, M_i^i) and (G^*, D_i, L_f, M_f) are IDE problems.

It follows from this proposition that we can use the algorithm from [32] to solve the Intent ICC problem.

The original IDE framework [32] requires that the micro-function be represented efficiently in order to achieve the time complexity of $O(ED^3)$. Our model does not meet these requirements: in particular, applying, composing, joining micro-function or testing for equality of micro-functions cannot be done in constant time. Indeed, the size of micro-functions grows with the number of branches, aliases and possible string arguments (see Equation 6 for an example with two branches). However, in practice we can find solutions to our IDE problem instances in reasonable time, as we show in Section 6.

6 Evaluation

This section describes an evaluation of the approach presented in the preceding sections, and briefly characterizes the use of ICC in Android applications. We also present a study of potential ICC vulnerabilities. Our implementation is called Epicc (Efficient and Precise ICC) and is available at <http://siis.cse.psu.edu/epicc/>. It is built on Heros [3], an IDE framework within Soot [34]. We also provide the version of Soot that we modified to handle pathological cases encountered with retargeted code.

In order to compute string arguments, we use a simple analysis traversing the interprocedural control flow graph of the application. The traversal starts at the call site and looks for constant assignments to the call arguments. If a string argument cannot be determined, we conservatively assume that the argument can be any string. As we show in Section 6.1, in many cases we are able to find precise string arguments. More complex analyses can be used if more precision is desired [7].

For points-to analysis and call graph construction, we use Spark [24], which is part of Soot. It performs a flow-sensitive, context-insensitive analysis. We approximate the call graph in components with multiple entry points. In order to generate a call graph of an Android application, we currently use a “wrapper” as an entry point. This wrapper calls each class entry point once, which may under-approximate what happens at runtime. This impacts a specification only if an ICC field (e.g., Intent) is modified in a way that depends on the runtime execution order of class entry points. Generally, if we assume that our model of components’ life cycle is complete and if the application does not use native calls or reflection, then our results are sound.

The analysis presented in this section is performed on two datasets. The first *random sample* dataset contains 350 applications, 348 of which were successfully analyzed after retargeting. They were extracted from the Google Play store⁶ between September 2012 and January 2013. The applications were selected at random from over 200,000 applications in our corpus. The second *popular application* dataset contains the top 25 most popular free applications from each of the 34 application categories in the Play store. The 850 selected applications were downloaded from that application store on January 30, 2013. Of those 850 applications, 838 could be retargeted and processed and were used in the experiments below. The 14 applications which were not analyzed were pathological cases where retargeting yielded code which could not be analyzed (e.g., in some cases the Dare tool generated offsets with integer overflow errors due to excessive method sizes), or where applications could not be processed by Soot (e.g., character encoding problems).

6.1 Precision of ICC Specifications

The first set of tests evaluates the technique’s *precision* with our datasets. We define the precision metric to be the percentage of source and sink locations for which a specification is identified without ambiguity. Ambiguity occurs when an ICC API method argument cannot be determined. These arguments are mainly strings of characters, which may be generated at runtime. In some cases, runtime context determines string values, which implies that our analysis cannot statically find them.

Recall the various forms of ICC. Explicit ICC identifies the communication sink by specifying the target’s package and class name. Conversely, implicit ICC identifies the sink through action, category, and/or data fields. Further, a mixed ICC occurs when a source or sink can take on explicit or implicit ICC values depending on the runtime context. Finally, the dynamic receiver ICC occurs when a sink binds to an ICC type through runtime context (e.g., Broadcast Receivers which identify the Intent Filter types when being registered). We seek to determine precise ICC specifications, where all fields of Intents or Intent Filters are known without ambiguity.

As shown in Table 1, with respect to the random sample corpus, we were able to provide unambiguous specifications for over 91% of the 7,835 ICC locations in the 348 applications. Explicit ICC was precisely analyzed more frequently ($\approx 98\%$) than implicit ICC ($\approx 88\%$). The remaining 7% of ICC containing mixed and dynamic receivers proved to be more difficult, where the precision rates are much lower than others. This is likely due to the fact that dynamic receivers involve finding more data

⁶Available at <https://play.google.com/store/apps>.

Random Sample					
	Precise	%	Imprecise	%	Total
Explicit	3,571	97.65%	86	2.35%	3,657
Implicit	3,225	88.45%	421	11.55%	3,646
Mixed	28	59.57%	19	40.43%	47
Dyn. Rec.	357	73.61%	128	26.39%	485
Total	7,181	91.65%	654	8.35%	7,835

Popular					
	Precise	%	Imprecise	%	Total
Explicit	27,753	94.43%	1,637	5.57%	29,390
Implicit	23,133	93.82%	1,525	6.18%	24,658
Mixed	509	85.12%	89	14.88%	598
Dyn. Rec.	4,161	95.81%	182	4.19%	4,343
Total	55,556	94.18%	3,433	5.82%	58,989

Table 1: Precision metrics

than Intents: Intent Filters limiting access to dynamic receivers can define several actions, and receivers can be protected by a permission (which we attempt to recover).

In the popular applications, we obtain a precise specification in over 94% of the 58,989 ICC locations in the 838 apps. Explicit ICC was slightly more precisely analyzed than implicit ICC. Mixed ICC is again hard to recover. This is not surprising, as mixed ICC involves different Intent values on two or more branches, which is indicative of a method more complex than most others.

A facet of the analysis not shown in the table is the number of applications for which we could identify unambiguous specifications for all ICC – called 100% precision. In the random sample, 56% of the applications could be analyzed with 100% precision, 80% of the applications with 90% precision, and 91% of the applications with 80% precision. In the popular applications, 23% could be analyzed with 100% precision, 82% could be analyzed with 90% precision and 94% with 80% precision. Note that a less-than-100% precision does not mean that the analysis failed. Rather, these are cases where runtime context determines string arguments, and thus *any* static analysis technique would fail.

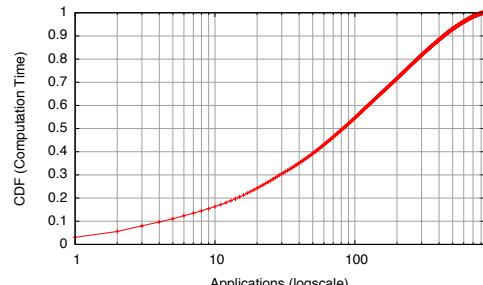
6.2 Computation Costs

A second set of tests sought to ascertain the computational costs of performing the IDE analysis using Epicc. For this task we collected measurements at each stage of the analysis and computed simple statistics characterizing the costs of each task on the random sample and the popular applications.

Experiment results show that ICC analysis in this model is feasible for applications in the Google Play store. We were able to perform analysis of all 348 applications in the random sample in about 3.69 hours of compute time. On average, it took just over 38 seconds to perform analysis for a single application, with a standard deviation of 99 seconds. There was high variance in the analysis run times. A CDF (cumulative distribution



(a) Random sample



(b) Popular applications

Figure 8: CDF of computation time

function) of the analysis computation time for all 348 applications is presented in Figure 8(a). It is clear from the figure that costs were dominated by a handful of applications; the top application consumed over 11% of the time, the top 5 consumed over 25% of the total time, and the top 29 consumed over 50% of the total time. These applications are large with a high number of entry points.

Analyzing the 838 popular applications took 33.58 hours, that is, 144 seconds per application. The standard deviation was 277 seconds. The average processing time is significantly higher than for the random sample. However, this is expected, as the average application size is almost 1,500 classes, which is significantly higher than the random sample (less than 400 classes per application). This is likely related to the popularity bias: one can expect frequently downloaded applications to have fully developed features as well as more complex/numerous features, which implies a larger code base. A CDF of the computation time for all 838 applications is presented in Figure 8(b). Once again, analysis time is dominated by a few applications. The top 5 consumed over 11% of the analysis time and the top 83 (less than 10% of the sample) consumed over 50% of the analysis time.

Processing was dominated by the standard Soot processing (e.g., translating classes to an intermediate representation, performing type inference and points-to analysis, building a call graph). It consumed 75% of the processing time in the random sample and 86% in the popular applications. It was itself dominated by the translation to Soot’s internal representation and by the call graph construction. The second most time-consuming task was the IDE analysis (which also includes the string analysis in our implementation). It took 15% of the pro-

cessing time with the random sample and 7% with the popular one. Finally, I/O operations accounted for most of the remainder of the processing time. Loading classes took 7% of the time in the random sample and 3% in the popular one. Database operations accounted for 2% of processing for the random sample and 3% for the popular applications. Other operations (e.g., parsing manifest files) took less than 1% of processing time.

6.3 Entry/Exit Point Analysis

This section briefly characterizes the exit (source) and entry (sink) points of Android applications in our data sets. Note that this analysis is preliminary and will be extended in future work.

An exit point is a location that serves as a source for ICC; i.e., the sending of an Intent. In the random sample, our analysis found 7,350 exit points which can transmit 10,035 unique Intent values. About 92% of these exit points had a single Intent specification, with the remaining exit points being able to take on 2 or more values. In two pathological cases, we noted an exit point that could have 640 different Intent values (most likely the result of contrived control flow or multiple aliasing for an Intent value). The popular applications had 48,756 exit points, associated with 316,419 Intent values. Single Intent specifications were found in 90% of exit points. We found 10 pathological cases where an exit point was associated with 512 Intent values or more. The use of key value data was more prevalent than we initially expected, in about 36% of exit points in the random sample. Key-value data was present in Intents in 46% of exit points in the popular applications.

Our study of entry points focused on the sinks of ICC that were either dynamically registered broadcast receivers or component interfaces (exported or not) identified in the application manifest. In the random sample, we were able to identify 3,863 such entry points associated with 1,222 unique intent filters. The popular applications comprised 25,291 entry points with 11,375 Intent Filters. 1,174 components were exported (and thus available to other applications) in the random sample, 7,392 in the popular applications. Of those, only 6% (67) of the exported components were protected by a permission in the random sample and 5% (382) were protected in the popular applications. This is concerning, since the presence of unprotected components in privileged applications can lead to confused deputy [21] attacks [17].

Oddly, we also found 23 components that were exported without any Intent Filter in the random sample and 220 in the popular sample. Conversely, we found 32 cases where a component had an Intent Filter but was not exported in the random sample and 412 in the popular one. The latter indicates that developers sometimes use

implicit Intents to address components within an application, which is a potential security concern, since these Intents may also be intercepted by other components. Lastly, application entry points were relatively narrow (with respect to intent types). Over 97% of the entry points received one Intent type in the random sample. Single Intent Filters were found in 94% of components protected by Intent Filters in the popular applications.

6.4 ICC Vulnerability Study

In this section, we perform a study of ICC vulnerabilities in our samples using Epicc and compare our results with ComDroid [6]. We look for the same seven vulnerabilities as in [6]. Activity and Service hijacking can occur when an Intent is sent to start an Activity or a Service without a specific target. Broadcast thefts can happen when an Intent is Broadcast without being protected by a signature or signatureOrSystem permission⁷. In all three cases, the Intent may be received by a malicious component, along with its potentially sensitive data.

Malicious Activity or Service launch and Broadcast injection are Intent spoofing vulnerabilities. They indicate that a public component is not protected with a signature or signatureOrSystem permission. It may be started by malicious components. These vulnerabilities can lead to permission leakage [17, 19, 25].

Finally, some Intent Broadcasts can only be sent by the operating system, as indicated by their action field. Broadcast Receivers can register to receive them by specifying Intent Filters with the appropriate action. However, these public components can still be addressed directly by explicit Intents. That is why the target Receivers should check the action field of the received Intent to make sure that it was sent by the system.

Table 2 shows the results of the study for the random and the popular samples. The first line shows the number of vulnerabilities identically detected by both analyses, the second line shows vulnerabilities detected by ComDroid only and the third line shows vulnerabilities detected by Epicc only. The last two lines show the total number of vulnerabilities found by each tool. For the three unauthorized Intent receipt vulnerabilities (first three columns), both ComDroid and Epicc indicate whether the sent Intent has extra data in the form of key-value pairs, and whether the Intent has the FLAG_GRANT_READ_URI_PERMISSION or the FLAG_GRANT_WRITE_URI_PERMISSION. These flags are used in Intents which refer to Content Provider data and may allow the recipient to read or write the data [6].

⁷The signature permission protection level only allows access to a component from an application signed by the same developer. The signatureOrSystem protection level additionally allows the operating system to start the component.

Vulnerability	Activity Hijacking		Service Hijacking		Broadcast Theft		Activity Launch		Service Launch		Broadcast Injection		System Broadcast w/o action check		Total vulnerabilities		
Sample	R	P	R	P	R	P	R	P	R	P	R	P	R	P	R	P	R
Identical	2,591	15,214	78	1,200	503	4,825	179	1,731	23	263	273	3,503	30	126	3,677	26,862	
ComDroid only	916	7,717	78	535	218	2,854	12	169	2	18	104	1,684	3	20	1,333	12,997	
Epicc only	181	2,079	3	151	23	297	4	20	0	1	4	43	77	580	292	3,171	
Total ComDroid	3,507	22,931	156	1,735	721	7,679	191	1,900	25	281	377	5,187	33	146	5,010	39,859	
Total Epicc	2,772	17,293	81	1,351	526	5,122	183	1,751	23	264	277	3,546	107	706	3,969	30,033	

Table 2: ICC vulnerability study results for the random sample (R) and the popular applications (P)

For the presence of flags and the detection of extra data, Epicc can precisely indicate when the value of an Intent depends on the execution path. On the other hand, a ComDroid specification does not make this distinction. When Epicc and ComDroid differ for a code location, we include flags in both the “ComDroid only” and “Epicc only” rows of Table 2.

The Activity hijacking vulnerabilities found by both ComDroid and Epicc are unsurprisingly common: they represent all cases where implicit Intents are used to start Activities. Service hijacking vulnerabilities are much less prevalent, which is correlated with the fact that Services are used less often than Activities. Broadcast theft vulnerabilities are quite common as well. As previously described in Section 6.3, few exported components are protected by permissions. Therefore, the high number of malicious Activity or Service launch as well as Broadcast injection vulnerabilities is not surprising. Note the discrepancy between the number of components without permissions and the total number of these vulnerabilities. A large portion of the components not protected by permissions are Activities with the `android.intent.action.MAIN` action and the `android.intent.category.LAUNCHER` category, which indicate that these components cannot be started without direct user intervention. They are therefore not counted as potential vulnerabilities.

If we consider the first three vulnerabilities (unauthorized Intent receipt), we can see that ComDroid flags a high number of locations where Epicc differs. A manual examination of a random subset of applications shows that these differences are either false positives detected by ComDroid or cases where Epicc gives a more precise vulnerability specification. We observed that a number of code locations are detected as vulnerable by ComDroid, whereas Soot does not find them to be reachable. Epicc takes advantage from the sound and precise Soot call graph construction to output fewer false positives. Additionally, the IDE model used by Epicc can accurately keep track of differences between branches (e.g., explicit/implicit Intent or URI flags), whereas ComDroid cannot. Note that when an Intent is implicit on one branch and explicit on another, ComDroid detects it as explicit, which is a false negative. On the other hand, the IDE model correctly keeps track of the possibilities.

With a few exceptions, the ComDroid and Epicc analyses detect the same possible malicious Activity and Service launches. That is expected, since both are detected by simply parsing the manifest file. The few differences can be explained by minor implementation differences or bugs in pathological cases. The Broadcast injection vulnerability shows stronger differences, with ComDroid detecting 377 cases for the random sample and 5,187 for the popular one, whereas Epicc only finds 277 and 3,546, respectively. Some of the Broadcast injections detected by ComDroid involved dynamically registered Broadcast Receivers found in unreachable code. Once again, the call graph used by Epicc proves to be an advantage. Many other cases involve Receivers listening to protected system Broadcasts (i.e., they are protected by Intent Filters that only receive Intents sent by the system). The list of protected Broadcasts used by ComDroid is outdated, hence the false positives.

Finally, there is a significant difference in the detection of the system Broadcasts without action check, with Epicc detecting 107 vulnerabilities in the random sample and 706 in the popular one, whereas ComDroid only detects 33 and 146, respectively. The first reason for that difference is that the ComDroid list of protected Broadcasts is outdated. Another reason is an edge case, where the Soot type inference determines Receivers registered using a `registerReceiver()` method as having type `android.content.BroadcastReceiver` (i.e., the abstract superclass of all Receivers). It occurs when several types of Receivers can reach the call to `registerReceiver()`. Since no Receiver code can be inspected, even though there may be a vulnerability, our analysis conservatively flags it as a vulnerability.

Overall, Epicc detects 34,002 potential vulnerabilities. On the other hand, ComDroid detects 44,869 potential security issues, that is, 32% more than Epicc. As detailed above, the extra flags found by ComDroid that we checked were all false positives. Further, the potential causes of unsoundness in Epicc (i.e., JNI, reflection and entry point handling) are also handled unsoundly in ComDroid. Thus, we do not expect the locations flagged by ComDroid but not by Epicc to be false negatives. The precision gain over ComDroid is significant and will help further analyses. Note that it is possible that both tools have false negatives in the presence of JNI, reflection,

or when the life cycle is not properly approximated. In particular, we found that 776 out of the 838 popular applications and 237 out of 348 applications in the random sample make reflective calls. Future work will seek to quantify how often these cause false negatives in practice. We will also attempt to determine if the locations flagged by Epicc are true positives.

7 Related Work

ComDroid [6] is the work most closely related to ours. Our work aims to formalize the notions it first captured. It is different in many aspects. First, ComDroid directly analyses Dalvik bytecode, whereas we use retargeted Java bytecode. This allows us to leverage analyses integrated with Soot (e.g., call graph). Also, unlike ComDroid, our analysis is fully interprocedural and context-sensitive. Second, our ICC model is sound and more detailed, taking multiple branches and aliasing into account. Thus, as shown in Section 6.4, our ICC vulnerability study produces fewer false positives. Finally, ComDroid seeks to find potential vulnerabilities, whereas our approach enables finding attacks for vulnerabilities in existing applications. This is done by keeping a database of analysis results and matching newly analyzed applications with applications in our database. This will allow us to identify problematic application combinations.

Several kinds of application analysis have been performed for the Android platform [10]. Permission analysis infers applications properties based on the permissions requested at install time. Kirin [13] uses permissions to flag applications with potential dangerous functionality. Other methods for permission analysis have been proposed [2, 15, 16], including analyses to detect over-privileged applications [15] or malware [36].

Dynamic analysis consists in analyzing applications while they are running. TaintDroid [11] performs dynamic taint tracking on Android. It exposes widespread leakage of personal data to third parties. An extension to TaintDroid handles implicit flows [18] by monitoring and recording control flow information. TaintDroid is also used in the AppFence system [22], which actively prevents sensitive data exfiltration from mobile devices. Alternative approaches dynamically prevent some classes of privilege escalation attack through ICC [4, 9]. Dynamic analyses such as TaintDroid are limited by the way they interact with the User Interface (UI). SmartDroid [35] tackles this issue by combining static and dynamic analyses. It is able to simulate the UI to expose hidden behavior for seven malwares. As we use static analysis we do not interact with the UI: the call graph is complete and does not depend on any runtime condition.

Static analysis consists in analyzing application code to infer useful properties without running the applica-

tion. Several approaches for static analysis have already been proposed for Android applications. Enck *et al.* use decompilation [28] followed by source code analysis to characterize security properties of applications [12]. Grace *et al.* perform a study of the dangers caused by 100 ad libraries found in a sample of 100,000 applications [20] through a reachability analysis on disassembled bytecode. Several analyses have statically found permission leaks [17, 19, 25], which happen when a privileged application leaks its capabilities to unprivileged ones. These analyses focus on finding paths between exposed entry points and sensitive API calls, whereas we focus on connecting exit points to entry points. Thus, these analyses could benefit from our ICC analysis.

ScanDal [23] attempts to soundly analyze information flow. It converts Dalvik bytecode to a formally defined intermediate language. Dangerous flows are detected using abstract interpretation. Its analysis is path-insensitive and has limited context-sensitivity. It finds some actual privacy leaks, but is limited by a high number of false positives and flows that are impossible to confirm.

Saint [30] modifies the Android framework to control application interaction. Every application comes with a policy describing how it uses permissions it declares. Policy compliance verification is a possible application of our tool but is out of the scope of this paper.

8 Conclusion

In this paper we have introduced an efficient and sound technique for inferring ICC specifications, and demonstrated its feasibility on a large collection of market applications. Future work will study a range of applications and analyses that exploit the database of ICC specifications. We will also explore a range of extensions that can use this information at runtime to identify potentially malicious communication between applications. Through these activities, we aim to aid the community’s efforts to gauge the security of market applications.

Acknowledgements

We thank Matthew Dering for providing our application samples. We also thank Atul Prakash, Patrick Traynor and our shepherd Ben Livshits for editorial comments during the writing of this paper. This material is based upon work supported by the National Science Foundation Grants No. CNS-1228700, CNS-0905447, CNS-1064944 and CNS-0643907. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation. This research is also supported by a Google Faculty Award.

References

- [1] ARTHUR, C. Feature phones dwindle as android powers ahead in third quarter. *The Guardian*, Nov. 2012. Available at <http://www.guardian.co.uk/technology/2012/nov/15/smartphones-market-android-feature-phones>.
- [2] BARRERA, D., KAYACIK, H. G., VAN OORSHOT, P. C., AND SOMAYAJI, A. A Methodology for Empirical Analysis of Permission-Based Security Models and its Application to Android. In *Proceedings of the ACM Conference on Computer and Communications Security* (Oct. 2010).
- [3] BODDEN, E. Inter-procedural data-flow analysis with ifds/ide and soot. In *Proceedings of the ACM SIGPLAN International Workshop on State of the Art in Java Program analysis* (2012). Available from <http://sable.github.com/heros/>.
- [4] BUGIEL, S., DAVI, L., DMITRIENKO, A., FISCHER, T., AND SADEGHI, A.-R. XManDroid: A New Android Evolution to Mitigate Privilege Escalation Attacks. Tech. Rep. TR-2011-04, Technische Universität Darmstadt, Germany, Apr. 2011.
- [5] BUGIEL, S., DAVI, L., DMITRIENKO, A., FISCHER, T., SADEGHI, A.-R., AND SHAstry, B. Towards taming privilege-escalation attacks on Android. In *Proceedings of the 19th Annual Network & Distributed System Security Symposium* (Feb. 2012).
- [6] CHIN, E., FELT, A. P., GREENWOOD, K., AND WAGNER, D. Analyzing Inter-Application Communication in Android. In *Proceedings of the 9th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys)* (2011).
- [7] CHRISTENSEN, A. S., MØLLER, A., AND SCHWARTZBACH, M. I. Precise analysis of string expressions. In *Proc. 10th International Static Analysis Symposium (SAS)* (June 2003), vol. 2694 of LNCS, Springer-Verlag, pp. 1–18. Available from <http://www.brics.dk/JSA/>.
- [8] DAVI, L., DMITRIENKO, A., SADEGHI, A.-R., AND WINANDY, M. Privilege Escalation Attacks on Android. In *Proc. of the 13th Information Security Conference (ISC)* (Oct. 2010).
- [9] DIETZ, M., SHEKHAR, S., PISETSKY, Y., SHU, A., AND WAL-LACH, D. S. Quire: Lightweight Provenance for Smart Phone Operating Systems. In *20th USENIX Security Symposium* (2011).
- [10] ENCK, W. Defending users against smartphone apps: Techniques and future directions. In *ICISS* (2011), pp. 49–70.
- [11] ENCK, W., GILBERT, P., CHUN, B.-G., COX, L. P., JUNG, J., McDANIEL, P., AND SHETH, A. N. TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones. In *Proc. of the 9th USENIX Symp. on Operating Systems Design and Implementation (OSDI)* (2010).
- [12] ENCK, W., OCTEAU, D., McDANIEL, P., AND CHAUDHURI, S. A Study of Android Application Security. In *Proceedings of the 20th USENIX Security Symposium* (August 2011).
- [13] ENCK, W., ONGTANG, M., AND McDANIEL, P. On Lightweight Mobile Phone Application Certification. In *Proceedings of the 16th ACM Conference on Computer and Communications Security (CCS)* (Nov. 2009).
- [14] ENCK, W., ONGTANG, M., AND McDANIEL, P. Understanding Android Security. *IEEE Security & Privacy Magazine* 7, 1 (January/February 2009), 50–57.
- [15] FELT, A. P., CHIN, E., HANNA, S., SONG, D., AND WAGNER, D. Android Permissions Demystified. In *Proc. of the ACM Conf. on Computer and Communications Security (CCS)* (2011).
- [16] FELT, A. P., GREENWOOD, K., AND WAGNER, D. The Effectiveness of Application Permissions. In *Proc. of the USENIX Conference on Web Application Development (WebApps)* (2011).
- [17] FELT, A. P., WANG, H. J., MOSHCHUK, A., HANNA, S., AND CHIN, E. Permission Re-Delegation: Attacks and Defenses. In *Proc. of the 20th USENIX Security Symp.* (August 2011).
- [18] GILBERT, P., CHUN, B.-G., COX, L. P., AND JUNG, J. Vision: Automated Security Validation of Mobile Apps at App Markets. In *Proceedings of the International Workshop on Mobile Cloud Computing and Services (MCS)* (2011).
- [19] GRACE, M., ZHOU, Y., WANG, Z., AND JIANG, X. Systematic Detection of Capability Leaks in Stock Android Smartphones. In *NDSS '12* (2012).
- [20] GRACE, M. C., ZHOU, W., JIANG, X., AND SADEGHI, A.-R. Unsafe exposure analysis of mobile in-app advertisements. In *Proceedings of the fifth ACM conference on Security and Privacy in Wireless and Mobile Networks* (2012), WISEC '12, ACM.
- [21] HARDY, N. The confused deputy: (or why capabilities might have been invented). *SIGOPS Oper. Syst. Rev.* 22, 4 (Oct. 1988).
- [22] HORNYACK, P., HAN, S., JUNG, J., SCHECHTER, S., AND WETHERALL, D. These Aren't the Droids You're Looking For: Retrofitting Android to Protect Data from Imperious Applications. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)* (2011).
- [23] KIM, J., YOON, Y., AND YI, K. Scandal: Static analyzer for detecting privacy leaks in android applications. In *MoST 2012: Workshop on Mobile Security Technologies 2012* (2012).
- [24] LHOTÁK, O., AND HENDREN, L. Scaling java points-to analysis using spark. In *Proceedings of the 12th international conference on Compiler construction* (2003), CC'03, Springer-Verlag.
- [25] LU, L., LI, Z., WU, Z., LEE, W., AND JIANG, G. Chex: statically vetting android apps for component hijacking vulnerabilities. In *Proc. of the 2012 ACM conference on Computer and communications security* (2012), CCS '12, ACM, pp. 229–240.
- [26] McDANIEL, P., AND ENCK, W. Not So Great Expectations: Why Application Markets Haven't Failed Security. *IEEE Security & Privacy Magazine* 8, 5 (September/October 2010), 76–78.
- [27] MLLOT, S. Google's bouncer malware tool hacked. *PC Magazine*, June 2012. Available from <http://www.pcmag.com/article2/0,2817,2405358,00.asp>.
- [28] OCTEAU, D., ENCK, W., AND McDANIEL, P. The ded Decompile. Tech. Rep. NAS-TR-0140-2010, Network and Security Research Center, Pennsylvania State University, USA, Sept. 2010. Available from <http://siis.cse.psu.edu/ded/>.
- [29] OCTEAU, D., JHA, S., AND McDANIEL, P. Retargeting android applications to java bytecode. In *Proceedings of the 20th International Symposium on the Foundations of Software Engineering* (November 2012). Available from <http://siis.cse.psu.edu/dare/>.
- [30] ONGTANG, M., McLAUGHLIN, S., ENCK, W., AND McDANIEL, P. Semantically Rich Application-Centric Security in Android. In *Proceedings of the 25th Annual Computer Security Applications Conference (ACSAC)* (Dec. 2009), pp. 340–349.
- [31] ROSENBERG, J. Google play hits 25 billion downloads. *Android - Official blog*, Sept. 2012. Available at <http://officialandroid.blogspot.com/2012/09/google-play-hits-25-billion-downloads.html>.
- [32] SAGIV, M., REPS, T., AND HORWITZ, S. Precise interprocedural dataflow analysis with applications to constant propagation. *Theor. Comput. Sci.* 167, 1-2 (Oct. 1996), 131–170.
- [33] SECURITY, N. Malware controls 620,000 phones, sends costly messages. *Help Net Security*, January 2013. Available from http://www.net-security.org/malware_news.php?id=2391.
- [34] VALLÉE-RAI, R., GAGNON, E., HENDREN, L. J., LAM, P., POMINVILLE, P., AND SUNDARESAN, V. Optimizing java bytecode using the soot framework: Is it feasible? In *Proc. of the 9th International Conf. on Compiler Construction* (2000), CC '00.
- [35] ZHENG, C., ZHU, S., DALI, S., GU, G., GONG, X., HAN, X., AND ZOU, W. Smartdroid: an automatic system for revealing ui-based trigger conditions in android applications. In *Proceedings of the second ACM workshop on Security and privacy in smartphones and mobile devices* (2012), ACM, pp. 93–104.
- [36] ZHOU, Y., WANG, Z., ZHOU, W., AND JIANG, X. 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* (Feb. 2012).

Jekyll* on iOS: When Benign Apps Become Evil

Tielei Wang, Kangjie Lu, Long Lu, Simon Chung, and Wenke Lee

School of Computer Science, College of Computing, Georgia Institute of Technology

{tielei.wang, kangjie.lu, long, pchung, wenke}@cc.gatech.edu

Abstract

Apple adopts the mandatory app review and code signing mechanisms to ensure that only approved apps can run on iOS devices. In this paper, we present a novel attack method that fundamentally defeats both mechanisms. Our method allows attackers to reliably hide malicious behavior that would otherwise get their app rejected by the Apple review process. Once the app passes the review and is installed on an end user’s device, it can be instructed to carry out the intended attacks.

The key idea is to make the apps remotely exploitable and subsequently introduce malicious control flows by rearranging signed code. Since the new control flows do not exist during the app review process, such apps, namely *Jekyll apps*, can stay undetected when reviewed and easily obtain Apple’s approval.

We implemented a proof-of-concept Jekyll app and successfully published it in App Store. We remotely launched the attacks on a controlled group of devices that installed the app. The result shows that, despite running inside the iOS sandbox, Jekyll app can successfully perform many malicious tasks, such as stealthily posting tweets, taking photos, stealing device identity information, sending email and SMS, attacking other apps, and even exploiting kernel vulnerabilities.

1 Introduction

Apple iOS is one of the most popular and advanced operating systems for mobile devices. By the end of June 2012, Apple had sold 400 million iOS devices [30], such as iPhone, iPad and iPod touch. Despite the tremendous popularity, in the history of iOS, only a handful of malicious apps have been discovered [24]. This is mainly attributed to the advanced security architecture of iOS and the strict regulations of the App Store.

*Jekyll is a character with dual personalities from the novel *The Strange Case of Dr. Jekyll and Mr. Hyde*.

In addition to the standard security features like Address Space Layout Randomization (ASLR), Data Execution Prevention (DEP), and Sandboxing, iOS enforces the mandatory App Review and code signing mechanisms [31]. App Review inspects every app submitted by third parties (in binary form) and only allows it to enter the App Store if it does not violate App Store’s regulations [5]. To further prohibit apps distributed through channels other than the App Store (i.e., unsigned apps), the code signing mechanism disallows unsigned code from running on iOS devices. As a result, all third-party apps running on iOS devices (excluding jailbroken devices [48]) have to be approved by Apple and cannot be modified after they have obtained the approval.

According to the official App Review guidelines [5], developers should expect their apps to go through a thorough inspection for all possible term violations. During this process, many reasons can lead to app rejections, such as stealing data from users and using private APIs reserved for system apps. Although the technical details of the review process remain largely unknown, it is widely believed that such a selective and centralized app distribution model has significantly increased the difficulty and cost for malicious or ill-intended apps to reach end users.

In this paper, we present a new attack method against the App Store reviewing process and the code signing mechanism. Using this method, attackers can create malicious or term-violating apps and still be able to publish them on App Store, which in turn open up new attack surfaces on iOS devices. We stress that our attack does not assume any specifics about how Apple reviews apps, but targets theoretical difficulties faced by any known methods to analyze programs. By demonstrating the power of this practical attack, we highlight the shortcomings of the pre-release review approach and call for more runtime monitoring mechanisms to protect iOS users in the future.

The key idea behind our attack is that, instead of sub-

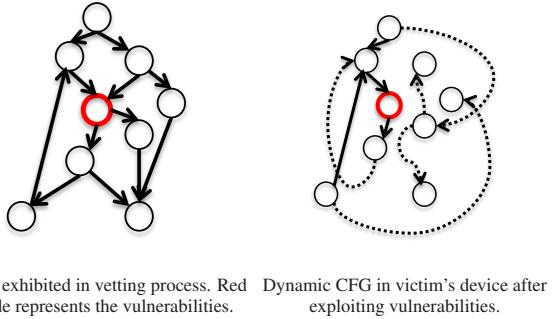


Figure 1: High Level Intuition

mitting an app that explicitly contains malicious functionalities to Apple, the attacker plants remotely exploitable vulnerabilities (i.e., backdoor) in a normal app, decomposes the malicious logic into small code gadgets and hides them under the cover of the legitimate functionalities. After the app passes the App Review and lands on the end user device, the attacker can remotely exploit the planted vulnerabilities and assemble the malicious logic at runtime by chaining the code gadgets together.

Figure 1 shows the high level idea. On the left is the app’s original control flow graph (CFG), which is what can be observed during the app review process, without the planted vulnerability being exploited. In comparison, on the right is the effective control flow graph the same app will exhibit during runtime, which differs from the left in the new program paths (represented by the dotted paths) introduced at runtime by the remote attackers (i.e., app developers). Since attackers can construct malicious functionalities through dynamically introducing new execution paths, even if the vetting process could check all possible paths in the left CFG (i.e., 100% path coverage), it cannot discover the malicious logic that is only to be assembled at runtime as per attacker’s commands. Apps so constructed bear benign looks and yet are capable of carrying out malicious logic when instructed; we call them Jekyll apps. By carefully designing the vulnerabilities and crafting the gadgets, Jekyll apps can reliably pass app review process and open up a new attack surface on iOS devices when installed. Specifically, an attacker can achieve the following general tasks via Jekyll apps:

First, Jekyll apps offer an approach to stealthily abuse user privacy and device resources, for instance, via private APIs¹, which may provide unrestricted access to certain sensitive resources and are intended for Apple’s internal use only. Explicit use of private APIs almost al-

ways gets an app rejected by App Store [4]. However, Jekyll apps can dynamically load, locate, and implicitly invoke the private APIs and thus reliably bypass the review checks. Comparing with simple obfuscation techniques (e.g., [7, 23, 25]), our approach hides the usage of private APIs in a way that is more resilient to non-trivial code analysis — without correctly triggering the planted vulnerabilities and arranging the code gadgets, the invocation of private APIs never appears in the code and execution of Jekyll apps.

Second, Jekyll apps open a window for attackers to exploit vulnerabilities in kernel space. Although the sandboxing policy in iOS limits the possibility and impact of exploiting kernel vulnerabilities [22] by third-party apps, certain attacks are still effective against vulnerable device drivers (i.e., IOKit drivers [49]).

Third, Jekyll apps also serve as a trampoline to attack other apps. On iOS, by requesting a URL, an app can launch another app that has registered to handle that URL scheme. However, this simplified IPC (Inter-process communication) mechanism may facilitate inter-app attacks. For instance, once new vulnerabilities have been found in Mobile Safari (the built-in web browser in iOS), an attacker can set up a malicious webpage exploiting such vulnerabilities, use the Jekyll app to direct the Mobile Safari to visit the booby-trapped website, and eventually compromise the browser app. Given the high privileges granted to Mobile Safari, the compromised browser will in turn provide the stepping stone for more powerful attacks, such as untethered jailbreak, as shown by the JailbreakMe attack [1] on old versions of iOS.

Attack Type	Attack Description	Affected Version
Abuse Device Resources	Sending SMS	iOS 5.x
	Sending Email	iOS 5.x
	Posting Tweet	iOS 5.x & 6.x
	Abusing Camera	iOS 5.x & 6.x
	Dialing	iOS 5.x & 6.x
	Manipulating Bluetooth	iOS 5.x & 6.x
	Stealing Device Info	iOS 5.x & 6.x
Attack Kernel	Rebooting system	iOS 5.x
Attack Other Apps	Crashing Mobile Safari	iOS 5.x & i6.x

Table 1: Attack summary on iPhone

We have implemented a proof-of-concept Jekyll app and submitted it to the App Store. The app successfully passed Apple’s review despite the hidden vulnerabilities and code gadgets that can be assembled to carry out malicious logic. Following the ethical hacking practice, we immediately removed the app from App Store once a group of experiment devices of our control had downloaded it. The download statistic provided by Apple later confirmed that the app had never been downloaded by any other users. By exploiting the vulnerabilities and chaining the planted gadgets in the app, we

¹Private APIs are undocumented and often security-critical APIs on iOS, see Section 2.2 for details.

remotely launched many malicious operations on our experiment devices, as summarized in Table 1. Even on iOS 6.1.2, the latest version of iOS at the time of our experiments, the Jekyll app can abuse the camera device to recode videos, post tweets, steal device identity information such as IMEI (the unique device identifier), manipulate the bluetooth device, attack Mobile Safari, and dial arbitrary number. We made a full disclosure of our attack scheme to Apple in March 2013 and have since been in correspondence with Apple.

In summary, the main contributions of our work are as follows:

- We propose a novel method to generate iOS apps that can pass App Review and synthesize new control flows as instructed remotely during runtime, without violating code signing. We call such malicious apps Jekyll apps. Given that arbitrary control flows can be introduced to such apps at runtime, the code signing mechanism on iOS is totally defenseless against Jekyll apps.
- We are the first to propose a dynamic analysis technique to discover the private APIs used to post tweets, send email, and send SMS without user’s consent on iOS. We incorporate these attacks, along with a set of previously known iOS attacks, into a Jekyll app to show its versatility.
- We successfully publish a proof-of-concept Jekyll app in Apple App Store and later launch remote attacks to a controlled group.
- We demonstrate that the security strategy to solely rely on pre-install review, as currently followed by Apple App Store, is ineffective against Jekyll apps and similar attacks. We discuss and advocate runtime security measures as a necessary step in advancing iOS security.

The rest of the paper is organized as follows. Section 2 introduces the background. Section 3 presents a motivating example and describes the design of our attack scheme. Section 4 demonstrates some of the malicious operations that can be carried out by Jekyll apps. Section 5 gives the implementation details and Section 6 compares our research to related work. Section 7 discusses the potential countermeasures against our attack and Section 8 concludes the paper.

2 Background

2.1 iOS Security

iOS provides a rich set of security features. We briefly introduce the related exploit mitigation mechanisms here.

Interested readers are referred to [31, 38] for the overall security architecture of iOS.

DEP and ASLR. Apple introduced the Data Execution Prevention (DEP) mechanism in iOS 2.0 and later the Address Space Layout Randomization (ASLR) mechanism in iOS 4.3 [21]. The DEP mechanism in iOS is based on the NX (eXecute Never) bit supported by the ARM architecture and the kernel prevents third party apps from requesting memory pages that are writeable and executable at the same time. Since data pages such as the stack and heap are marked non-executable and code pages are marked executable but non-writeable, DEP prevents the traditional code injection attacks that need to write payloads into memory and execute them.

ASLR randomizes a process’s memory layout. If a third-party app is compiled as a position-independent executable (PIE), the locations of all memory regions in its process’s address space, including the main executable, dynamic libraries, stack, and heap, are unpredictable. As an important complementary to DEP, ASLR makes it very difficult for attackers to launch return-to-libc based or return-oriented programming based attacks (see Section 2.3). However, ASLR in iOS only enforces the module level randomization, that is, executable modules are loaded into unpredictable memory regions, but the internal layout of each module remains unchanged. Thus, the ASLR implementation is vulnerable to information leakage vulnerabilities [45]. If an attacker can obtain the absolute address of a function in a module, she is able to infer the memory layout of that entire module.

Privilege Separation and Sandboxing. iOS employs traditional UNIX file permission mechanisms to manage the file system and achieve the basic privilege separation. While all third-party apps run as the non-privileged user `mobile`, only a few most import system processes run as the privileged user `root`. As a result, third-party apps are not able to change system configurations.

To enforce isolation among apps that all run as the same user `mobile`, iOS utilizes the sandboxing mechanism. iOS sandbox is implemented as a policy module in the TrustedBSD mandatory access control framework [8]. Each app contains a plist file in XML format, which declares a set of entitlements for the special capabilities or security permissions in iOS. When an app is launched, iOS determines its sandbox policy according to its entitlements.

Although the built-in apps in iOS, such as Mobile Safari, run as the non-privileged user `mobile`, they may be granted with special privileges via reserved entitlements. For instance, Mobile Safari has an entitlement called `dynamic-codesigning`, which allows Mobile Safari to allocate a writable and executable memory buffer and generate executable code on the fly—a security exception made for Mobile Safari’s Just-in-Time

(JIT) JavaScript engine to improve performance.

As for third-party apps, Apple applies a one-size-fits-all sandbox policy called `container`. According to the study in [51], in iOS 4.3, this permissive policy allows third-party apps to read the user’s media library, interact with a few IOKit User Clients, communicate with the local Mach RPC servers over the bootstrap port, access the network, etc. On top of the default access granted by the `container` policy, third party apps can also request for two extra entitlements: one for using the iCloud storage and one for subscribing to the push notification service. Finally, even though the `container` policy has undergone significant improvements and is becoming more restrictive over time, as we show in this paper, our Jekyll app, even running in sandbox, still poses a significant threat to the user’s privacy and system security.

Also, in contrast to other mobile platforms, such as Android, which use the declarative permissions to regulate each app individually, iOS applies *the default sandbox configuration on most third-party apps*, which consequently share the same broad set of privileges. As of iOS 6, only a few sensitive operations, such as accessing location information and contact book and sending push notifications, have to be explicitly acknowledged by users before they can proceed.

Code signing, App Store, and App Review. Along with the release of iOS 2.0 in 2008, Apple opened the App Store, an application distribution platform for iOS devices. Third-party developers are required to submit their apps to App Store for distribution. Since then, iOS has enforced the mandatory code signing mechanism to ensure only the executables that have been approved and signed by Apple are allowed to run on iOS devices. The study in [37] presents the implementation details of iOS code signing mechanism. In comparison with DEP, code signing mechanism is more strict. In a DEP-enabled system, attackers can compromise a process using ROP attacks and then download a new binary and run it. This does not apply to iOS because iOS will refuse to run the new binary if it is not signed by a trusted authority.

To release an app through App Store, a third-party developer has to participate in Apple’s iOS developer program and submit the app to Apple for review. The app is signed and published by Apple only after it passes the review process. In addition to business benefits, the mandatory review process helps Apple prevent malicious apps from entering App Store.

2.2 Public and Private Frameworks

iOS provides the implementation of its system interfaces in special packages called frameworks. A framework is a directory that contains a dynamic shared library and the related resources such as images, localization strings,

and header files. Native iOS apps are built on top of these frameworks and written in the Objective-C programming language, a superset of C language.

Besides the public frameworks, iOS also contains a set of private frameworks that are not allowed to be used in third-party apps. Even in public frameworks, there are some undocumented APIs (i.e., private APIs) that cannot be used by third-party apps. In fact, these private frameworks and APIs are reserved for the built-in apps and public frameworks. Apple ships all public and private frameworks as part of the iOS Software Development Kit (SDK). Third-party developers can find all these frameworks in their own development environment. It is worth noting that, since iOS 3.x, Apple has combined all frameworks into a single cache file called `dyld_shared_cache` in iOS devices to improve performance [21].

Moreover, the creation of dynamic libraries by third-party developers is not supported by the iOS SDK, which makes the public frameworks the only shared libraries to link in iOS apps. To prevent apps from dynamically loading private frameworks or unofficial libraries, some standard UNIX APIs are also considered as private by Apple, such as `dlopen` and `dlsym` that support runtime loading of libraries. During the app review process, linking to private frameworks or importing private APIs can directly result in app rejections from Apple App Store.

2.3 Code Reuse and ROP Attack

Reusing the code within the original program is an effective way to bypass DEP and code signing mechanism. Solar Designer first suggested return-to-libc [16], which reuses existing functions in a vulnerable program to implement attacks. Shacham et al. proposed the Return-Oriented Programming (ROP) exploitation technique in 2007 [44]. The core idea behind ROP attacks is to utilize a large number of instruction sequences ending with ret-like instructions (e.g., `ret` on x86 and `pop{pc}` on ARM) in the original program or other libraries to perform certain computation. Since attackers can control the data on the stack and ret-like instructions will change the execution flow according to the data on the stack, a crafted stack layout can chain these instruction sequences together. Figure 2 shows a simple ROP example that performs addition and storage operations on the ARM platform. Specifically, constant values `0xdeadbeaf` and `0xffffffffffff` are loaded to the registers `r1` and `r2` by the first two gadgets, respectively. Next, an addition operation is performed by the third gadget. At last, the addition result (`0xdeadbeae`) is stored on the stack by the fourth gadget.

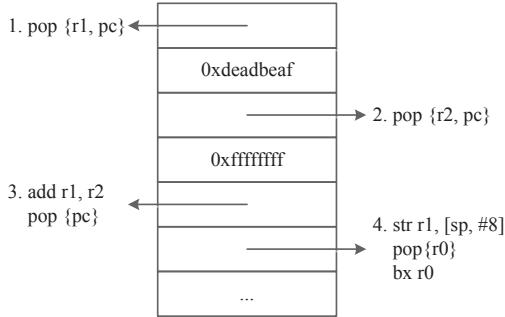


Figure 2: A ROP example

3 Attack Design

Before introducing the design of our attack scheme, we first discuss an example attack, which demonstrates the feasibility of such attacks and helps illustrate the design details in the rest of this section.

3.1 Motivating Example

Suppose the attacker’s goal is to steal the user’s contacts. To this end, the attacker first creates a normal app, a greeting card app for instance, which can download greeting cards from a remote server and then send them to the user’s friends. The pseudo code in Figure 3 presents the workflow of the app, which requires access to user’s address book and the network for legitimate reasons. However, direct abuses of these privileges to send the whole address book over the network can be easily detected. In fact, multiple systems (e.g., [17–19, 26]) have been proposed to detect malicious apps by identifying code paths or execution traces where sensitive data is first acquired and then transported out of the device, and we assume the app review process will also be able to detect and reject such apps.

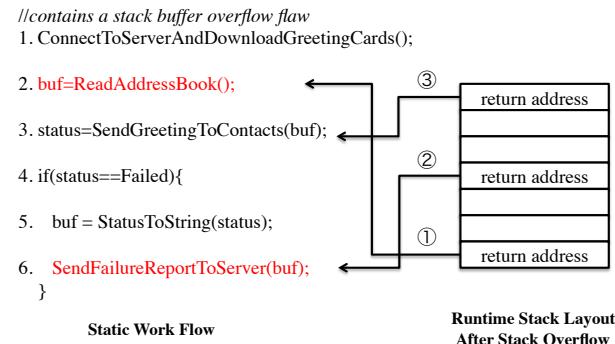


Figure 3: Running Example

However, our example app (as shown in Figure 3) does not contain any feasible code path to leak the address book after reading it at line 2. As such, our example app appears to be compliant with Apple’s privacy policy and can be expected to pass the app review.

To achieve the goal of stealing the user’s contact while avoiding the direct approach that will guarantee rejection by App Store, the attacker instead hides vulnerabilities in the `ConnectToServerAndDownloadGreetingCards` function (line 1 in Figure 3). Subsequently, when the app runs on a victim’s iOS device and tries to download greeting cards from the server controlled by the attacker, the server exploits the planted vulnerabilities to remotely manipulate the app’s stack into the one shown on the right side of Figure 3. The contaminated stack layout will change the original control flows of the app. Instead of sequentially executing the statements from line 2 to line 6, the compromised app first reads the address book into a buffer (line 2 in Figure 3), and then directly invokes the `SendFailureReportToServer` function at line 6 to send the content of the buffer (i.e., address book) to the server. Finally, the app resumes the normal execution by returning the control back to line 3. Note that the attacker will avoid revealing the above behavior to Apple and only exploit the vulnerabilities after the app has passed the app review.

Malicious developers can freely design the vulnerabilities to bootstrap the attacks. For instance, the app can deliberately leak its memory layout information to the remote server so that ASLR is completely ineffective. Based on the memory layout information, attackers can launch attacks by reusing the existing code inside the app. As a result, DEP and code signing cannot prevent the exploit. Furthermore, by using iOS private APIs, attackers can accomplish more sophisticated attacks, even though the app runs in the sandbox. In other words, once the app gets installed, existing security mechanisms on iOS will be of no defense against the attack.

3.2 Attack Scheme Overview

The high level idea of our attack scheme is very intuitive. The attacker creates a normal app in which he plants vulnerabilities and hides code gadgets along side the normal functionalities. After the app passes Apple’s app review and gets installed on victims’ devices, the attacker exploits the vulnerabilities and assembles the gadgets in a particular order to perform malicious operations.

For our attack to be successful, the planted vulnerabilities should allow us to defeat the ASLR, DEP, and code signing mechanisms in iOS, and at the same time be hardly detectable. To this end, we design an information leakage vulnerability through which the app deliberately leaks its partial runtime memory layout informa-

tion to the remote attacker. Thus, the attacker can infer the locations of the pre-deployed gadgets, making ASLR useless. Next, we plant a buffer overflow vulnerability in the app through which the attacker can smash the stack layout and hijack the app’s control flow. The carefully designed stack layout will chain together the gadgets to accomplish malicious tasks.

To avoid the vulnerabilities from being detected in the review process, the communication between the app and the server is encrypted, and all the vulnerabilities have special trigger conditions. Considering the fact that no source code but only the executable is provided to the review process, even if advanced vulnerability detection technologies like fuzz testing and dynamic symbolic execution are employed, it is unlikely for app review process to discover artificially planted and obscured vulnerabilities.

Finally, the hidden gadgets should be discretely distributed in the app and mingled with the normal functionalities, without explicit control flow or and data flow connections. To do this, we create a number of infeasible branches across the entire code space and hide gadgets under these infeasible branches. In addition, we organize the common operations useful for both legitimate and malicious functionalities into individual functional gadgets.

3.3 Bypassing ASLR via Information Leakage

The ASLR mechanism loads the app executable and other dynamic libraries at different random locations for each run, and this causes some difficulties in the process of chaining up our gadgets. However, since native apps are written in Objective-C, it is very easy to plant information leakage vulnerabilities to bypass ASLR and recover the addresses of our gadgets. In the following, we present two examples of how this can be achieved.

First, we can take advantage of an out-of-bounds memory access vulnerability to read a function pointer, and then send the value back to the remote server. Specifically, we can use a C code snippet similar to Figure 4. In this case, the app assigns the address of a public function to the function pointer in a C structure, and pretends to transmit the user name to the server. However, the server can control the size parameter of the function `memcpy` and is able to accurately trigger an out-of-bounds read. As a result, the address of the public function is leaked. Based on this address, we can infer the memory layout of corresponding executable file.

Alternatively, we can take advantage of type confusion vulnerabilities and features of Objective-C objects to leak address information. Most objects in Objective-C programs inherit from a common class called `NSObject`. The first field of these objects points

```
struct userInfo{
    char username[16];
    void* (*printName)(char*);
} user;
...
user.printName = publicFunction;
...
n = attacker_controllable_value; //20
memcpy(buf, user.username, n); //get function ptr
SendToServer(buf);
```

Figure 4: Information Disclosure Vulnerability I

to a `Class` structure that stores information about the object’s type, inheritance hierarchy, member methods, etc. These `Class` structures follow the same naming convention (i.e., a common prefix `_objc_class_$_`) and are stored at fixed offsets in the executable files. Using this information, we can also infer the address information of the entire executable file. Figure 5 demonstrates how this method works. First, we create an Objective-C object with the `myObject` pointer pointing to the object. After that, we convert `myObject` into an integer pointer by using explicit type-casting. Finally, by dereferencing the integer pointer, we copy the address value of the `Class` structure into the variable `UID`, and send it to the remote server.

```
//create an object
SomeClass* myObject = [[SomeClass alloc] init];
...
int UID = *(int*)myObject; //type confusion
...
SendToServer(UID);
```

Figure 5: Information Disclosure Vulnerability II

Since many of the malicious operations in Table 1 rely on private APIs, some discussion on how we invoke private APIs in our attack is in order. To this end, we need to be able to dynamically load private frameworks and locate private APIs, and we employ two special APIs, `dlopen()` and `dlsym()`. `dlopen()` is used to load and link a dynamic library specified by filename and return an opaque handle for the library. `dlsym()` is used to get the address of a symbol from a handle returned from `dlopen()`. These two functions are implemented in a library named `libdyld.dylib`. Since there is no evidence to show that the exported APIs in this library can be used by third-party apps, we should avoid directly referencing to any APIs in this library.

Fortunately, we find that both APIs are commonly used by public frameworks due to the need for dynamically loading shared libraries and obtaining the absolute addresses of symbols in the libraries. In particular, in order to support PIE (Position Independent Executable),

public frameworks invoke imported APIs through trampoline functions. The trampoline functions here consist of a short sequence of instructions that first load the absolute address of a specific function from an indirect symbol table and then jump to that address. The indirect symbol table is initially set up by the linker at runtime. Therefore, if we can identify the trampolines for `dlopen` and `dlsym` in a public framework, our app can use the trampolines to indirectly invoke `dlopen` and `dlsym`.

The task of identifying usable trampolines is simple. With the help of a debugger, we set function breakpoints at `dlopen` and `dlsym` and run a test app on a physical device. When the debug session hits a breakpoint, we examine the call stack to find out the trampoline function and its relative offset to the beginning of the module. Thanks to the fact that ASLR on iOS work at the granularity of modules, we can always infer the addresses of these trampolines from the address of a public function in the same module leaked by our Jekyll app using the vulnerabilities described before. Finally, we note that trampolines for `dlopen` and `dlsym` can be found in many essential frameworks, such as `UIKit` and `CoreGraphics`.

3.4 Introducing New Execution Paths via Control-Flow Hijacking

A key design of our attack scheme is to dynamically introduce new execution paths that do not exist in the original app to perform the malicious operations. In order to achieve this, we plant a vulnerability in the Jekyll app, through which we can corrupt data on the stack and overwrite a function return address (or a function pointer). When the function returns, instead of returning to the original call site, the execution will proceed to a program point that is specified by the altered return address on the stack. Although iOS employs the Stack-Smashing Protector method to detect stack-based overflows, we can accurately overwrite the function return address without breaking the stack canary.

```
void vulnerableFoo(int i, int j) {
    int buf[16];
    ...
    if(fakeChecks(i)) {
        buf[i] = j; //overwrite return address
    ...
    return;
}
```

Figure 6: Control Flow Hijacking Vulnerability

Specifically, we use an out-of-bounds write vulnerability as shown in Figure 6 to hijack the control flow. In this case, both `i` and `j` are controlled by the attacker.

Variable `i` is used to index a local integer array. Since the offset from the starting address of this local array to the memory slot for the function’s return address is fixed, a carefully crafted `i` can overwrite the return address via an array element assignment without breaking the stack canary [10]. We can also add fake boundary checks on `i` in the function to prevent the vulnerability from being easily detected. The new return address stored in `j` points to a gadget that shifts the stack frame to a memory region storing data supplied by the attacker. After that, the new stack layout will chain the gadgets together. By using the existing code in the app, we can defeat DEP and code signing. Since our method for introducing new execution paths is essentially return-oriented-programming, interested readers are referred to [15] and [33] for the details of ROP on the ARM platform.

3.5 Hiding Gadgets

In traditional ROP attack scenarios, attackers have to search for usable gadgets from existing binary or libraries using the Galileo algorithm [44]. However, in our case, the attacker is also the app developer, who can freely construct and hide all necessary gadgets, either at the basic block or function level. This advantage makes our attacks significantly less difficult and more practical to launch than ROP attacks.

For the common functional units (such as converting a `char*` to `NSString` and invoking a function pointer), which are useful for both malicious and legit operations of the app, we implement them in individual functions. As a result, we can simply reuse such functions in our attack based on the return-to-libc like exploitation technique. For the special gadgets that are not easily found in existing code, we manually construct them by using ARM inline assembly code [32] and hide them in infeasible branches. In our Jekyll app, we have planted and hidden all gadgets that are required by traditional ROP attacks [15], such as memory operations, data processing (i.e., data moving among registers and arithmetic/logical operations), and indirect function calls.

To create the infeasible branches, we use the opaque constant technique [34]. For instance, in Figure 7 we set a variable to a non-zero constant value derived from a complicated calculation, and perform a fake check on that variable. Since the compiler cannot statically determine that the variable holds a constant value, it will generate code for both branches. As a result, we can reliably embed the gadgets using similar techniques.

Finally, we will conclude this section with a concrete example of our ROP attack. Figure 8 shows the original source code for dialing attack (see Section 4.2), which loads a framework into process memory, locates a private API called `CTCallDial` in the framework, and fi-

```

int i = Opaque_constant_calculation();
if(i == 0)
{ //hide a gadget in this branch
    asm volatile(
        "pop {r2}"
        "bx r2"
    );
}

```

Figure 7: Hide an indirect call gadget

nally invokes that function. Accomplishing the equivalent functionality through the ROP technique is very easy, because many function level gadgets are available in our Jekyll app. Specifically, we can find trampolines for `dlopen` and `dlsym` in public frameworks (see Section 3.3), and can also reuse existing code in our Jekyll app to implement the indirect call and the conversion from `char*` to `NSString` (the argument type of the function `CTCallDial` is `NSString`).

```

1. void* h = dlopen("CoreTelephony", 1);
2. void (*CTCallDial)(NSString*)=dlsym(h, "CTC-
   allDial");
3. CTCallDial(@"111-222-3333");

```

Figure 8: Attack code for dialing

In addition to these function level gadgets, we also utilize a few simple basic block level gadgets that are used to prepare and pass function arguments, recover the stack pointer, and transfer the control back to the normal execution. For example, the first four arguments of a function on iOS are passed through the registers R0–R3. Before jumping into the target function, we can use a gadget like `pop{r0,r1,pc}` to set up the function’s parameters. Such block level gadgets are ubiquitous in the existing code.

4 Malicious Operations

In this section, we introduce the malicious operations we can perform using Jekyll apps. We present how to post tweets and send email and SMS without the user’s knowledge in Section 4.1, describe more private APIs based attacks in Section 4.2, and demonstrate Jekyll app’s ability to exploit kernel vulnerabilities and attack other apps in Section 4.3 and Section 4.4.

4.1 Under the Hood: Posting Tweets and Sending Email and SMS

Since iOS 5.0, third-party apps are allowed to send Twitter requests on behalf of the user, by using the public APIs in a framework called Twitter. After setting the

initial text and other content of a tweet, the public API called by the app will present a tweet view to the user, and let the user decide whether to post it or not, as shown in Figure 9. However, we find that the tweet view in Figure 9 can be bypassed by using private APIs, i.e., our app can post tweets without the user’s knowledge. Next, we describe how we discover the private APIs needed for achieving this goal.



Figure 9: The default UI for a tweet view

Our intuition is that if we know the event handling function that is responsible for the “Send” button click event, our app can directly invoke that function to post the tweet, without the need to present the tweet view to the user.

To do this, we created a simple app that uses the Twitter framework to post tweets, and run the app in the debug model. We developed a dynamic analysis tool based on LLDB, a scriptable debugger in the iOS SDK, to log the function invocation sequence after the “Send” button is clicked. In the following, we will present some details about our tool.

In Objective-C, all object method invocations are dispatched through a generic message handling function called `objc_msgSend`. A method invocation expression in Objective-C like `[object methodFoo:arg0]` will be converted into a C function call expression like

```
objc_msgSend(object, "methodFoo:", arg0).
```

Moreover, iOS follows the ARM standard calling convention. The first four arguments of a function are passed through the registers R0–R3, and any additional arguments are passed through the stack. For the C function expression above, the arguments will be passed as follows: R0 stores `object`, R1 stores the starting address of the method name (i.e., “`methodFoo:`”), and R2 stores `arg0`.

Our dynamic analysis tool sets a conditional breakpoint at the `objc_msgSend` function. When the breakpoint is triggered after the user clicks the “Send” button, the tool logs the call stack, gets the target method name through the register R1, and retrieves the type information of the target object and other arguments (stored in the registers R0, R2 and R3) by inspecting their Class structures (see Section 3.3).

According to the information in the log, we can easily identify the relevant Objective-C classes and private APIs for posting tweets. For instance, in iOS 6.x, we find that a tweet is composed through the method “`setStatus:`” in a class called `SLTwitterStatus`, and then is posted through the method “`sendStatus:completion:`” in a class called `SLTwitterSession`. Our Jekyll app will dynamically load the Twitter framework, create instances from these classes, and invoke private APIs to post tweets without the user’s knowledge.

We also extended the idea to find critical private APIs for sending email and SMS. As in the case of posting Tweets, third-party apps are able to set the initial text and other content of an email or SMS, and present the email or SMS view to the user. In iOS 5.x, we successfully implemented the code to send email and SMS without the user’s knowledge. Specifically, we find that an email is first composed by a method of the class `MessageWriter`, and then is sent to a service process via an inter-process communication (IPC) interface `CPDistributedMessagingCenter`. Eventually, the service process will send the email out. In the case of sending SMS, we find that, the content of an SMS is first converted into an XPC message, and the XPC message is subsequently passed to an XPC service (another kind of IPC interfaces in iOS) named `com.apple.chatkit.clientcomposeserver.xpc`. By using such private APIs, our Jekyll app is able to compose email and SMS objects, pass them to the corresponding service processes, and automatically send them without the user’s knowledge. An independent study simultaneously reported how to send SMS in this manner; interested readers are referred to [20] for details.

However, in iOS 6, Apple introduced a new concept called remote view to enhance the security of email and SMS services. Specifically, a third-party app only passes the initial content of an email or SMS to the corresponding system services. These system service processes will then generate the message view, and let the user make further changes and final decision. Since the message view runs in a separate process, the third-party app is no longer able to invoke the handler function for the “Send” button click event.

4.2 Camera, Bluetooth, Device ID, and Dialing

The iOS developer community has accumulated extensive knowledge of using private APIs and proposed many attacks against jailbroken iOS devices. We integrated some previously known attacks into our Jekyll app. Since these attacks heavily use private APIs, any app that explicitly launches these attacks will most certainly be re-

jected by Apple. However, our Jekyll app can dynamically load the private frameworks and hide the invocations to private APIs, and successfully passes the App Review.

Next, we briefly introduce the private APIs that we utilized to achieve the following tasks without alerting the users: take photos, switch on/off bluetooth, steal the device identity information, and dial arbitrary numbers. The operations in this subsection work in both iOS 5.x and iOS 6.x.

- **Abuse cameras.** Our Jekyll app is able to stealthily turn on the camera in iOS devices to record videos without the user’s knowledge; this can be achieved by creating and assembling the object instances of a set of classes such as `AVCaptureDeviceInput` and `AVCaptureVideoDataOutput` in the `AVFoundation` framework. Jekyll app can also extract every frame of a video stream and transfer the images back to the server.
- **Switch Bluetooth.** By using the APIs in a private framework `BluetoothManager`, our Jekyll app can directly manipulate the Bluetooth device, such as turning it on or off.
- **Steal Device Identity.** To obtain the device identity information, we take advantage of a private function called `CTServerConnectionCopyMobileEquipmentInfo` in the `CoreTelephony` framework. This function can return the device’s the International Mobile Station Equipment Identity (IMEI), the International Mobile Subscriber Identity (IMSI), and the Integrated Circuit Card Identity (ICCID).
- **Dial.** By invoking the private API `CTCallDial` in the `CoreTelephony` framework, our Jekyll app can dial arbitrary numbers. Note that, this API supports to dial not only phone numbers, but also GSM service codes [3] as well as carrier-specific numbers. For instance, by dialing `*21*number#`, Jekyll app can forward all calls to the victim’s phone to another phone specified by `number`.

4.3 Exploiting Kernel Vulnerabilities

Since they run directly on iOS, native apps are able to directly interact with the iOS kernel and its extensions, making the exploitation of kernel vulnerabilities possible. Even though the sandbox policy limits third-party apps to only communicate with a restricted set of device drivers, and thus significantly reduces the attack surface for kernel exploitation, security researchers still managed to find vulnerabilities in this small set of device drivers [49].

In our Jekyll app, we hide the gadgets that can enable us to communicate with the accessible device drivers. Specifically, Jekyll app can dynamically load a framework called `IOKit`, in which Jekyll app further locates the required APIs such as `IOServiceMatching`, `IOServiceOpen` and `IOConnectCallMethod` to create and manipulate connections to device drivers. Therefore, our Jekyll app provides a way for attackers to exploit kernel vulnerabilities. We demonstrate this by exploiting a kernel NULL pointer dereference vulnerability in iOS 5.x, disclosed in [49]. The exploitation of this vulnerability causes the iOS devices to reboot.

4.4 Trampoline Attack

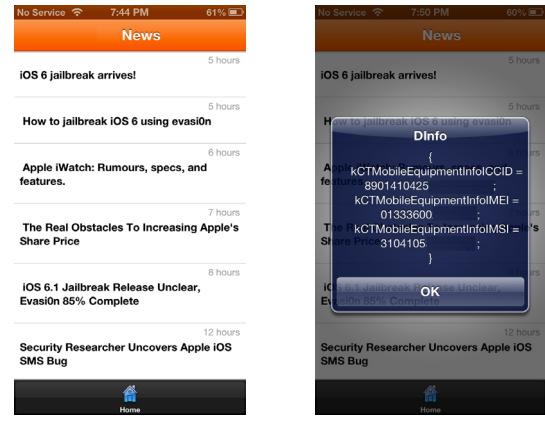
Due to the sandboxing mechanism, iOS apps are restricted from accessing files stored by other apps. However, iOS provides a form of inter-process communication (IPC) among apps using URL scheme handlers. If an app registers to handle a URL type, other apps can launch and pass messages to this app by opening a URL scheme of that type. The `http`, `mailto`, `tel`, and `sms` URL schemes are supported by built-in apps in iOS. For example, an app opening a `http` URL will cause the built-in web browser Mobile Safari to launch and load the webpage. Since attackers can fully control the content in a URL request, our Jekyll app has the ability to attack other apps that have vulnerabilities when handling malformed URL requests.

In our proof-of-concept Jekyll app, we demonstrated an attack against Mobile Safari; in particular, we prepared a web page containing malicious JavaScript code that can trigger an unpatched vulnerability in Mobile Safari. Through our Jekyll app, we can force the victim's Mobile Safari to access this web page. Finally, Mobile Safari will crash when loading the webpage due to a memory error. JailbreakMe [1], a well-known jailbreak tool, completes the untethered jailbreak through exploiting a vulnerability in Mobile Safari and then exploiting a kernel vulnerability. If new vulnerabilities in Mobile Safari are disclosed by other researchers in the future, we can simply take advantage of these new vulnerabilities to launch similar powerful attacks.

5 Jekyll App Implementation

We have implemented a proof-of-concept Jekyll app based on an open source news client called News:yc [2]. The original News:yc app fetches news from a server, and allows the user to share selected news items through email. We modified News:yc in several places. First, we configured it to connect to a server controlled by us. Second, we planted vulnerabilities and code gadgets in the app. These vulnerabilities are triggerable by special

news contents, and the code gadgets support all the malicious operations listed in Table 1. Third, we modified the app to use a secure protocol that provides authenticated and encrypted communication, so that the app client only accepts data from our server. In addition, the server was configured to deliver exploits only to the clients from specific IP addresses, which ensures that only our testing devices can receive the exploits. Figure 10.a shows the snapshot of the app.



a. The main UI of the app

b. After an attack, device identity is popped up for illustration purposes

Figure 10: Snapshots of the app

We submitted the app to Apple and got Apple's approval after 7 days. Figure 11 shows the approval notification from Apple. Once the app was on App Store, we immediately downloaded it into our testing devices and removed it from App Store. We have data to show that only our testing devices installed the app. The server has also been stopped after we finished the testing.

The testing results are summarized in Table 1. By exploiting the vulnerabilities and chaining the planted gadgets, we can send email and SMS and trigger a kernel vulnerability on iOS 5.x, and post tweets, record videos, steal the device identity, manipulate bluetooth, dial arbitrary number, and attack Mobile Safari on both iOS 5.x and iOS 6.x. We show the attack of stealing device identity in Figure 10.b. We have made a full disclosure of our attack to Apple.

6 Related Work

Jailbreak, which obtains the root privilege and permanently disables the code signing mechanism, represents the majority of efforts to attack iOS [38]. Since jailbreak usually relies on a combination of vulnerabilities found in the iOS kernel, the boot loaders, and even the firmware, Apple and hackers have long played a cat-and-mouse game. However, due to Apple's increasing efforts

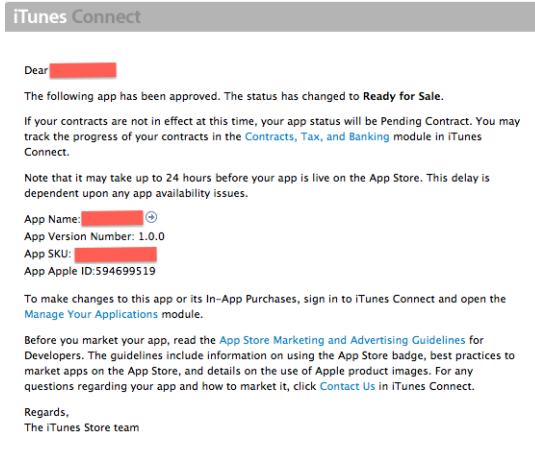


Figure 11: The approval notification from Apple

to secure iOS and keep fixing known bugs, it is becoming extremely difficult to find exploitable vulnerabilities in newer versions of iOS. Our attack does not try to achieve a jailbreak on iOS devices, instead, it takes advantage of the intrinsic incapability of the App Review process and the design flaws of iOS to deliver various types of malicious operations remotely, which cannot be trivially addressed via software updates. Note that, it is possible for Jekyll apps to take advantage of the vulnerabilities used by jailbreak tools to compromise iOS devices.

C. Miller [37] recently discovered a vulnerability in the iOS code signing mechanism, which allows attackers to allocate a writeable and executable memory buffer. He demonstrated that, by exploiting this vulnerability, a malicious app can safely pass the app review process if it generates malicious code only at runtime. However, Apple had instantly fixed the issue, and therefore, effectively blocked apps that use similar methods to load or construct malicious code during runtime.

In contrast, Jekyll apps do not hinge on specific implementation flaws in iOS. They present an incomplete view of their logic (i.e., control flows) to app reviewers, and obtain the signatures on the code gadgets that remote attackers can freely assemble at runtime by exploiting the planted vulnerabilities to carry out new (malicious) logic. In addition, the lack of runtime security monitoring on iOS makes it very hard to detect and prevent Jekyll apps. Considering that ROP attacks can achieve Turing-completeness [9] and automatic ROP shellcode generation is also possible [29, 43], the attack scheme in this paper significantly generalizes the threat in [37].

Return-Oriented Programming (ROP) [44], without introducing new instructions, carries out new logic that is not embodied in the original code. ROP and its variants [11, 29, 33, 36] allow attackers to create new control flows of a program at runtime via code gadget rear-

rangements, obviating the need for code injections that are prevented by DEP and code signing. Jekyll apps also employ code gadget rearrangements to alter runtime control flows—an idea inspired by ROP. However, our attack differs from ROP in both the assumption and the goal. Traditional ROP attack targets at programs that are out of the attacker’s control and its power is often limited by the availability of useful code gadgets.

In comparison, Jekyll apps are created and later exploited by the same person, who has the ultimate control of the gadget availability. On the other hand, traditional ROP attackers have no concern about hiding potential code gadgets and their inter-dependencies, whereas we do so that Jekyll app can bypass existing and possible detections. Currently, we need to manually construct the ROP exploits that are responsible for chaining gadgets together. However, previous studies [29, 43] have demonstrated the possibility of automatically generating ROP shellcode on the x86 platform. We leave the automatic ROP shellcode generation for Jekyll apps as future work. In addition, M. Prati [40] proposed a way to hide ROP gadgets in open source projects with a purpose to evade the code audit of the projects. This implies that even Apple could audit the source code of third-party apps in the future, detecting the hidden gadgets is still quite challenging.

Jekyll apps also share a common characteristic with trojan and backdoor programs [13], that is, the malice or vulnerabilities of attacker’s choice can be freely planted into the program, which later cooperates with the attacker when installed on a victim’s device. In fact, Jekyll app can be deemed as an advanced backdoor app that stays unsuspicious and policy-abiding when analyzed during the app review process, but turns into malicious at runtime only when new control flows are created per attacker’s command.

Thus far Apple’s strict app publishing policies and review process [5] have helped keep malicious apps out of iOS devices [41]. Automated static analysis methods, such as [17, 26], were also proposed to assist the review process in vetting iOS apps. However, as we have demonstrated with our design and evaluation of Jekyll apps, malicious apps can easily bypass human reviewers and automatic tools if their malicious logic is constructed only at runtime. This demonstrates the limitations of Apple’s current strategy that solely relies on app reviewing to find malicious apps and disallows any form of security monitoring mechanism on iOS devices.

7 Discussion

In this section, we discuss a number of possible countermeasures against Jekyll apps and analyze the effectiveness as well as the feasibility of these countermeasures.

7.1 Possible Detection at App Review Stage

Two possible directions that the app reviewers may pursue to detect Jekyll apps are: (i) discover the vulnerabilities we plant; (ii) identify the code gadgets we hide.

We emphasize that discovering software vulnerabilities using static analysis alone is fundamentally an undecidable problem [35], even without considering the powerful adversary in our attack who can arbitrarily obscure the presence of the vulnerabilities. Dynamic analysis based vulnerability detection approaches can also be easily defeated by using complicated trigger conditions and encrypted input data. We argue that the task of making all apps in App Store vulnerability-free is not only theoretically and practically difficult, but also quite infeasible to Apple from an economic perspective because such attempts will significantly complicate the review tasks, and therefore, prolong the app review and approval process that is already deemed low in throughput by third-party app developers.

To simplify the engineering efforts, our current implementation of Jekyll app directly includes some code gadgets in an isolated fashion (i.e., unreachable from program entry points), essentially leaving them as dead code that may be detectable and in turn removed during app review process. However, given our freedom to craft the app, it is totally possible to collect all gadgets from the code that implements the legitimate functionalities of the app, without the need to hide extra gadgets as dead code.

In summary, even though the hidden vulnerabilities and gadgets might take unusual forms comparing with regular code, accurately detecting Jekyll apps (e.g., based on statistical analysis) is still an open challenge. Thus, detecting Jekyll apps in App Review process via vulnerability discovery or gadgets identification is not a feasible solution.

7.2 Possible Mitigation through Improved or New Runtime Security

Generally, improving the existing security mechanisms or introducing more advanced runtime monitoring mechanisms can limit Jekyll apps' capability to perform malicious operations. However, completely defeating Jekyll apps is not easy.

- A natural idea to limit Jekyll apps is to technically prevent third-party apps from loading private frameworks or directly invoking private APIs. However, Jekyll apps do not have to dynamically load private frameworks. As we discussed, since many public frameworks rely on these private frameworks, Jekyll apps can reasonably link to these public frameworks so that certain private frameworks will

also be loaded into the process space by the system linker. A more strict execution environment like Native Client [50] can help prevent the apps from directly invoking private APIs by loading private frameworks into a separate space and hooking all invocations. However, since iOS public and private frameworks are tightly coupled, applying such a mechanism to iOS is quite challenging.

- Fine-grained ASLR such as [27, 39, 46] can greatly reduce the number of gadgets that we can locate during runtime even with the help of the planted information leakage vulnerabilities. Although expanding the scale and refining the granularity of the information leakage can help obtain a detailed view of the memory layout, Jekyll apps may lose the stealthiness due to the increased exposure of the vulnerabilities and increased runtime overhead.
- A fine-grained permission model, sandbox profile, or user-driven access control policy [28, 42] can also help limit the damages done by Jekyll apps. However, simply using Android-like permission system will not be an unsurmountable obstacle to Jekyll apps. As long as a Jekyll app can reasonably require all permissions, it can still carry out certain attacks successfully. A user-driven access control model [28, 42] also cannot stop Jekyll apps from abusing the access already granted and attacking other apps or the kernel. Take the greeting card app in Section 3.1 as an example. After the user allows the greeting card app to access the address book, it is very hard to prevent the app from leaking the information.
- Since Jekyll apps heavily rely on control flow hijacking vulnerabilities, advanced exploit prevention techniques such as CFI [6] may effectively limit Jekyll apps. CFI ensures that runtime control-flow transfers conform with the rules that are derived from the static analysis of the program and the constraints inferred from the execution context. MoCFI [14] and PSiOS [47] brought the same idea to iOS with a caveat that they require jailbroken devices. Despite its high performance overhead and low adoption rate in practice, CFI is generally deemed effective against conventional ROP attacks, which partially inspired the design of Jekyll apps. In principle, if properly implemented and deployed on iOS, CFI can significantly increase the complexity of designing Jekyll apps and force attackers to trade code flexibility for success. Although skilled attackers presumably can either employ very systematic non-control data attacks [12] to perform malicious operations or use function-level gadgets to bypass

CFI, given their freedom to craft the gadgets in our attack, they may have to sacrifice the stealthiness of Jekyll apps to some extent due to the increased distinguishability caused by such techniques.

- Type-safe programming languages like Java are immune to low-level memory errors such as buffer overflows. Thus, if we can enforce that third-party apps be developed in type-safe programming languages, we can prevent the problems of planted control flow hijacking or information leakage vulnerabilities in the apps.

In summary, we advocate the official support for runtime security monitoring mechanisms on iOS. Our design of Jekyll apps intends to motivate such mechanisms, which can protect iOS against advanced attacks and ensure that the app review practice and regulations receive their maximum efficacy.

8 Conclusion

In this paper, we presented a novel attack scheme that can be used by malicious iOS developers to evade the mandatory app review process. The key idea is to dynamically introduce new execution paths that do not exist in the app code as reviewed by Apple. Specifically, attackers can carefully plant a few artificial vulnerabilities in a benign app, and then embed the malicious logic by decomposing it into disconnected code gadgets and hiding the gadgets throughout the app code space. Such a seemingly benign app can pass the app review because it neither violates any rules imposed by Apple nor contains functional malice. However, when a victim downloads and runs the app, attackers can remotely exploit the planted vulnerabilities and in turn assemble the gadgets to accomplish various malicious tasks.

We demonstrated the versatility of our attack via a broad range of malicious operations. We also discussed our newly discovered private APIs in iOS that can be abused to send email and SMS and post tweets without the user’s consent.

Our proof-of-concept malicious app was successfully published on App Store and tested on a controlled group of users. Even running inside the iOS sandbox, the app can stealthily post tweets, take photos, gather device identity information, send email and SMS, attack other apps, and even exploit kernel vulnerabilities.

Acknowledgements

We thank our shepherd Benjamin Livshits and the anonymous reviewers for their valuable comments. This material is based upon work supported in part by the National

Science Foundation under grants no. CNS-1017265 and no. CNS-0831300, and the Office of Naval Research under grant no. N000140911042. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation or the Office of Naval Research.

References

- [1] JailbreakMe. <http://www.jailbreakme.com/>.
- [2] Newsyc, the open source news client for iOS. <https://github.com/Xuzz/newsyc>.
- [3] Unstructured supplementary service data. http://en.wikipedia.org/wiki/Unstructured_Supplementary_Service_Data.
- [4] Apple’s worldwide developers conference keynote address, June 2010. <http://www.apple.com/apple-events/wwdc-2010/>.
- [5] Apple’s app store review guidelines, 2013. <https://developer.apple.com/appstore/resources/approval/guidelines.html>.
- [6] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-flow integrity principles, implementations, and applications. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, Alexandria, VA, USA, 2005.
- [7] A. Bednarz. Cut the drama: Private apis, the app store & you. 2009. <http://goo.gl/4eVr4>.
- [8] D. Blazakis. The apple sandbox. In *Blackhat DC*, Jan 2011.
- [9] 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 (CCS)*, Alexandria, VA, USA, 2008.
- [10] Bulba and Kil3r. Bypassing stackguard and stackshield. *Phrack Magazine*, 56(5), 2000.
- [11] 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 (CCS)*, Chicago, IL, USA, Oct 4-8, 2010.
- [12] S. Chen, J. Xu, E. C. Sezer, P. Gaurar, and R. K. Iyer. Non-control-data attacks are realistic threats. In *Proceedings of the 14th conference on USENIX Security Symposium*, pages 12–12, 2005.
- [13] S. Dai, T. Wei, C. Zhang, T. Wang, Y. Ding, Z. Liang, and W. Zou. A framework to eliminate backdoors from response-computable authentication. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, 2012.
- [14] L. Davi, R. Dmitrienko, M. Egele, T. Fischer, T. Holz, R. Hund, S. Nrnberger, and A. reza Sadeghi. Mocfi: A framework to mitigate control-flow attacks on smartphones. In *In Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2012.
- [15] L. Davi, A. Dmitrienko, A.-R. Sadeghi, and M. Winandy. Return-oriented programming without returns on arm. Technical Report HGI-TR-2010-002, System Security Lab, Ruhr University Bochum, Germany, 2010.
- [16] S. designer. *Bugtraq*, Aug, 1997. return-to-libc attack.

- [17] M. Egele, C. Kruegel, E. Kirda, and G. Vigna. Pios: Detecting privacy leaks in ios applications. In *18th Annual Network and Distributed System Security Symposium (NDSS)*, February 2011.
- [18] 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 9th USENIX conference on Operating systems design and implementation*, OSDI'10, 2010.
- [19] J. Engler, S. Law, J. Dubik, and D. Vo. Ios application security assessment and automation: Introducing sira. In *Black Hat USA*, LAS VEGAS, 2012.
- [20] K. Ermakov. Your flashlight can send sms. <http://blog.ptsecurity.com/2012/10/your-flashlight-can-send-sms-one-more.html>, Oct 2012.
- [21] S. Esser. Antidote 2.0 -ASLR in iOS. In *Hack In The Box(HITB)*, Amsterdam, May 2011.
- [22] S. Esser. Ios kernel exploitation. In *Black Hat USA*, LAS VEGAS, 2011.
- [23] D. ETHERINGTON. Iphone app contains secret game boy advance emulator, get it before it's gone. March 2013. <http://goo.gl/OGyc0>.
- [24] A. P. Felt, M. Finifter, E. Chin, S. Hanna, and D. Wagner. A survey of mobile malware in the wild. In *Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices (SPSM)*, pages 3–14, 2011.
- [25] J. Han, S. M. Kywe, Q. Yan, F. Bao, R. H. Deng, D. Gao, Y. Li, and J. Zhou. Launching generic attacks on ios with approved third-party applications. In *11th International Conference on Applied Cryptography and Network Security (ACNS 2013)*. Banff, Alberta, Canada, June 2013.
- [26] J. Han, Q. Yan, D. Gao, J. Zhou, and R. H. Deng. Comparing Mobile Privacy Protection through Cross-Platform Applications. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 2013.
- [27] J. D. Hiser, A. Nguyen-Tuong, M. Co, M. Hall, and J. W. Davidson. Ilr: Where'd my gadgets go? In *Proceedings of the 33rd IEEE Symposium on Security and Privacy*, pages 571–585, San Francisco, CA, USA, May 2012.
- [28] J. Howell and S. Schechter. What you see is what they get: Protecting users from unwanted use of microphones, cameras, and other sensors. In *The Web 2.0 Security & Privacy Workshop (W2SP)*, 2010.
- [29] R. Hund, T. Holz, and F. C. Freiling. Return-oriented rootkits: Bypassing kernel code integrity protection mechanisms. In *Proceedings of the 18th USENIX Security Symposium*, Montreal, Canada, Aug, 2009.
- [30] iOS Market Statistics, 2012. <http://goo.gl/LSK7I/>.
- [31] iOS Security, May 2012. http://images.apple.com/ipad/business/docs/iOS_Security_May12.pdf.
- [32] H. Kipp. Arm gcc inline assembler cookbook. 2007. <http://www.ethernut.de/en/documents/arm-inline-asm.html>.
- [33] T. Kornau. Return oriented programming for the arm architecture. Master's thesis, Ruhr-University Bochum, Germany, 2009.
- [34] C. Kruegel, E. Kirda, and A. Moser. Limits of Static Analysis for Malware Detection. In *Proceedings of the 23rd Annual Computer Security Applications Conference (ACSAC)*, Miami Beach, Florida, USA, Dec, 2007.
- [35] D. Larochelle and D. Evans. Statically detecting likely buffer overflow vulnerabilities. In *Proceedings of the 10th conference on USENIX Security Symposium*, Berkeley, CA, USA, 2001.
- [36] K. Lu, D. Zou, W. Wen, and D. Gao. Packed, printable, and polymorphic return-oriented programming. In *Proceedings of the 14th International Symposium on Recent Advances in Intrusion Detection (RAID)*, Menlo Park, California, USA, September 2011.
- [37] C. Miller. Inside ios code signing. In *Symposium on Security for Asia Network (Syscan)*, Taipei, Nov 2011.
- [38] C. Miller, D. Blazakis, D. DaiZovi, S. Esser, V. Iozzo, and R.-P. Weinmann. *iOS Hacker's Handbook*. Wiley, 1 edition edition, May 2012.
- [39] V. Pappas, M. Polychronakis, and A. D. Keromytis. Smashing the gadgets: Hindering return-oriented programming using in-place code randomization. In *Proceedings of the 33rd IEEE Symposium on Security and Privacy*, pages 601–615, San Francisco, CA, USA, May 2012.
- [40] M. PRATI. *ROP gadgets hiding techniques in Open Source Projects*. PhD thesis, University of Bologna, 2012.
- [41] P. Roberts. Accountability, not code quality, makes ios safer than android. April 2012. <http://goo.gl/ZaXhj>.
- [42] F. Roesner, T. Kohno, A. Moshchuk, B. Parno, H. J. Wang, and C. Cowan. User-driven access control: Rethinking permission granting in modern operating systems. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, Washington, DC, USA, 2012.
- [43] E. J. Schwartz, T. Avgerinos, and D. Brumley. Q: Exploit hardening made easy. In *Proceedings of USENIX Security*, San Francisco, CA, USA, 2011.
- [44] 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)*, Alexandria, VA, USA, Oct. 29-Nov. 2, 2007.
- [45] H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh. On the effectiveness of address-space randomization. In *Proceedings of the 11th ACM conference on Computer and communications security*, pages 298–307, Washington DC, USA, 2004.
- [46] R. Wartell, V. Mohan, K. W. Hamlen, and Z. Lin. Binary stirring: self-randomizing instruction addresses of legacy x86 binary code. In *Proceedings of the 2012 ACM conference on Computer and communications security (CCS)*, Raleigh, NC, USA, Oct, 2012.
- [47] T. Werthmann, R. Hund, L. Davi, A.-R. Sadeghi, and T. Holz. Psios: Bring your own privacy & security to ios devices. In *8th ACM Symposium on Information, Computer and Communications Security (ASIACCS 2013)*, May 2013.
- [48] Wikipedia. iOS jailbreaking. 2013. http://en.wikipedia.org/wiki/IOS_jailbreaking.
- [49] H. Xu and X. Chen. Find your own ios kernel bug. In *Power of Community (POC)*, Seoul, Korea, 2012.
- [50] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaki, N. Narula, and N. Fullagar. Native client: A sandbox for portable, untrusted x86 native code. In *Proceedings of the 2009 30th IEEE Symposium on Security and Privacy*, 2009.
- [51] D. A. D. Zovi. Ios 4 security evaluation. In *Blackhat USA*, Las Vegas, NV, Aug 2011.

Measuring the practical impact of DNSSEC Deployment

Wilson Lian
UC San Diego

Eric Rescorla
RTFM, Inc.

Hovav Shacham
UC San Diego

Stefan Savage
UC San Diego

Abstract

DNSSEC extends DNS with a public-key infrastructure, providing compatible clients with cryptographic assurance for DNS records they obtain, even in the presence of an active network attacker. As with many Internet protocol deployments, administrators deciding whether to deploy DNSSEC for their DNS zones must perform cost/benefit analysis. For some fraction of clients—those that perform DNSSEC validation—the zone will be protected from malicious hijacking. But another fraction of clients—those whose DNS resolvers are buggy and incompatible with DNSSEC—will no longer be able to connect to the zone. Deploying DNSSEC requires making a cost-benefit decision, balancing security for some users with denial of service for others.

We have performed a large-scale measurement of the effects of DNSSEC on client name resolution using an ad network to collect results from over 500,000 geographically-distributed clients. Our findings corroborate those of previous researchers in showing that a relatively small fraction of users are protected by DNSSEC-validating resolvers. And we show, for the first time, that enabling DNSSEC measurably increases end-to-end resolution failures. For every 10 clients that are protected from DNS tampering when a domain deploys DNSSEC, approximately one ordinary client (primarily in Asia) becomes unable to access the domain.

1 Introduction

The *Domain Name System* (DNS) [32], used to map names to IP addresses, is notoriously insecure; any active attacker can inject fake responses to DNS queries, thus corrupting the name → address mapping. In order to prevent attacks on DNS integrity, the *Internet Engineering Task Force* (IETF) has developed DNSSEC [4], a set of DNS extensions which allows DNS records to be digitally signed, thus preventing—or at least detecting—tampering.

Over the past several years, public enthusiasm for DNSSEC has increased significantly. In July 2010, the DNSSEC root zone (containing all top level domains) was signed; in March 2011, .com, the largest top level domain, was signed; in January 2012, Comcast announced that they had switched all of their DNS resolvers to do DNSSEC validation and that

they had DNSSEC-signed all customer domains they were serving [30]. Moreover, protocol designs which *depend* on DNSSEC have started to emerge. For instance, DANE [20] is a DNS extension that uses DNS to authenticate the name → public key binding for SSL/TLS connections. Obviously, DANE is not secure in the absence of DNSSEC, since an attacker who can man-in-the-middle the SSL/TLS connection can also forge DNS responses.

Despite the effort being poured into DNSSEC, actual deployment of signed records at the end-system level has remained quite limited. As of February 2013, VeriSign Labs’ Scoreboard¹ measured 158,676 (.15%) of .com domains as secured with DNSSEC. As with many Internet protocol deployments, there is a classic collective action problem: because the vast majority of browser clients do not verify DNSSEC records or use resolvers which do, the value to a server administrator of deploying a DNSSEC-signed zone is limited. Similarly, because zones are unsigned, client applications and resolvers have very little incentive to perform DNSSEC validation.

A zone administrator deciding whether to deploy DNSSEC must weigh the costs and benefits of:

- The fraction of clients whose resolvers validate DNSSEC records and therefore would be able to detect tampering if it were occurring and DNSSEC were deployed.
- The fraction of clients which fail with valid DNSSEC records and therefore will be unable to reach the server whether or not tampering is occurring.

In this paper, we measure these values by means of a large-scale study using Web browser clients recruited via an advertising network. This technique allows us to sample a cross-section of browsers behind a variety of network configurations without having to deploy our own sensors. Overall, we surveyed 529,294 unique clients over a period of one week. Because of the scale of our study and the relatively small error rates we were attempting to quantify, we encountered several pitfalls that can arise in ad-recruited

¹Online: <http://scoreboard.verisignlabs.com/>. Visited 20 February 2013.

browser measurement studies. Our experience may be relevant to others who wish to use browsers for measurements, and we describe some of these results in Section 4.2.

Ethics. Our experiment runs automatically without user interaction and is intended to measure the behavior and properties of hosts along the paths from users to our servers rather than the users themselves. We worked with the director of UC San Diego’s Human Research Protections Program, who certified our study as exempt from IRB review.

2 Overview of DNS and DNSSEC

A DNS name is a dot-separated concatenation of labels; for example, the name `cs.ucsd.edu` is comprised of the labels `cs`, `ucsd`, and `edu`. The DNS namespace is organized as a tree whose nodes are the labels and whose root node is the empty string label. The name corresponding to a given node in the tree is the concatenation of the labels on the path from the node to the root, separated by periods.

Associated with each node are zero or more *resource records* (RRs) specifying information of different types about that node. For example, IP addresses can be stored with type A or AAAA RRs, and the name of the node’s authoritative name servers can be stored in type NS RRs. The set of all RRs of a certain type² for a given name is referred to as a *resource record set* (RRset).

2.1 Delegation

DNS is a distributed system, eliminating the need for a central entity to maintain an authoritative database of all names. The DNS namespace tree is broken up into **zones**, each of which is owned by a particular entity. Authority over a subtree in the domain namespace can be delegated by the owner of that subtree’s parent. These delegations form zone boundaries. For example, a name registrar might delegate ownership of `example.com` to a customer, forming a zone boundary between `.com` and `example.com` while making that customer the authoritative source for RRsets associated with `example.com` and its subdomains. The customer can further delegate subdomains of `example.com` to another entity. Figure 1 depicts an example DNS tree.

2.2 Address resolution

The most important DNS functionality is the resolution of domain names to IP addresses (retrieving

²And class, but for our purposes class is always IN, for “Internet.”

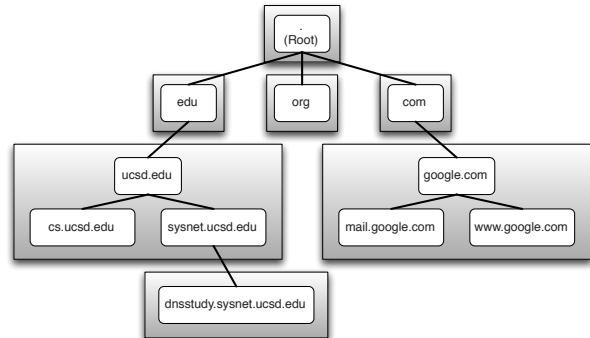


Figure 1: Example DNS name tree. Shaded boxes represent zone boundaries. Edges that cross zone boundaries are delegations.

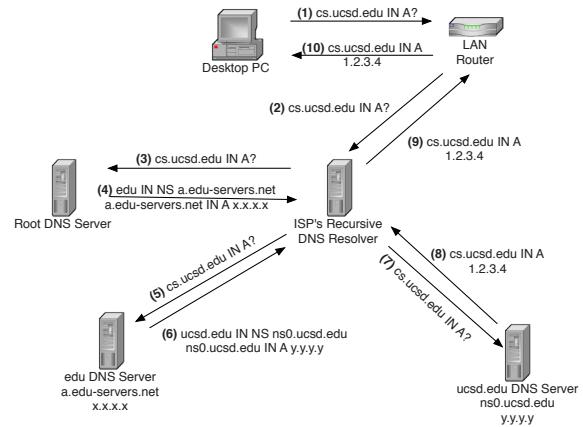


Figure 2: Simplified DNS address resolution procedure for `cs.example.tld`. In this example, there are at most one nameserver and one IP address per name.

type A or AAAA RRsets). Domain name resolution is performed in a distributed, recursive fashion starting from the root zone, as shown in Figure 2. Typically, end hosts do not perform resolution themselves but instead create DNS queries and send them to *recursive resolvers*, which carry out the resolution to completion on their behalves. When a nonrecursive DNS server receives a query that it cannot answer, it returns the name and IP address of an authoritative name server as far down as possible along the path to the target domain name. The recursive resolver then proceeds to ask that server. In this fashion, the query eventually reaches a server that can answer the query, and the resolution is complete. This recursive process is bootstrapped by hardcoding the names and IP addresses of root nameservers into end hosts and recursive resolvers.

2.3 DNS (in)security

The original DNS design did not provide any mechanisms to protect the integrity of DNS response messages. Thus, an active network attacker can launch a woman-in-the-middle attack to inject her own responses which would be accepted as if they were legitimate. This attack is known as *DNS spoofing*. Moreover, because recursive resolvers typically cache responses, a single spoofed response can be used to perform a *DNS cache poisoning* attack, which results in future responses to requests for the same RRset returning the bogus spoofed response. The mechanisms by which DNS cache poisoning is carried out are outside the scope of this work but have been studied more formally in [38]. DNS spoofing and cache poisoning may be used to compromise any type of DNS RR.

2.4 DNSSEC to the rescue

The *Domain Name System Security Extensions* (DNSSEC) [4], aim to protect against DNS spoofing attacks by allowing authoritative nameservers to use public key cryptography to digitally sign RRsets. Security-aware recipients of a signed RRset are able to verify that the RRset was signed by the holder of a particular private key, and a chain of trust from the root zone downwards ensures that a trusted key is used to validate signatures.

While DNSSEC adds a number of new RR types, the *DNSKEY*, *RRSIG*, *DS* only the records are relevant for our purposes; we describe them briefly here.

DNSKEY: DNSKEY records are used to hold public keys. Each zone authority generates at least one public/private key pair, using the private keys to sign RRsets and publishing the public keys in Domain Name System Key (DNSKEY) resource records.

RRSIG: When a zone is signed, a resource record signature (RRSIG) resource record is generated for each RRset-public key pair. In addition to containing a cryptographic signature and the name and type of the RRset being signed, the RRSIG RR specifies a validity window and the name of the signing key's owner.

DS: Lastly, the Delegation Signer (DS) RR type links signed zones to establish the chain of trust. Each DS RR contains the digest of one of the sub-zone's DNSKEY RRs.

DNSSEC's security is built on the chain of trust model. Starting from a "trust anchor," a validator attempts to trace a chain of endorsements from the root all the way to the RRset being validated; I.e., that each DNSKEY or DS record along the path and the final RRSet is correctly signed by the parent's

public key. If a chain of trust can be constructed all the way to the trust anchor, then the validating resolver can have confidence that the information in that RR is correct—or at least that it is cryptographically authenticated.

Because DNSSEC is a retrofit onto the existing insecure DNS, it is explicitly designed for incremental deployment, and insecure (i.e., unsigned) domains can coexist with secure domains. Thus, DNSSEC-capable resolvers should be able to resolve unsigned domains, and non-DNSSEC resolvers should be able to resolve DNSSEC-signed domains, though of course they will not gain any security value. In order to make this work, DNSSEC records are designed to be backwards-compatible with existing resolvers, and DNSSEC resolvers are able to distinguish zones which simply are not signed from those which are signed but from which an attacker has stripped the signatures (the DS record is used for this purpose).

Unfortunately, while DNSSEC is *designed* to be backwards compatible, it is known [9] that there are some network elements which do not process DNSSEC records properly. The purpose of this work is to determine the frequency of such elements and in particular their relative frequency to elements which actually validate DNSSEC signatures and thus benefit from its deployment.

3 Methodology

In order to address this question, we conducted a large-scale measurement study of web browsers in the wild. In particular, we sought to measure two quantities:

- What fraction of clients validate DNSSEC records and therefore would be able to detect tampering if it were occurring and DNSSEC were deployed?
- What fraction of clients fail with valid DNSSEC records and therefore will be unable to reach the server whether or not tampering is occurring?

Answering these questions requires taking measurements from a large number of clients. We gathered our clients by purchasing ad space from an online advertising network; the ad network enabled us to host an ad at a fixed URL which would be loaded in an iframe on various publishers' web sites. Our ad included JavaScript code to drive the experiment and was executed without any user interaction upon the loading of the ad iframe in clients' browsers. In order to minimize sampling bias, our ad campaign did not target any particular keywords or countries.

However, because our measurements were sensitive to the reliability of the participants’ Internet connections, we configured our ad campaign to target desktop operating systems, to the exclusion of mobile users.

Our client-side “driver script” (discussed in detail in § 3.1) induces participants’ browsers to load 1×1 -pixel images (“test resources”) from various domains. This is a standard technique for inducing the browser to load resources from different origins than the containing document. These domains fall into the following three classes:

- *nosec* — without DNSSEC
- *goodsec* — with correctly-configured DNSSEC
- *badsec* — with DNSSEC against which we simulate misconfiguration or tampering by an active network attacker

The *goodsec* and *badsec* zones were signed with 1024-bit keys³ using RSA-SHA1.

If we observe an HTTP request for a test resource, we conclude that the participant’s browser was able to resolve that type of domain. Otherwise, we conclude that it was not.

These three domain classes allow us to assess the client/resolver’s DNSSEC behavior. The *nosec* domain class serves as a control, representing the state of the majority of the sites on the web. Failed loads from the *goodsec* domain class allow us to measure the fraction of clients which would not be able to reach a DNSSEC-enabled site, even in the absence of an attack. Failed loads from the *badsec* domain class tell us about the fraction of clients which detect and react to DNSSEC tampering.

During each ad impression, the driver script attempts to resolve and load a total of 27 test resources. They are distributed as follows: one *nosec* domain, one *goodsec* domain, and 25 different *badsec* domains. Each *badsec* variant simulates an attack against DNSSEC at a different point in the chain of trust, and as we will see in Section 4, certain validating resolvers exhibit bugs that cause some *badsec* domains to be treated as correctly-signed.

3.1 Client-side experiment setup

Figure 3 shows how our driver script is embedded in an ad in a publisher’s web page. We provide the ad network with an ad located at a static URL which is wrapped in an iframe by the ad network. The publisher places an iframe in its web page whose source

³We attempted to use 2048-bit keys, but at the time of the experiment, our domain registrar, GoDaddy, did not support keys that large.

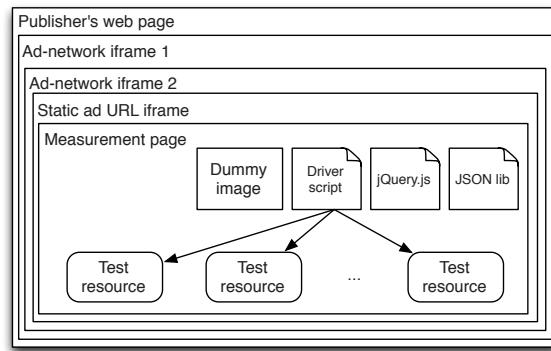


Figure 3: Client-side experiment setup

points to the iframe wrapping the ad. Our ad page residing at the static URL iframes the **measurement page**, which contains the JavaScript driver program. Each instance of the measurement page and all requests generated by it are linked by a version 4 UUID [29] placed in both the URL query string and the domain name (with the exception of the measurement page, which only has it in the query string).

The measurement page loads a dummy ad image and 3 pieces of JavaScript which are the following:

- A minified jQuery⁴ [26] library hosted by jquery.com
- A JSON encoding and decoding library hosted on our servers
- The experiment’s JavaScript “driver script”

The measurement page body’s `onLoad` handler commences the experiment by invoking the driver script. The driver script randomizes the order of a list of *nosec*, *goodsec*, and *badsec* domains then iterates over that list, creating for each domain an image tag whose source property points to an image hosted on that domain. The creation of the image tag causes the participant’s browser to attempt to resolve the domain name and load an image from it. Because we need to gather data for all domains in the list before the participant navigates away from the web page containing the ad, the driver script does not wait for each image to complete its load attempt before proceeding to the next domain. Instead, it creates all of the image tags in rapid succession. The driver script also registers `onLoad` and `onError` callbacks on each image tag created to monitor whether each load succeeds or fails. When a callback for an image fires, the outcome of the load, along with

⁴We used jQuery to minimize browser compatibility issues.

info about the browser, are sent via jQuery’s AJAX POST mechanism to a PHP logging script on our servers. Once the driver script detects that all image tags have invoked either an `onLoad` or `onError` callback, it creates a final image tag whose source domain is a unique *nosec* domain (`UUID.complete.dnsstudy.ucsd.edu`). A DNS query for such a domain serves as a “completion signal” and allows us to identify UUIDs where the user did not navigate away from the ad publisher’s page before completing the trial. We discarded the data from any page load which did not generate the completion signal.

3.2 Identifying successful DNS resolution

Our original intent was to use `onLoad` and `onError` handlers attached to each test resource’s image tag to measure the outcome of the HTTP requests for test resources. If the `onLoad` handler was called, we would record a successful HTTP request; if instead the `onError` handler was called, we would record a failed HTTP request. These results are reported back to our servers via AJAX POST. However, we found 9754 instances of the `onError` handler firing, the test resource subsequently being loaded, and then the `onLoad` handler firing. For another 1058 test resource loads, the `onLoad` handler fired, despite our receiving neither the corresponding DNS lookups nor the HTTP requests for the test resources in question. Consequently, we looked to different avenues for identifying resolution success.

Because we are not able to ascertain the result of a DNS lookup attempt via direct inspection of the DNS caches of our participants and their recursive resolvers, we must infer it from the presence of an HTTP request whose `Host` header or request line specifies a particular test resource’s domain name as an indicator of DNS resolution success. Thus, if we observed a completion signal for a particular UUID but did not observe an HTTP request associated with that UUID for a certain test resource type, we infer that the DNS resolution for that UUID-test resource pair failed. Note however that we can record a completion signal after observing just a DNS query for it: what matters is whether the driver script attempted to load the completion signal resource, not whether it succeeded in doing so.

This strategy has the potential to over-estimate the number of DNS resolution failures due to TCP connections that are attempted and are dropped or aborted before the HTTP request is received by our servers. The only source of this type of error that we are able to control is our HTTP servers’ ability to accept the offered TCP-connection load at all times

throughout the experiment. We describe our serving infrastructure in Section 3.4. We believe it is sufficiently robust against introducing this type of error.

3.3 Cache control

Because requests fulfilled by cache hits do not generate HTTP and DNS logs that we can analyze, we took measures, described in Table 1, to discourage caching. Most importantly, the use of a fresh, random UUID for each ad impression serves as a cache-buster, preventing cache hits in both DNS resolvers and browsers.

If, despite our efforts, our static ad page is cached, causing the measurement page to be requested with a cached UUID, we must detect it and give the current participant a fresh UUID. To this end, we used a memcached cache as a UUID dictionary to detect when the measurement page was loaded with a stale UUID. If this occurred, the stale measurement page was redirected to one with a fresh UUID.

3.4 Serving infrastructure

To run our study, which generates large bursts of traffic, we rented 5 `m1.large` instances running Ubuntu 10.04 on Amazon’s Elastic Compute Cloud (EC2). All 5 instances hosted identical BIND 9 (DNS), nginx (HTTP), and beanstalkd (work queue) servers. The nginx servers supported PHP 5 CGI scripts via FastCGI. Tables 2 and 3 show the adjustments made to the servers’ configuration parameters to ensure a low rate of dropped connections.

One instance ran a MySQL server, another ran a memcached server. To increase our EC2 instances’ ability to accept large quantities of short TCP connections, we configured our machines to timeout connections in the FIN-WAIT-2 state after only a fraction of the default time and to quickly recycle connections in the TIME-WAIT state. This was accomplished by setting the sysctl variables `tcp_fin_timeout` and `tcp_tw_recycle` to 3 and 1, respectively.

3.4.1 DNS & BIND 9

All 5 EC2 instances ran identical BIND 9 DNS servers providing authoritative DNS resolution for all *nosec*, *goodsec*, and *badsec* domains. We used Round Robin DNS to distribute load across all 5 DNS and web servers. In order to reduce the chance of load failures due to large reply packets, our DNS servers were configured (using BIND’s `minimal-responses` option) to refrain from sending unsolicited RRs that are not mandated by the DNS specification. Specifically, we only send the extra DNSSEC RRs in response to queries which include the DNSSEC OK option (approximately two thirds of all queries).

Type	Value	Used on
HTTP header	<code>Cache-Control: no-cache, must-revalidate</code>	static ad page, measurement page, driver script
HTTP header	<code>Expires: Sat, 26 Jul 1997 00:00:00 GMT</code>	static ad page, measurement page, driver script
HTML <meta>	<code>http-equiv="Pragma" content="no-cache"</code>	static ad page, measurement page
HTML <meta>	<code>http-equiv="Expires" content="-1"</code>	static ad page, measurement page

Table 1: Description of the HTTP and HTML anti-caching measures and their uses.

worker_processes	8
worker_rlimit_nofile	65,535
worker_connections	65,000

Table 2: Non-default nginx server config params.

PHP_FCGI_CHILDREN	50
PHP_FCGI_MAX_REQUESTS	65,000

Table 3: Non-default PHP FastCGI config params.

Our 5 BIND servers are authoritative for all domain names used in our study except for the domain of the static ad URL that iframes the measurement page. Because we were not interested in measuring resolution of those domains we hosted their authoritative servers on Amazon Route 53 DNS service.

3.5 Data gathering

Our analysis of the behavior of participants’ browsers and resolvers is based on the following 3 data sources: nginx access logs, BIND request logs, and MySQL tables containing the outcomes and browser info reported by AJAX POST messages.

Nginx was configured to use its default “common log format” [34], which includes a timestamp of each request, the URL requested, the user agent string, among other details about the request and its corresponding response. However, BIND’s log format is hardcoded and compiled into the binary. Its default logging behavior only provides basic information about queries (e.g., a timestamp, the domain name, the source IP and port). It does not provide information about replies and excludes certain important diagnostic fields. We modified and recompiled BIND to add enhanced logging of requests and replies. Log lines for requests were modified to include the request’s transaction ID and the value of the sender’s UDP payload size from the EDNS0 OPT RR (if present) [39]. We added support for reply logs that include the transport-layer protocol in use, the size of the reply, and the transaction ID.

With these additional log details, we are able to link requests to replies and determine if a lookup fell back to TCP due to truncation of the UDP reply. BIND logs are also used to identify the UUIDs for which a completion signal was sent as well as to determine which resolvers were associated with a particular UUID.

The client-side driver script AJAX POSTs the outcome of each test resource load along with additional metadata regarding the experiment and the state of the browser environment under which it is running. These data are logged by our servers.

3.6 Experiment scheduling

In our preliminary test runs of the study, we found that the successful load rates for test resources varied depending on the time of day at which the experiment was conducted. To account for this variability, we conducted an extended study lasting for a full week. Every two hours, we paid ad network enough for 10,000 impressions to be shown.

4 Results

In this section we describe the results of our measurements. We begin by providing an overview of our data. Then, in Section 4.1 we describe our measurements of the differential impact of DNSSEC on resolution success. Finally, in Section 4.2, we describe a number of confounding network artifacts that plague any attempt to use advertisement surveys to measure small signals against the background of a noisy Web environment.

Over the course of the 84 segments of our week-long experiment, we collected data from 529,294 ad impressions, receiving DNS queries from 35,010 unique DNS resolvers. Figure 4 shows the distribution of unique resolvers performing resolution for each UUID. The distribution has a long tail, although 98% of UUIDs used at most 25 resolvers. We mapped each resolver’s IP address to its ASN and found that 92.75% of the clients surveyed were observed using recursive resolvers whose IP addresses resided in the same ASN, and 99.12% used resolvers

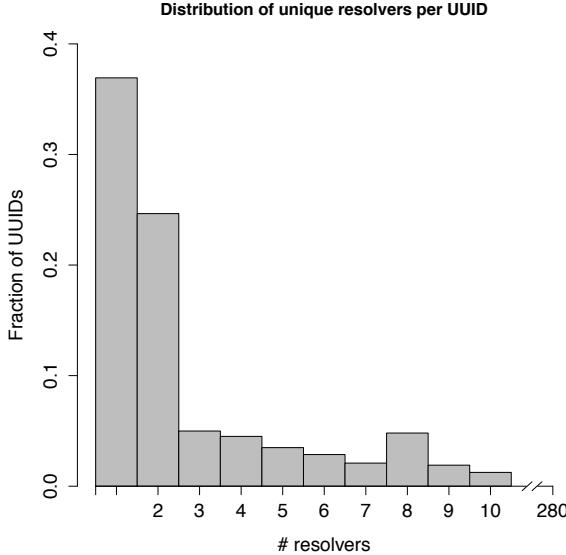


Figure 4: Distribution of the number of unique resolvers observed performing DNS resolution per UUID. Tail not shown.

in two or fewer ASNs. This is consistent with our expectation that most users use their default DNS resolvers provided by their ISPs, while a small percentage of “power users” might configure their systems to take advantage of open resolvers such as Google Public DNS.⁵

As shown in Figure 5 each ad buy results in a delay of approximately 20 minutes from the time we released funds to the ad network, at which point impressions start to appear. Incoming traffic spikes for 15 minutes, peaking around 25 minutes into the run and tapering off for the remainder of the first hour.

We also witnessed considerable drop-off at each stage of executing experiment code in the participants’ browsers. Figure 6 illustrates the number of UUIDs observed reaching each stage of the experiment. 15.88% of the ad impressions that we paid for did not even manage to load the driver script and only 63.02% of the impressions we paid for actually resulted in a completed experiment. This compares favorably with past studies. For instance, prior work by Huang et al. [22], which also used ad networks to recruit participants to run experiment code, had only a 10.97% total completion rate.

4.1 DNSSEC Resolution Rates

The first question we are interested in answering is the impact on load failure rates of introducing DNSSEC for a given domain. Table 4 shows the

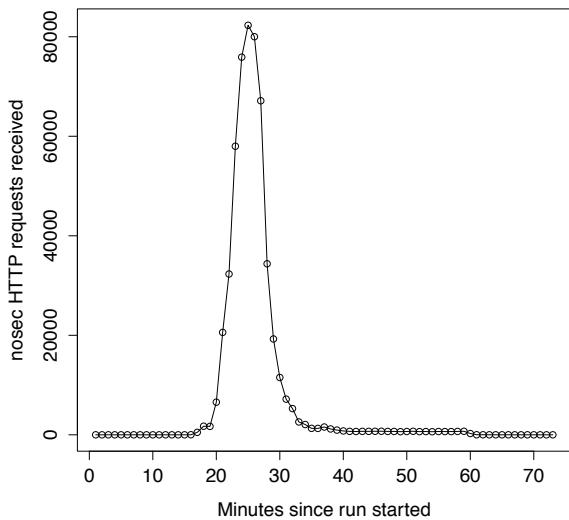


Figure 5: Plot showing total number of requests received during each minute after the start of a run, aggregated over all runs.

Class	Failure rate	CI 0.99
nosec	0.7846%	0.7539% - 0.8166%
goodsec	1.006%	0.9716% - 1.042%
badsec	2.661%	2.649% - 2.672%

Table 4: Failure rates for each class of test resource.

raw failure rates across each class of test resource, where the failure rate is defined as one minus the quotient of the number of successful test resource loads and the number of attempted resource loads across all UUIDs for which we received a completion signal. This table is sufficient to draw some initial conclusions. First, as evidenced by the low failure rate of *badsec* domains the vast majority of end hosts and their recursive resolvers do not perform DNSSEC validation. If all end hosts or recursive resolvers verified DNSSEC, we would expect a *badsec* failure rate of 100%, instead of the observed value of 2.661%. Thus, the increased security value of DNSSEC-signing a domain is relatively low, as most resolvers will not detect tampering against DNSSEC-signed domains.

Second, DNSSEC-signed domains—even validly signed domains—have a higher failure rate than non-DNSSEC-signed domains: just DNSSEC-signing a domain increases the failure rate from around

⁵<https://developers.google.com/speed/public-dns/>

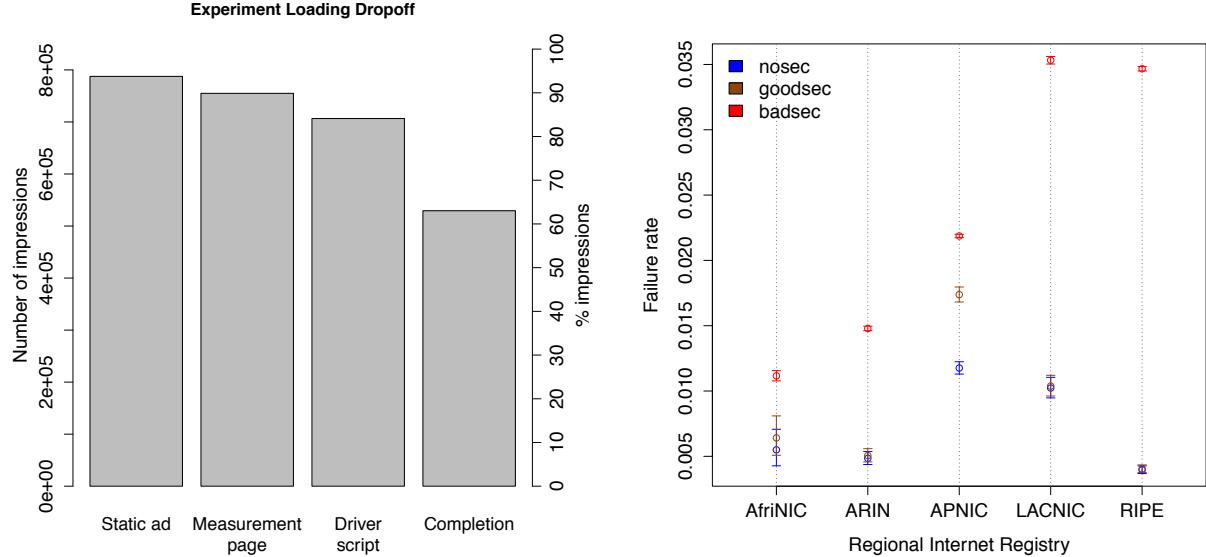


Figure 6: Plot of UUIDs that reached each stage of the experiment.

0.7846% to 1.006% (though this value is very sensitive to geographic factors, as discussed in the following section). While this is not a huge difference, it must be compared to the detection rate of bad domains, which is also very small. Moreover, because resolvers which cannot process DNSSEC at all appear to “detect” bogus DNSSEC records, the *badsec* failure rate in Table 4 is actually an overestimate of clients behind DNSSEC-validating resolvers, which is probably closer to 1.655% (the difference between the *badsec* and *goodsec* rates).

4.1.1 Geographic Effects

As mentioned above, the raw numbers are somewhat misleading because the failure rates are very geographically dependent. In order to explore this dependence we categorized each test case (UUID) by geographic area based on the resolver IP observed performing resolution for a domain containing the UUID.⁶ We used the CAIDA prefix-to-AS mapping dataset [11] to determine the Autonomous System Number (ASN) for each for client’s resolver IP address and then assigned each client to the Regional Internet Registry (RIR) which is responsible for that AS, as listed in Table 5.

⁶If there was more than one resolver associated with a particular UUID, our analytics package chose one arbitrarily during the process of merging the records. If we restrict our analysis to clients which only use one resolver, the overall error rate goes down, but our results are qualitatively similar, with the error rates being 0.0046, 0.0055, and 0.0119, for *nosec*, *goodsec*, and *badsec*, respectively.

Figure 7: Failure rates broken down by resolver IP RIR. Error bars indicate a 95 percent binomial proportion confidence interval.

As shown in Figure 7, resolution failure rates vary widely by region, as does the difference in resolution rates between *nosec*, *goodsec*, and *badsec*. In particular, while all five regions show a significant difference (2-proportion z-test, $p < 0.0001$) between aggregate *badsec*-domain outcomes and *nosec* & *goodsec* outcomes, only APNIC (Asia Pacific) shows a significant difference between *nosec* and *goodsec* (McNemar’s test, $p < 0.0001$). While AfriNIC (Africa) shows a qualitative difference, we do not have enough data points to determine whether it is statistically significant. Note that in general APNIC seems to have an elevated resolution failure rate; LACNIC (Latin America) does as well but still does not show a significant difference between *nosec* and *goodsec*. We drilled down into the resolvers responsible for anomalous failure rates and present our findings in Sections 4.1.3, & 4.1.4.

4.1.2 The Impact of Packet Size and TCP Fallback

One commonly-expressed concern with DNSSEC is that it increases the size of DNS responses and, consequently, failure rates. Ordinarily, DNS requests and responses are carried over UDP, which limits the maximum size of the responses. DNS has two mechanisms to allow responses larger than the 512-byte limit defined in RFC 1035 [33]:

- Resolution can fall back to TCP if the server supports it.

Name	Abbreviation	Frequency	Percentage
African Network Information Centre	AfriNIC	10,914	2.062%
American Registry for Internet Numbers	ARIN	75,577	14.28%
Asia-Pacific Network Information Centre	APNIC	200,366	37.86%
Latin America and Caribbean Network Information Centre	LACNIC	62,925	11.89%
Réseaux IP Européens Network Coordination Centre	RIPE NCC	179,492	33.91%
Unclassifiable		20	< 0.001%

Table 5: Table listing the 5 Regional Internet Registries (RIRs). The Frequency and Percentage columns indicate the number and relative prevalence of UUIDs for which at least one DNS query originated from each region.

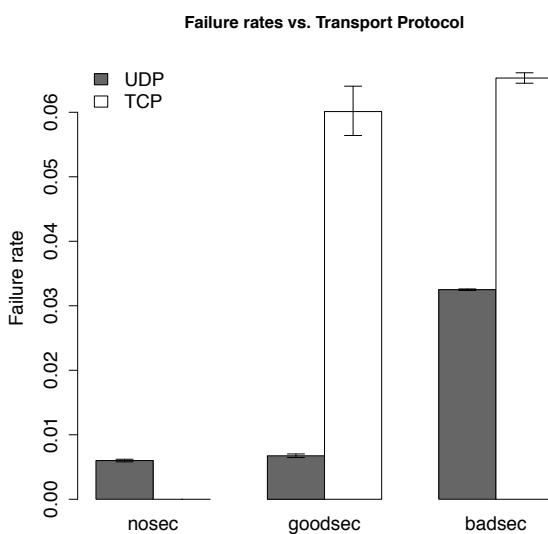


Figure 8: Failure rates broken down by DNS transport protocol. Error bars indicate a 95 percent binomial proportion confidence interval.

- Clients can advertise a larger maximum UDP datagram size via the EDNS0 OPT pseudo-RR [39].

Unfortunately, both of these mechanisms can cause problems for some intermediaries [7, 8, 10]. Because the resolver behavior is observable on the server, we can directly measure the impact of these strategies on test resource load failures.

In order to look more closely at these effects, we first filtered out the data for the 4,739,669 (33.25%) lookup requests we received which did not have the DNSSEC OK flag set. The DNSSEC OK flag announces the query source’s willingness to receive DNSSEC RRs, and thus when it is not set, our resolver simply sends the requested records with-

out the DNSSEC RRs.⁷ Non-DNSSEC OK lookups for DNSSEC resources appear to have similar success rates to *nosec* resources. Out of the remaining 9,516,394 (66.75%) transactions where DNSSEC OK was indicated, 4.22% of *goodsec* and 4.064% of *badsec* lookups fell back to TCP. These TCP lookups had dramatically higher failure rates: 6.011% for *goodsec* and 6.531% for *badsec* compared to 0.6742% for *goodsec* and 3.249% for *badsec* when UDP was used. For *nosec*, resolution never fell back to TCP, and the failure rate of 0.6%⁸. was similar to that for *goodsec* with UDP. Figure 8 summarizes these findings.

The similar UDP failure rates for *nosec* and *goodsec* suggest that it is the TCP fallback that results from DNSSEC’s increased response sizes, and not the bigger responses themselves, that is the major contributor to the elevated *goodsec* failure rate.

TCP fallback in the DNS resolution for one component of a web page can have a negative impact on the load rate of other components on the page, even if their DNS lookups do not themselves fall back to TCP. If we partition the UUIDs into those that fall back to TCP for at least one test resource and those that never fall back to TCP, we find that the *nosec* failure rates are 1.0791% for the former and 0.7617% for the latter. We have not explored these effects in detail, but it seems likely that the failed resolution slows down the retrieval of the rest of the resources, thus causing failures.

We also found that accurate path MTU prediction is crucial for maintaining high resolution success. For 13,623 test resources (0.0953% of the 14,291,174 total), we observed that recursive resolvers overestimated the UDP path MTU, advertised an inflated

⁷If multiple queries were present we considered the DNSSEC OK flag to be set if any of the queries had it. 2.771% of test resources exhibited variation in this flag.

⁸This *nosec* failure rate is lower than the one found in Table 4 because, to be consistent with the *goodsec* and *badsec* failure rate calculations in this section, it excludes failed test resource loads for which we did not observe a DNS lookup attempt.

value via EDNS0, and subsequently had to retry the lookup with a smaller advertised value. Test resources whose lookups included this path MTU discovery behavior failed to load 14.09% of the time compared to 2.519% for those that did not.

4.1.3 Case Study: *badsec-b8* validation anomaly

We compared the failure rates of the *badsec* domains and observed that the *badsec-b8* variant exhibited a significantly lower failure rate (1.480%) than all other *badsec* types (McNemar’s test applied pairwise against each other *badsec* variant, $p < 0.01$). In *badsec-b8*, we simulated an invalid DNSKEY RRSIG RR by incrementing the labels field of the RR data and signing it with a correctly-authenticated key. The labels field in an RRSIG RR is used for matching RRSIGs to the RRsets they authenticate when wildcards are involved. For example, if a zone declares the `*.foo.com` wildcard name, then RRSIGs for the RRsets of names matching `*.foo.com` (e.g., `www.foo.com`) would have a labels field value of 2. Section 5.3.1 of RFC 4035 [5] stipulates that an RRSIG RR must have a labels field value less than or equal to the number of labels in the owner name of the RRset that it authenticates.

To identify resolvers responsible for this validation anomaly, we first partitioned the set of UUIDs by the IP address of the resolver associated with that UUID. Using the partitioned dataset, we identified 124 resolvers whose failure rate for *badsec-b8* was significantly lower than that of each of the other *badsec* variants (McNemar’s test, $p < 0.01$). Moreover, for 123 of these resolvers, the *badsec-b8* and *goodsec* failure rates did not significantly differ at the .01 level (McNemar’s test).

With the cooperation of one of the ISPs whose resolvers exhibited the validation anomaly, we were granted access to query their closed resolvers and were able to manually reproduce the errant validation. We also added -1, +2, and +100 to the RRSIG labels field values and found that the resolvers incorrectly accepted all of the increased values, but not the decreased value, suggesting that the DNS server implementation in use reversed the inequality for testing the labels field.

We were unable to devise a cache-poisoning attack that leverages this validation error under any reasonable threat model.

4.1.4 Case Study: *badsec-c12* validation anomaly

The failure rate of the *badsec-c12* variant (2.521%) differed significantly from those of all other *badsec*

domains and was the second lowest, after *badsec-b8* (McNemar’s test, $p < 0.01$). The *badsec-c12* subdomain attacks the DNSSEC chain of trust by not providing the RRSIG RR for the test resource’s type A RRset. A properly-validating server should not consider the affected A RRset validated unless it were able to retrieve and validate its RRSIG.

All 32 of the resolvers in our dataset that exhibited this validation anomaly belonged to the same /22 subnetwork controlled by one particular ISP, as did 45 of the 49 resolvers for which McNemar’s test showed significantly-different ($p < 0.01$) failure rates between *nosec* and *goodsec*. Customers using this ISP’s recursive resolvers suffer from the worst of both worlds. They are not only more vulnerable to a man-in-the-middle attack against DNSSEC, but also less likely to be able to access a domain with DNSSEC enabled than one without. We were unable to obtain access to the ISP’s closed recursive resolvers to try to manually reproduce the incorrect validation behavior.

Due to the reduced size of responses omitting RRSIG RRs for type A queries, no *badsec-c12* DNS resolutions fell back to TCP, and for the 32 resolvers exhibiting the validation anomaly, *badsec-c12*’s failure rate (1.245%) was significantly less than that of *goodsec* (13.81%) (McNemar’s test applied separately for each resolver, $p < 0.01$).

4.1.5 Case Study: Comcast

In January 2012, Comcast announced that it had finished deploying DNSSEC within its network and that its residential customers would thenceforth be protected by DNSSEC-validating DNS resolvers [30]. We identified dynamic Comcast IP end hosts in our dataset using a list of IP prefixes published by Comcast [13]. One should expect that Comcast end hosts in our dataset would fail on *goodsec* at a lower than average rate and *badsec* at a higher than average rate. Indeed, the 582 Comcast end hosts observed exhibited a 0.1718% failure rate for *goodsec* and a 92.5636% failure rate for *badsec*. For comparison, Comcast end hosts failed on *nosec* domains 0.1718% of the time. This result is consistent with the expectation that a network that is properly configured for DNSSEC will have identical behavior for *nosec* and *goodsec*. Our measurements indicate that the majority of the difference between Comcast’s *badsec* failure rate and 100% is caused by users who are not using Comcast for recursive resolution and therefore do not benefit from Comcast’s DNSSEC verification; if we exclude end-hosts that use resolvers outside of Comcast’s AS the *badsec* failure rate improves to 98.6544%.

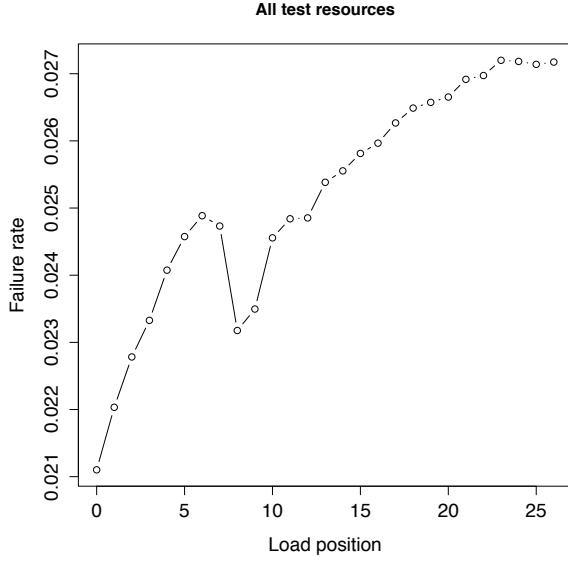


Figure 9: Plot of failure rate across all test resource types versus load order.

Percentile	Test duration (seconds)
50%	9
90%	31
95%	50
98%	100

Table 6: Percentiles from the distribution of test durations, measured from the time of the measurement page load to the time of the completion signal.

4.2 Measurement Difficulties

Because our primary measurement endpoint is the browser’s failure to retrieve a resource, we are very sensitive to any other sources of failure other than the ones we are attempting to measure; by contrast, many previous studies such as [22] measured between multiple different success outcomes, which were distinguishable from failures. In order to minimize these effects, we investigated other potential sources of failure closely, as described below.

4.2.1 Resource Load Sequence

Recall from Section 3.1, that test resource loads are initiated one after another in a random order. Because the test takes some time (see Table 6) to complete, there are a variety of conditions which can cause the test to abort prematurely. This suggests that the order in which resources are loaded may impact the error rate.

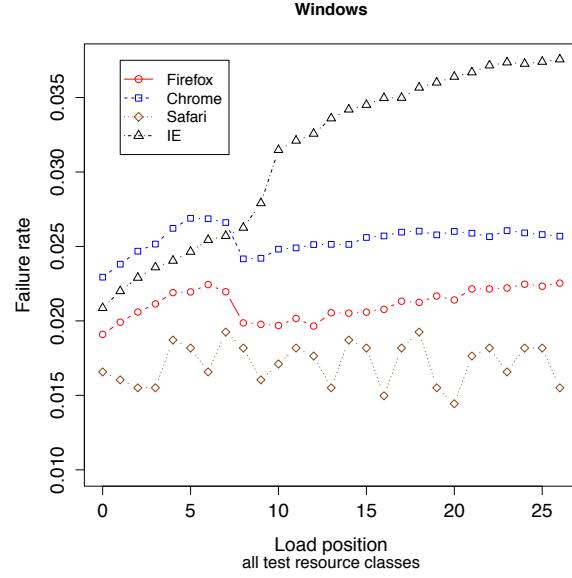


Figure 10: Failure rate versus load order across all test resources for Windows clients.

Figure 9 shows the overall failure rate versus load position (note that the first resource is at position 0). While the overall trend seems consistent with failures getting progressively worse with later resources, the sharp spike and then subsequent decline between positions 5 and 9 seems anomalous. In order to explore this further, we broke down the the failure rate by browser and operating system.

As Figures 10 and 11 make clear, Chrome and Firefox on Mac and Windows both show the same pattern of a failure spike around resources 5-8, whereas the same browsers on Linux (Figure 12) as well as both Safari and Internet Explorer show a generally linear trend (though the break around resource 9 for Internet Explorer is also puzzling). We leave the explanation of these anomalies for future work.

4.2.2 Latent UUIDs

Our analysis uncovered 3616 UUIDs for which we received completion signals without corresponding measurement page loads during the one-week experiment window. We refer to these UUIDs as **latent UUIDs** because we observed DNS and HTTP requests for FQDNs that included them in our logs prior to the start of our experiment window. There are two plausible explanations for the existence of latent UUIDs:

1. **Browser caching.** Modern web browsers cache users’ recent and open tabs to allow for restoration of the browsing session in case of

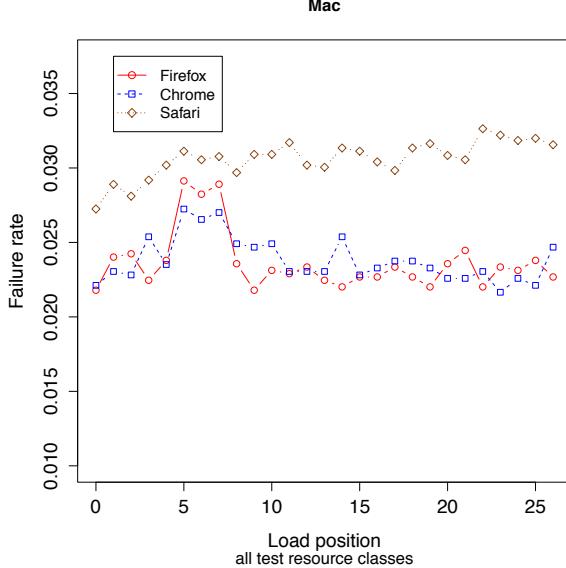


Figure 11: Failure rate versus load order across all test resources for Mac clients.

a crash, browser termination, or accidental tab closure. Half of the 18 latent UUIDs that had HTTP requests during the experiment window appeared within the first 33 hours of the experiment window, and 11 of them loaded the measurement page within the 24 hours leading up to the start of the experiment window. Thus, it is plausible that browser caching explains some of the latent UUIDs.

2. **DNS caching with eager renewal.** To improve DNS cache hit rates and, consequently, reduce client latency, Cohen and Kaplan [12] proposed a caching scheme wherein DNS caches issue unsolicited queries to authoritative name-servers for cached RRs whose TTLs have expired, even if no client queried for the RR at the time of the renewal. This mechanism is a documented feature in the Cisco IOS Distributed Director [1] and has been implemented by others [45]. Our log data strongly supports this explanation, as all latent UUIDs (by definition) appeared in the DNS logs, but only 18 had HTTP requests during the experiment window.

Latent UUIDs are not included in our analysis, as we cannot guarantee that our log data extends far enough into the past to cover them. Furthermore, our analysis only includes UUIDs for which we observed both the measurement page load and completion signal within the experiment window.

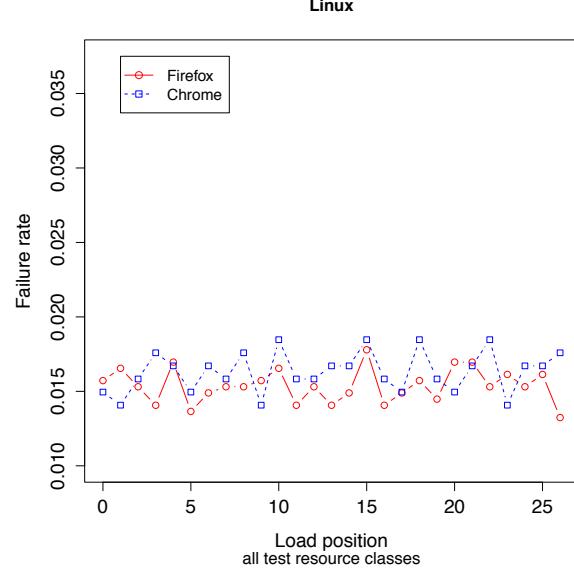


Figure 12: Failure rate versus load order across all test resources for Linux clients.

5 Discussion

The benefit from DNSSEC-signing a domain is upper-bounded by the number of clients which actually validate DNSSEC-signed records. As our measurements show, the fraction of clients which do so is less than 3%.⁹ Moreover, this benefit is only obtained if DNSSEC either deters attacks or allows detection of attacks. By contrast, for a site with worldwide users, our results indicate that deploying DNSSEC in the current environment amounts to a self-inflicted partial attack on one's own site on the order of 0.2214% (the difference between the *goodsec* and *nosec* failure rates). For a site without significant Asian usage, the tradeoff looks more attractive, and for a site with largely Asian usage it looks less attractive.

The major source of increased failure rates from DNSSEC deployment appears to be that increased packet sizes force clients into DNS over TCP rather than DNS over UDP. The failure rate for DNS over TCP is approximately 10 times larger than DNS over UDP. This phenomenon is strongly localized to Asia/Pacific browsers.

Some potential future developments could change this calculation. First, a significant number of ISPs could deploy validating resolvers. As shown by the

⁹Here we interpret the *badsec* failure rate as an upper bound on the fraction of end users protected by DNSSEC validation because some fraction of the failures may be due to the fact that DNSSEC was enabled rather than validation failure

Comcast data in Section 4.1.5, unilateral deployment of DNSSEC by ISPs can have a very large impact on the behavior of their customers. If a few of the large ISPs were to deploy validating resolvers, our measured *badsec* failure rate would no doubt have been much higher and their customers would have obtained some level of defense against attackers outside the ISP’s network. (Validation at the ISP resolver level does not provide defense against attackers located between the resolver and the customer.)

Second, there could be widespread deployment of a technology such as DANE that depends on DNSSEC. As mentioned above, DNSSEC for A records does not provide security against on-path attackers, who can intercept the traffic between the client and the server. Defending against such attackers requires some sort of cryptographic protocol such as SSL/TLS. By contrast, if DANE is used to attest to end-user certificates, then DNSSEC combined with DANE-based certificates can provide security against on-path attackers and thus a significantly greater benefit. Even with DANE, however, the collective action problem of simultaneous client and server deployment persists. In fact, it is worse since DANE’s security requires that the client do DNSSEC validation—ISP-level validation is not sufficient.

Our results also serve as a caution for future researchers: Advertisement network based studies—especially those which attempt to measure success or failure—are very sensitive to variation in client and network behavior. In particular, there is significant variation both between browsers and operating systems and by request order within the same browser/operating system pair. These variations are of the same order of magnitude as the signal we are trying to measure and thus present a significant challenge. Additionally, they may indicate actual problems with the browsers—or at least opportunities for improvement. We are currently working with browser vendors to attempt to determine the reason for these anomalies.

6 Related Work

Several research groups have performed measurements related to DNSSEC.

The SecSpider project [35, 36, 43] has surveyed DNSSEC-secured zones since the DNSSEC rollout, quantifying the rollout using metrics of availability, verifiability, and validity. Deccio et al. [14, 15] surveyed representative DNSSEC domains for misconfiguration. Both of these projects focus on properties of the authoritative DNS server zone data, rather than the behavior of resolvers or caches.

Several research groups have attempted to characterize the overhead (to clients, servers, and the network) from deploying DNSSEC. Ager, Dreger, and Feldmann [2] used a trace of DNS traffic as the basis for a testbed experiment. They noted the possibility of overhead arising from packet fragmentation. Wijngaards and Overeinder [42] described the design of a DNSSEC resolver and validator and compared its performance to an ordinary DNS resolver. Migault, Girard, and Laurent [31] measured the overhead of DNSSEC resolution in a lab setting, including the NSEC3 option.

Gudmundsson and Crocker [18] measured the deployment of DNSSEC-capable resolvers using traces of DNS queries made to the .org servers. Glynn [17] surveyed DNSSEC deployment in Ireland, highlighting the possibility that large responses would suffer fragmentation, and noting the geographic variation in client path MTU.

Dietrich [16] reports on a study of one component in DNSSEC resolution: users’ home DSL routers. With the cooperating of network operators, the study’s authors tested the behavior of 36 routers in a testbed environment. The DNS proxies in more than half of these routers were incompatible with DNSSEC; several of the tested routers could not be used with DNSSEC even if their internal DNS proxy were bypassed.

Herzberg and Shulman [19] describe several challenges to wide-scale DNSSEC deployment. They observe that large-response fragmentation not only reduces performance but can be the basis of downgrade and forgery attacks on permissive resolvers.

Pappas and Keromytis [37] performed a distributed measurement of resolution failures in the aftermath of the May 5, 2010 signing of the DNS root. They made resolution attempts from hundreds of geographically dispersed nodes (e.g., Tor exit nodes), which allowed them to observe the behavior of many DNS resolvers. Whereas their conclusions focused on the effects of rolling out DNSSEC on the DNS root-level servers, our measurements target the DNS in its steady state behavior nearly two years later.

Krishnan and Monroe [28] performed a large-scale measurement of browser DNS prefetching and characterized its security and privacy implications. Using a trace-based cache simulator, they showed that the additional overheads induced by prefetching would increase the overhead of deploying DNSSEC.

The Netalyzr platform [27] allows interested users to measure and report on properties of their Internet connection. Netalyzr has uncovered widespread DNS manipulation by ISPs [40, 41].

Zhang et al. [44] included client-side DNS measurement code in a software package used by millions. They identified several ISPs that manipulate DNS results, allowing them to proxy and modify Web searches.

Ager et al. [3] asked friends to run DNS measurement code on their systems. Their data analysis focused on DNS performance.

Honda et al. [21] asked IETF colleagues to run a measurement tool, TCPExposure; this tool generated TCP segments with various properties, allowing Honda et al. to observe how middleboxes between clients and their servers handle different TCP extensions. Of all the related work, Honda et al.'s is the closest in spirit to ours. They sought to measure the compatibility of hypothetical future protocols with deployed middleboxes; we measure the interaction between DNSSEC and today's network infrastructure.

Rather than deploy custom software to users, we wrote JavaScript code that triggers DNS resolution, and served this code in an ad we placed with a display ad network. This strategy for enlisting users was pioneered by Barth, Jackson, and their coauthors [6, 25], who used it to measure the Web platform. This strategy was also recently used by Huston and Michaelson [23, 24] to measure the deployment of DNSSEC-capable resolvers and to describe their geographic distribution. Unlike our study, Huston did not also measure the prevalence of DNSSEC-intolerant resolvers.

7 Summary

While DNS name resolution is a key part of the Internet infrastructure, it has long been known that it is seriously insecure. DNSSEC is designed to repair that insecurity. We report on a large ad network based study designed to measure both the current state of deployment *and* the extent to which deploying DNSSEC-signed domains creates collateral damage in the form of failed resolutions of valid domains.

Our measurements confirm previous reports that DNSSEC deployment is proceeding quite slowly. Less than 3% of clients failed to retrieve resources hosted on DNSSEC-signed domains with broken signatures. This indicates that either these clients—or their resolvers—are not doing DNSSEC validation or they are not hard-failing on broken validations, which is effectively the same as not validating at all. Moreover, about 1.006% of clients fail to retrieve validly DNSSEC-signed resources (as compared to 0.7846% of unsigned resources). In other words, for every ten clients a site protects by using DNSSEC, it self-DoSes about one client. This effect is principally

due to TCP fallback to accommodate larger DNSSEC packet sizes and is strongly localized to Asian users.

Finally, we report on a number of new measurement artifacts that can affect the results of advertising network based studies, including some browser-specific anomalies which may reveal opportunities for improvement in those browsers. In future work, we hope to explore further the specific causes of these anomalies.

Acknowledgements

The authors thank the anonymous reviewers and our shepherd, Tara Whalen. We also thank Duane Wessels, Casey Deccio, Cynthia Taylor, and Stephen Checkoway for their feedback on the paper, Philip Stark for suggestions about the analysis, and Collin Jackson for his help in acquiring an advertising network advertiser account. This material is based upon work supported by the MURI program under AFOSR Grant No. FA9550-08-1-0352.

References

- [1] Distributed director cache auto refresh. https://www.cisco.com/en/US/docs/ios/12_2t/12_2t8/feature/guide/ftrefrsh.pdf.
- [2] B. Ager, H. Dreger, and A. Feldmann. Predicting the DNSSEC overhead using DNS traces. In R. Calderbank and H. Kobayashi, editors, *Proceedings of CISS 2006*, pages 1484–89. IEEE Information Theory Society, Mar. 2006.
- [3] B. Ager, W. Mühlbauer, G. Smaragdakis, and S. Uhlig. Comparing DNS resolvers in the wild. In M. Allman, editor, *Proceedings of IMC 2010*, pages 15–21. ACM Press, Nov. 2010.
- [4] R. Arends, R. Austein, M. Larson, D. Massey, and S. Rose. DNS Security Introduction and Requirements. RFC 4033 (Proposed Standard), Mar. 2005. Updated by RFC 6014.
- [5] R. Arends, R. Austein, M. Larson, D. Massey, and S. Rose. Protocol Modifications for the DNS Security Extensions. RFC 4035 (Proposed Standard), Mar. 2005. Updated by RFCs 4470, 6014.
- [6] A. Barth, C. Jackson, and J. C. Mitchell. Robust defenses for cross-site request forgery. In P. Syverson and S. Jha, editors, *Proceedings of CCS 2008*, pages 75–88. ACM Press, Oct. 2008.
- [7] R. Bellis. DNS Proxy Implementation Guidelines. RFC 5625 (Best Current Practice), Aug. 2009.
- [8] R. Bellis. DNS Transport over TCP - Implementation Requirements. RFC 5966 (Proposed Standard), Aug. 2010.
- [9] R. Bellis and L. Phifer. Test report: DNSSEC impact on broadband routers and firewalls. Online: <https://www.dnssec-deployment.org/wp-content/uploads/2010/03/DNSSEC-CPE-Report.pdf>, Sept. 2008.
- [10] R. Braden. Requirements for Internet Hosts - Application and Support. RFC 1123 (Standard), Oct. 1989. Updated by RFCs 1349, 2181, 5321, 5966.
- [11] Caida routevviews prefix to as mappings dataset (pfx2as). <http://www.caida.org/data/routing/routevviews-prefix2as.xml>.

- [12] E. Cohen and H. Kaplan. Proactive caching of DNS records: Addressing a performance bottleneck. *Computer Networks*, 41(6):707–26, 2003.
- [13] What are comcast's dynamic ip ranges? <http://postmaster.comcast.net/dynamic-IP-ranges.html>.
- [14] C. Deccio, J. Sedayao, K. Kant, and P. Mohapatra. A case for comprehensive DNSSEC monitoring and analysis tools. In R. Clayton, editor, *Proceedings of SATIN 2011*, Apr. 2011. Online: <http://conferences.npl.co.uk/satin/agenda2011.html>.
- [15] C. Deccio, J. Sedayao, K. Kant, and P. Mohapatra. Quantifying and improving DNSSEC availability. In G. Rouskas and X. Zhou, editors, *Proceedings of ICCCN 2011*. IEEE Communications Society, July 2011.
- [16] T. Dietrich. DNSSEC support by home routers in Germany. Presented at RIPE 60, May 2010. Online slides: http://ripe60.ripe.net/presentations/Dietrich-DNSSEC_Support_by_Home_Routers_in_Germany.pdf.
- [17] W. J. Glynn. Measuring DNS vulnerabilities and DNSSEC challenges from an irish perspective. In R. Clayton, editor, *Proceedings of SATIN 2011*, Apr. 2011. Online: <http://conferences.npl.co.uk/satin/agenda2011.html>.
- [18] Ó. Guðmundsson and S. D. Crocker. Observing DNSSEC validation in the wild. In R. Clayton, editor, *Proceedings of SATIN 2011*, Apr. 2011. Online: <http://conferences.npl.co.uk/satin/agenda2011.html>.
- [19] A. Herzberg and H. Shulman. Towards adoption of dnssec: Availability and security challenges. Cryptology ePrint Archive, Report 2013/254, 2013. <http://eprint.iacr.org/>.
- [20] P. Hoffman and J. Schlyter. The DNS-Based Authentication of Named Entities (DANE) Transport Layer Security (TLS) Protocol: TLSA. RFC 6698 (Proposed Standard), Aug. 2012.
- [21] M. Honda, Y. Nishida, C. Raiciu, A. Greenhalgh, M. Handley, and H. Tokuda. Is it still possible to extend TCP? In P. Thiran and W. Willinger, editors, *Proceedings of IMC 2011*, pages 181–94. ACM Press, Nov. 2011.
- [22] L.-S. Huang, E. Y. Chen, A. Barth, E. Rescorla, and C. Jackson. Talking to yourself for fun and profit. In H. J. Wang, editor, *Proceedings of W2SP 2011*. IEEE Computer Society, May 2011.
- [23] G. Huston. Counting DNSSEC. Online: <https://labs.ripe.net/Members/gih/counting-dnssec>, Sept. 2012.
- [24] G. Huston and G. Michaelson. Measuring DNSSEC performance. Online: <http://www.potaroo.net/ispcol/2013-05/dnssec-performance.html>, May 2013.
- [25] C. Jackson, A. Barth, A. Bortz, W. Shao, and D. Boneh. Protecting browsers from DNS rebinding attacks. *ACM Trans. Web*, 3(1), Jan. 2009.
- [26] jquery: The write less, do more, javascript library. <http://jquery.com>.
- [27] C. Kreibich, B. Nechaev, N. Weaver, and V. Paxson. Netalyzr: Illuminating the edge network. In M. Allman, editor, *Proceedings of IMC 2010*, pages 246–59. ACM Press, Nov. 2010.
- [28] S. Krishnan and F. Monroe. An empirical study of the performance, security and privacy implications of domain name prefetching. In S. Bagchi, editor, *Proceedings of DSN 2011*, pages 61–72. IEEE Computer Society and IFIP, June 2011.
- [29] P. Leach, M. Mealling, and R. Salz. A Universally Unique Identifier (UUID) URN Namespace. RFC 4122 (Proposed Standard), July 2005.
- [30] J. Livingood. Comcast completes dnssec deployment. <http://blog.comcast.com/2012/01/comcast-completes-dnssec-deployment.html>.
- [31] D. Migault, C. Girard, and M. Laurent. A performance view on dnssec migration. In H. Lutfiyya and Y. Diao, editors, *Proceedings of CNSM 2010*, pages 469–74. IEEE Communications Society, Oct. 2010.
- [32] P. Mockapetris. Domain names - concepts and facilities. RFC 1034 (Standard), Nov. 1987. Updated by RFCs 1101, 1183, 1348, 1876, 1982, 2065, 2181, 2308, 2535, 4033, 4034, 4035, 4343, 4035, 4592, 5936.
- [33] P. Mockapetris. Domain names - implementation and specification. RFC 1035 (Standard), Nov. 1987. Updated by RFCs 1101, 1183, 1348, 1876, 1982, 1995, 1996, 2065, 2136, 2181, 2137, 2308, 2535, 2845, 3425, 3658, 4033, 4034, 4035, 4343, 5936, 5966, 6604.
- [34] nginx httplogmodule. <http://wiki.nginx.org/HttpLogModule>.
- [35] E. Osterweil, D. Massey, and L. Zhang. Deploying and monitoring DNS security (DNSSEC). In C. Payne and M. Franz, editors, *Proceedings of ACSAC 2009*, pages 429–38. ACM Press, Dec. 2009.
- [36] E. Osterweil, M. Ryan, D. Massey, and L. Zhang. Quantifying the operational status of the DNSSEC deployment. In K. Papagiannaki and Z.-L. Zhang, editors, *Proceedings of IMC 2008*, pages 231–42. ACM Press, Oct. 2008.
- [37] V. Pappas and A. D. Keromytis. Measuring the deployment hiccups of DNSSEC. In J. L. Mauri, T. Strufe, and G. Martinez, editors, *Proceedings of ACC 2011*, volume 192 of *CCIS*, pages 44–53. Springer-Verlag, July 2011.
- [38] S. Son and V. Shmatikov. The hitchhiker's guide to DNS cache poisoning. In S. Jajodia and J. Zhou, editors, *Proceedings of SecureComm 2010*, volume 50 of *LNICST*, pages 466–83. Springer-Verlag, Sept. 2010.
- [39] P. Vixie. Extension Mechanisms for DNS (EDNS0). RFC 2671 (Proposed Standard), Aug. 1999.
- [40] N. Weaver, C. Kreibich, B. Nechaev, and V. Paxson. Implementations of Netalyzr's DNS measurements. In R. Clayton, editor, *Proceedings of SATIN 2011*, Apr. 2011. Online: <http://conferences.npl.co.uk/satin/agenda2011.html>.
- [41] N. Weaver, C. Kreibich, and V. Paxson. Redirecting DNS for ads and profit. In N. Feamster and W. Lee, editors, *Proceedings of FOCI 2011*. USENIX, Aug. 2011.
- [42] W. C. Wijngaards and B. J. Overeinder. Securing DNS: Extending DNS servers with a DNSSEC validator. *Security & Privacy*, 7(5):36–43, Sept.–Oct. 2009.
- [43] H. Yang, E. Osterweil, D. Massey, S. Lu, and L. Zhang. Deploying cryptography in Internet-scale systems: A case study on DNSSEC. *IEEE Trans. Dependable and Secure Computing*, 8(5):656–69, Sept.–Oct. 2011.
- [44] C. Zhang, C. Huang, K. W. Ross, D. A. Maltz, and J. Li. Inflight modifications of content: Who are the culprits? In C. Kruegel, editor, *Proceedings of LEET 2011*. USENIX, Mar. 2011.
- [45] Z. Zhang, L. Zhang, D.-E. Xie, H. Xu, and H. Hu. A novel dns accelerator design and implementation. In C. S. Hong, T. Tonouchi, Y. Ma, and C.-S. Chao, editors, *Proceedings of APNOMS 2009*, volume 5787 of *LNCS*, pages 458–61. Springer-Verlag, Sept. 2009.

ExecScent: Mining for New C&C Domains in Live Networks with Adaptive Control Protocol Templates

Terry Nelms^{1,2}, Roberto Perdisci^{3,2}, and Mustaque Ahamed^{2,4}

¹Damballa, Inc.

²Georgia Institute of Technology – College of Computing

³University of Georgia – Dept. of Computer Science

⁴New York University Abu Dhabi

Abstract

In this paper, we present ExecScent, a novel system that aims to mine new, previously unknown C&C domain names from *live* enterprise network traffic. ExecScent automatically learns control protocol templates (CPTs) from examples of known C&C communications. These CPTs are then *adapted* to the “background traffic” of the network where the templates are to be deployed. The goal is to generate *hybrid* templates that can *self-tune* to each specific deployment scenario, thus yielding a better trade-off between true and false positives for a given network environment. To the best of our knowledge, ExecScent is the first system to use this type of adaptive C&C traffic models.

We implemented a prototype version of ExecScent, and deployed it in three different large networks for a period of two weeks. During the deployment, we discovered many new, previously unknown C&C domains and hundreds of new infected machines, compared to using a large up-to-date commercial C&C domain blacklist. Furthermore, we deployed the new C&C domains mined by ExecScent to six large ISP networks, discovering more than 25,000 new infected machines.

1 Introduction

Code reuse is common practice in malware [18, 20]. Often, new (polymorphic) malware releases are created by simply re-packaging previous samples, or by augmenting previous versions with a few new functionalities. Moreover, it is not uncommon for the source code of successful malware to be sold or leaked on underground forums, and to be reused by other malware operators [14].

Most modern malware, especially botnets, consist of (at least) two fundamental components: a *client agent*, which runs on victim machines, and a *control server* application, which is administered by the malware owner.

Because code reuse applies to both components¹, this naturally results in *many different malware samples sharing a common command-and-control (C&C) protocol*, even when control server instances owned by different malware operators use different C&C domains and IPs.

In this paper, we present ExecScent, a novel system that aims to mine new, previously unknown C&C domain names from *live* enterprise network traffic (see Figure 1). Starting from a *seed* list of known C&C communications and related domain names found in malware-generated network traces, ExecScent aims to discover new C&C domains by taking advantage of the commonalities in the C&C protocol shared by different malware samples. More precisely, we refer to the C&C protocol as the set of specifications implemented to enable the malware control application logic, which is defined at a higher level of abstraction compared to the underlying transport (e.g., TCP or UDP) or application (e.g., HTTP) protocols that facilitate the C&C communications. ExecScent aims to automatically learn the unique traits of a given C&C protocol from the seed of known C&C communications to derive a *control protocol template* (CPT), which can in turn be deployed at the edge of a network to detect traffic destined to new C&C domains.

ExecScent builds *adaptive* templates that also learn from the *traffic profile* of the network where the templates are to be deployed. The goal is to generate *hybrid* templates that can *self-tune* to each specific deployment scenario, thus yielding a better trade-off between true and false positives for a given network environment. The intuition is that different networks have different traffic profiles (e.g., the network of a financial institution may generate very different traffic compared to a technology company). It may therefore happen that a CPT could (by chance) raise a non-negligible number of false posi-

¹For example, web-based malware control panels can be acquired in the Internet underground markets and re-deployed essentially *as is*, while the client agents can be obtained using *do-it-yourself* malware creation kits [10].

tives in a given network, say Net_A , while generating true C&C domain detections and no false positives in other networks. We take a pragmatic approach, aiming to automatically identify these cases and lowering the “confidence” on that CPT *only* when it is deployed to Net_A . This allows us to lower the overall risk of false positives, while maintaining a high probability of detection in other networks. We further motivate the use of adaptive templates in Section 3.

ExecScent focuses on HTTP-based C&C protocols, because studies have shown that HTTP-based C&C communications are used by a large majority of malware families [26] and almost all known mobile bots [31]. Moreover, many enterprise networks employ strict egress filtering firewall rules that block all non-web traffic. This forces malware that target enterprise networks to use HTTP (or HTTPS) as the communication protocol of choice. It is also important to notice that many modern enterprise networks deploy web proxies that enforce SSL man-in-the-middle² (SSL-MITM). Therefore, enterprise networks can apply ExecScent’s templates at the web proxy level to discover new C&C domains even in cases of HTTPS-based C&C traffic.

Our system is different from previously proposed URL-based C&C signature generation systems [23, 30]. Unlike previous work, we build templates that can adapt to the deployment network and that model the entire content of HTTP requests, rather than being limited to the URL string. We show in Section 5 that this allows us to obtain a much better trade-off between true and false positives, compared to “statically” modeling only the URL of C&C requests.

Anomaly-based botnet detection systems such as [15, 16] typically require that more than one host in the monitored network be compromised with the same malware type, do not scale well to large networks, and are not able to directly attribute the suspected C&C communications to a specific malware family. Unlike these systems, ExecScent can detect C&C communications initiated by a single malware-infected machine with low false positives, and can attribute the newly discovered C&C domains to a known malware family name or malware operator (i.e., the name of the cyber-criminal group behind the malware operation). In turn, the *labeled* C&C domain names discovered by ExecScent may also be deployed in existing lightweight malware detection systems based on DNS traffic inspection, thus contributing to the detection and attribution of malware infections in very large networks (e.g., ISP networks) where monitoring all traffic may not be practically feasible.

Currently, we identify the seed of known C&C traffic required by ExecScent to learn the control protocol tem-

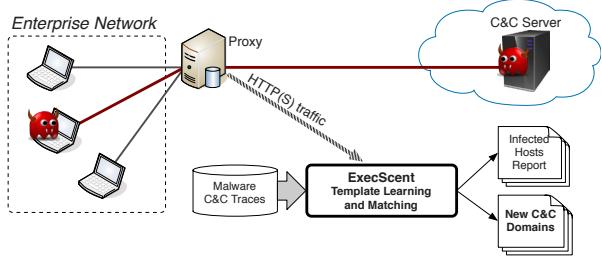


Figure 1: ExecScent deployment overview. Adaptive control protocol templates are learned from both malware-generated network traces and the live network traffic observation. The obtained adaptive templates are matched against new network traffic to discover new C&C domains.

plates by leveraging blacklists of known C&C domain names. C&C discovery systems based on dynamic analysis such as Jackstraws [17] may also be used for this purpose. However, unlike ExecScent, while Jackstraws may be useful to find “seed” C&C traffic, its system-call-based detection models [17] cannot be deployed to detect new C&C domains in live network traffic.

In summary, we make the following contributions:

- We present ExecScent, a novel system for mining new malware C&C domains from live networks. ExecScent automatically learns C&C traffic models that can adapt to the deployment network’s traffic. This *adaptive* approach allows us to greatly reduce the false positives while maintaining a high number of true positives. To the best of our knowledge, ExecScent is the first system to use this type of adaptive C&C traffic models.
- We implemented a prototype version of ExecScent, and deployed it in three different large networks for a period of two weeks. During the deployment, we discovered many new, previously unknown C&C domains and hundreds of new infected machines, compared to using a large up-to-date commercial C&C domain blacklist.
- We deployed the new C&C domains mined by ExecScent to six large ISP networks, discovering more than 25,000 new infected machines.

2 System Overview

The primary goal of ExecScent is to generate control protocol templates (CPTs) from a seed of known malware-generated HTTP-based C&C communications. We then use these CPTs to identify new, previously unknown C&C domains.

²See <http://crypto.stanford.edu/ssl-mitm/>, for example.

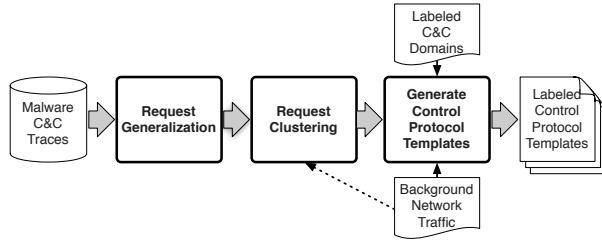


Figure 2: ExecScent system overview.

ExecScent automatically finds common traits among the C&C protocol used by different malware samples, and encodes these common traits into a set of CPTs. Each template is labeled with the name of the malware family or (if known) criminal operator associated with the C&C traffic from which the CPT is derived. Once a CPT is deployed at the edge of a network (see Figure 1), any new HTTP(S) traffic that matches the template is classified as C&C traffic. The domain names associated with the matched traffic are then flagged as C&C domains, and attributed to the malware family or operator with which the CPT was labeled.

Figure 2 presents an overview of the process used by ExecScent to generate and label the CPTs. We briefly describe the role of the different system components in this section, deferring the details to Section 4.

Given a large repository of malware-generated network traces, we first reconstruct all HTTP requests performed by each malware sample. Then, we apply a *request generalization* process, in which (wherever possible) we replace some of the request parameters (e.g., URL parameter values) with their data type and length, as shown in the example in Figure 3. Notice that ExecScent considers the entire content of the HTTP requests, not only the URLs (see Section 3.2), and the generalization process is applied to different parts of the request header. The main motivation for applying the generalization step is to improve the accuracy of the *request clustering* process, in which we aim to group together malware-generated requests that follow a similar C&C protocol.

Once the malware requests have been clustered, we apply a *template learning* process in which we derive the CPTs. Essentially, a CPT summarizes the (generalized) HTTP requests grouped in a cluster, and records a number of key properties such as the structure of the URLs, the set of request headers, the IP addresses contacted by the malware, etc. Furthermore, the templates associate a malware-family label to each template (see Section 4.4 for details).

Before the templates are deployed in a network, we *adapt* the CPTs to the “background traffic” observed in

that network. In particular, for each template component (e.g., the generalized URL path, the user-agent string, the request header set, etc.), we compute how frequently the component appeared in the deployment network. CPT components that are “popular” in the background traffic will be assigned a lower “match confidence” for that network. On the other hand, components that appear very infrequently (or not at all) in the traffic are assigned a higher confidence. We refer to these “rare” components as having high *specificity*, with respect to the deployment network’s traffic. The intuitions and motivations for this approach are discussed in more detail in the next section.

After deployment, an HTTP request is labeled as C&C if it matches a CPT with *high similarity and specificity*. That is, if the request closely matches a CPT and the matching CPT components have high specificity (i.e., rarely appeared) in that particular deployment network.

3 Approach Motivations and Intuitions

In this section, we discuss the intuitions that motivated us to build adaptive control protocol templates. Furthermore, we discuss the advantages of considering the entire content of C&C HTTP requests, rather than limiting ourselves to the URL strings, as done in previous work [23, 30].

3.1 Why Adaptive Templates?

As most other traffic models, ExecScent’s CPTs, which are derived from and therefore can match C&C traffic, may be imperfect and could generate some false positives. To minimize this risk, ExecScent builds *adaptive* control protocol templates that, besides learning from known malware-generated C&C traffic, also learn from the traffic observed in the network where the templates are being deployed. Our key observation is that different enterprise networks have different traffic profiles. The traffic generated by the computer network of a financial institute (e.g., a large bank) may look quite different from traffic at a manufacturing company (e.g., a car producer) or a technology company (e.g., a software-development company). It may therefore happen that a CPT could (by chance) raise a non-negligible number of false positives in a given network, say Net_X , and several true detections with no or very few false positives in other networks. Intuitively, our objective is to automatically identify these cases, and lower the “confidence” on that template when it matches traffic from Net_X , while keeping its confidence high when it is deployed elsewhere.

For example, assume Net_B is a US bank whose hosts have rarely or never contacted IPs located, say, in China. If an HTTP request towards an IP address in China is found, this is by itself an anomalous event. Intuitively, if

the request also matches a CPT, our confidence on a correct match (true C&C communication) can be fairly high. On the other hand, assume Net_A is a car manufacturer with partners in China, with which Net_A 's hosts communicate frequently. If an HTTP request in Net_A matches a CPT but is directed towards an address within one of the IP ranges of the manufacturer's partners, our confidence on a correct match should be lowered.

More specifically, consider the following hypothetical scenario. Assume we have a template τ that matches an HTTP request in both Net_A and Net_B with a *similarity* score s . For simplicity, let us assume the score s is the same for both Net_A 's traffic and Net_B 's traffic. Suppose also that the server's IP (or its /24 prefix) associated with the matching traffic is ip_a for Net_A and ip_b for Net_B . Also, suppose that ip_a is “popular” in network Net_A , whereas ip_b has very low popularity in Net_B because it has never been contacted by hosts in that network. Because ip_a is very popular in Net_A , meaning that a large fraction (e.g., more than 50%) of the hosts in Net_A has contacted the domain in the past, it is likely that the template τ is fortuitously matching benign traffic, thus potentially causing a large number of false positives in Net_A . On the other hand, because ip_b has very low popularity in Net_B , it is more likely that the match is a true detection, or that in any case τ will generate very few (potentially only one) false positives in Net_B . Consequently, based on a model of recent traffic observed in Net_A and Net_B , we should lower our confidence in τ for the matches observed in Net_A , but not for Net_B . In other words, τ should automatically *adapt* to Net_A to “tune down” the false positives. At the same time, keeping the confidence in τ high for Net_B means that we will still be able to detect C&C communications that match τ , while keeping the risk of false positives low. We generalize this approach to *all* other components of ExecScent's templates (e.g., the structure of the URLs, the user-agent strings, the other request headers, etc.), in addition to the destination IPs.

Overall, our confidence on a match of template τ in a given network Net_X will depend on two factors:

- *Similarity*: a measure of how closely an HTTP request matches τ .
- *Specificity*: a measure of how specific (or *rare*) are the components of τ with respect to Net_X 's traffic.

An HTTP request is labeled as C&C if it matches a CPT with both high similarity and high specificity. We show in Section 5 that this approach outperforms C&C models that do not take such specificity into account.

3.2 Why Consider All Request Content?

Malware C&C requests typically need to carry enough information for a malware agent running on a victim to (loosely) authenticate itself with the C&C server. Intuitively, the C&C server wants to make sure that it is talking to one of its *bots*, thus avoiding exposure of its true nature or functionalities to crawlers or security researchers who may be probing the server as part of an investigation. This is often achieved by using a specific set of parameter names and values that must be embedded in the URL for the C&C requests to be successful. Previous work on automatic URL signature generation has shown promising results in such cases [23, 30]. However, some malware (e.g., TDL4 [1]) exchanges information with the C&C by first encrypting it, encoding it (e.g., using base-64 encoding), and embedding it in the URL path. Alternatively, identifier strings can also be embedded in fields such as `user-agent` (e.g., some malware samples use their MD5 hash as `user-agent` name), encoded in other request headers (e.g., in the `referrer`), or in the body of POST requests. Therefore, only considering URLs may not be enough to accurately model C&C requests and detect new C&C domains, as supported by our experimental results (Section 5).

4 System Details

We now detail the internals of ExecScent. Please refer to Section 2 for a higher-level overview of the entire system.

4.1 Input Network Traffic

As we mentioned in Section 1, ExecScent focuses on HTTP-based malware, namely malware that leverage HTTP (or HTTPS) as a base network protocol on top of which the malware control protocol is “transported”. To this end, ExecScent takes in as input a feed of malware-generated HTTP traffic traces (in our evaluation, we use a large set of malware traces provided to us by a well-known company that specializes in malware defense).

It is worth remembering that while some malware may use HTTPS traffic as a way to evade detection, this does not represent an insurmountable obstacle in our deployment scenarios (see Figure 1). In fact, many enterprise networks, which represent our target deployment environment, already deploy web proxy servers that can perform SSL-MITM and can therefore forward the clear-text HTTP requests to ExecScent's template matching module, e.g., using the ICAP protocol (RFC 3507). Also, malware samples that appear to be using HTTPS traffic may be re-analyzed in a controlled environment that includes an SSL-MITM proxy interposed between the (vir-

tual) machine running the sample and the egress router. After all, HTTPS-based malware that do not support or choose not to run when an SSL-MITM proxy is present will also fail to run in enterprise networks that have a similar setting, and are therefore of less interest.

4.2 Request Generalization

As we discuss in the following sections, to obtain quality control protocol templates we first need to group similar C&C requests. To this end, an appropriate similarity metric needs to be defined before clustering algorithms can be applied. Previous works that propose URL-centric clustering systems [23,30] are mainly based on string similarity measures. Essentially, two URLs are considered similar if they have a small edit distance, or share a number of substrings (or tokens). However, these systems do not take into account the fact that URLs often contain variables whose similarity is better measured according to their data *type* rather than considering specific sequences of characters. Consider the two hypothetical C&C requests in Figure 3. Taken as they are (Figure 3a), their distance is relatively large, due to the presence of several different characters in the strings. To avoid this, ExecScent uses a set of heuristics to detect strings that represent data of a certain type, and replaces them accordingly using a placeholder tag containing the data type and string length (Figure 3b).

For example, we would identify “fa45e” as lowercase hexadecimal because it contains numeric characters and the alphabetic characters are all valid lowercase hexadecimal digits. The data types we currently identify are *integer*, *hexadecimal* (upper, lower and mixed case), *base64* (standard and “URL safe”) and *string* (upper, lower and mixed case). In addition, for integer, hexadecimal and string we can identify the data type plus additional punctuation such as “.” or “:” (e.g., 192.168.1.1 would be identified as a data type of integer+period of length 11). Furthermore, our heuristics can easily be extended to support data types such as IP address, MAC address, MD5 hash and version number.

This generalization process allows us to define a better similarity metric (Section 4.7), which is instrumental to obtaining higher quality C&C request clusters. Notice also that while previous works such as [23,30] focus only on URL strings, ExecScent takes the entire request into account. For example, in Figure 3 the user-agent strings are MD5s, and can be generalized by replacing the specific MD5 strings with the appropriate data type and length information.

(a)	Request 1: GET /m90bmV0DQo=/cnc.php?v=121&cc=IT Host: www.bot.net User-Agent: 680e4a9a7eb391bc48118baba2dc8e16 ... Request 2: GET /bWFsd2FyZQ0KDQo=/cnc.php?v=425&cc=US Host: www.malwa.re User-Agent: dae4a66124940351a65639019b50bf5a ...
(b)	Request 1: GET /<Base64;12>/cnc.php?v=<Int;3>&cc=<Str;2> Host: www.bot.net User-Agent: <Hex;32> ... Request 2: GET /<Base64;16>/cnc.php?v=<Int;3>&cc=<Str;2> Host: www.malwa.re User-Agent: <Hex;32> ...

Figure 3: Example C&C requests: (a) original; (b) generalized.

4.3 Request Clustering

Before extracting the templates, we group together similar C&C requests. This clustering step simply aims to assist the automatic CPT generation algorithm, improving efficiency and yielding templates that are at the same time *generic* enough to match similar (but not identical) C&C communications in new traffic, and *precise* enough to generate very few or no false positives.

We perform C&C request clustering in two phases. During the first phase, we coarsely group C&C requests based on their destination IPs. Specifically, given two C&C requests, we group them together if their destination IPs reside in /24 (or class C) networks that *share a DNS-based relationship*. Namely, we consider two /24 networks as related if there exists at least one domain name that within the last 30 days resolved to different IP addresses residing in the two different networks. To find such relationships, we rely on a large passive DNS database [12].

In the second phase, we consider one coarse-grained cluster at a time, and we further group a cluster’s C&C requests according to a content similarity function. We use an agglomerative hierarchical clustering algorithm to group together C&C requests within a coarse-grained cluster that carry similar generalized URLs, similar user-agent strings, similar numbers of HTTP header fields and respective values, etc. When measuring the similarity between two requests, we take into account both the *similarity* and *specificity* of the requests’ content, where the specificity (or low “popularity”) can be measured with respect to a dataset of traffic recently collected from different networks (dashed arrow in Figure 2). For a more detailed definition of the similarity

T1) Median URL path: <Base64:14>/cnc.php
T2) URL query component: {v=<Int:3>, cc=<String:2>}
T3) User Agent: {<Hex:32>}
T4) Other headers: {((Host:13), (Accept-Encoding:8))}
T5) Dst nets: {172.16.8.0/24, 10.10.4.0/24, 192.168.1.0/24}
Malware family: {Trojan-A, BotFamily-1}
 URL regex: GET /.*?(cclv)=
Background traffic profile: specificity scores used to adapt the CPT to the deployment environment

Figure 4: Example CPT.

function used in the clustering step, we refer the reader to Section 4.7.

4.4 Generating CPTs

Once C&C requests have been clustered, a control protocol template (CPT) is generated from each cluster. At this stage, we consider only clusters that contain at least one HTTP request to a known C&C domain. Each template represents a summary of all C&C requests in a cluster, and contains the following components, as also shown in Figure 4:

- τ₁) *Median URL path*: median path string that minimizes the sum of edit distances from all URL paths in the requests (see [11] for a definition of median string). *Intuition*: although the URL path may vary significantly from one malware installation to another, we observed many cases in which there exist “stable” path components that are unique to a specific malware family or operation.
- τ₂) *URL query component*: stores the set of parameter names, value types and lengths observed in the query component [5] of each of the URLs. *Intuition*: URL parameters are often used by malware to convey information about the infected host, such as its OS version, a unique identifier for the infected machine, etc.
- τ₃) *User-agent*: the set of all different (generalized) user-agent strings found in the requests. *Intuition*: the user-agent is one of the most abused HTTP headers by malware, and is sometimes used as a loose form of authentication.
- τ₄) *Other headers*: the set of other HTTP headers observed in the requests. For each header, we also store the length of its value string. *Intuition*: the set of header names, their order and values are sometimes unique to a malware family.

τ₅) *Dst. networks*: the set of all destination /24 networks associated with the C&C requests in the cluster. *Intuition*: in some cases, the C&C server may be relocated to a new IP address within the same (possibly “bullet-proof”) network.

- *Malware family*: the (set of) malware family name(s) associated to the known C&C requests in the cluster.

In addition, each CPT includes the following deployment-related information:

- *URL regex*: to increase the efficiency of the template matching phase (Section 4.6), each template includes a regular expression automatically generated from the set of URL strings in the requests. The URL regex is intentionally built to be very generic and is used during deployment for the sole purpose of filtering out traffic that is extremely unlikely to closely match the entire template, thus reducing the cost of computing the similarity between HTTP requests in live traffic and the template.
- *Background traffic profile*: information derived from the traffic observed in the deployment environment within the past W days (where W is a system parameter). This is used for computing the specificity of the CPT components, thus allowing us to adapt the CPT to the deployment network, as explained in detail in Section 4.5.

Notice that a CPT acts as the *centroid* for the cluster from which it was derived. To determine if a new request is similar enough to a given cluster, we only need to compare it with the CPT, rather than all of the clustered C&C requests. Therefore, CPTs provide an efficient means of measuring the similarity of a new request to the C&C protocol used by the clustered malware samples.

4.5 Adapting to a Deployment Network

As explained in Section 3.1, once the CPTs are deployed, an HTTP request is labeled as C&C if it matches a CPT τ with both high *similarity* and *specificity*. To this end, we first need to compute a specificity score for each element of the k -th component τ_k of τ , which indicates how “unpopular” that element is with respect to the traffic profile in the deployment network (notice that $k = 1, \dots, 5$, as shown in Figure 4 and Section 4.4).

For example, to compute the specificity scores for τ_3 , we first compute a *host-based popularity* score hp_{ua_i} for each user-agent string ua_i in the set τ_3 . We consider the number of hosts hn_{ua_i} in the deployment network that generated an HTTP request containing ua_i during the last

W days, where W is a configurable time-window parameter. We define $hp_{ua_i} = \frac{hn_{ua_i}}{\max_j\{hn_{ua_j}\}}$, where the max is taken over all user-agent strings ua_j observed in the deployment network’s traffic. Similarly, we compute a *domain-based popularity* score dp_{ua_i} , based on the number of distinct destination domain names dn_{ua_i} with one or more HTTP requests that contain ua_i . We define $dp_{ua_i} = \frac{dn_{ua_i}}{\max_j\{dn_{ua_j}\}}$. The intuition is that a user-agent string can only be considered truly popular if it spans many hosts and domains. On the other hand, we do not want to consider a ua_i as very popular if it has high host-based popularity (e.g., “Windows-Update-Agent”) but low domain-based popularity (e.g., because the only domain on which it is used is `microsoft.com`). Finally, we define the specificity score for ua_i as $\sigma_{3,ua_i} = 1 - \min(hp_{ua_i}, dp_{ua_i})$. In a similar way, we compute a specificity score σ_{4,hd_i} for each header element hd_i in τ_4 .

To compute the specificity scores for τ_5 , we simply compute the host-based popularity hp_{net_i} for each /24 network prefix $net_i \in \tau_5$, and we define a separate score $\sigma_{5,net_i} = (1 - hp_{net_i})$ for each prefix.

4.5.1 URL Specificity

Computing the specificity of the components of a URL is more complex, due to the large variety of unique URLs observed every day on a given network. To address this problem, we rely on a supervised classification approach. First, given a dataset of traffic collected from a large network, we extract all URLs, and learn a *map of URL word frequencies*, where the “words” are extracted by tokenizing the URLs (e.g., extracting elements of the URL path, filename, query string, etc.). Then, given a new URL, we translate it into a feature vector in which the statistical features measure things such as the average frequency of single “words” in the tokenized URL, the average frequency of word bigrams in the query parameters, the frequency of the file name, etc. (to extract the frequency values for each word found in the URL we lookup the previously learned map of word frequencies).

After we translate a large set of “background traffic URLs” into feature vectors, we train an SVM classifier [8] that can label new URLs as either *popular* or *unpopular*. To prepare the training dataset we proceed as follows. We first rank the “background URLs” according to their domain-based popularity (i.e., URLs that appear on requests to multiple sites on different domain names are considered as more popular). Then, we take a sample of URLs from the top and from the bottom of this ranking, which we label as *popular* and *unpopular*, respectively. We use this labeled dataset to train the SVM classifier, and we rely on the max-margin approach used by the SVM [9] to produce a model that can generalize

to URLs not seen during training.

During the operational phase (once the SVM classifier is trained and deployed), given a URL u_i , we can first translate u_i into its corresponding feature vector v_i , as described above, and feed v_i to the SVM classifier. The classifier can then label u_i as either *popular* or *unpopular*. In practice, though, rather than considering these class labels, we only take into account the classification score (or confidence) associated with the *popular* class³. Therefore, the SVM’s output can be interpreted as follows: the higher the score, the more u_i “looks like” a popular URL, when compared to the large set of URLs observed in the background traffic. Finally, the specificity score for the URL is computed as $\sigma_{u_i} = 1 - p_{u_i}$, where p_{u_i} is the SVM output for URL u_i .

Now, let us go back to consider the template τ and its URL-related components τ_1 and τ_2 (see Figure 4). We first build a “median URL” u_m by concatenating the median URL path (τ_1) to the (sorted) set of generalized parameter names and values (τ_2). We then set the similarity scores $\sigma_1 = \sigma_2 = \sigma_{u_m}$, where σ_{u_m} is the specificity of u_m .

4.6 Template Matching

Template matching happens in two phases. As mentioned above, each template contains an URL regular expression automatically derived from the C&C requests in a cluster. Given a new HTTP request r , to test whether this request matches a template τ , we first match r ’s URL to τ ’s URL regex. It is worth noting that, as mentioned in Section 4.4, the URL regex is intentionally built to be very generic, and is merely used to efficiently filter out traffic that is extremely unlikely to match the entire template. Furthermore, we check if the destination IP of r resides within any of the /24 prefixes in τ (specifically in component τ_5). If neither the URL regex nor the destination IP have a match, we assume r does not match τ . Otherwise, we proceed by considering the entire content of request r , transforming r according to the request generalization process (see Section 4.2), and measuring the overall *matching score* $S(r, \tau)$ between the (generalized) request r and the template τ .

In summary, the score S is obtained by measuring the similarity between all the components of the request r and the respective components of the template τ . These similarity measures are then weighted according to their specificity, and the matching score $S(r, \tau)$ is computed as the average of all weighted component similarities. A detailed definition of the similarity functions and how specificity plays an explicit role in computing $S(r, \tau)$ is given in Section 4.7.

³We calibrate the classification scores output by the SVM classifier using the method proposed by Platt [24].

If $S(r, \tau)$ exceeds a tunable detection threshold θ , then the request r will be deemed a C&C request and the domain name associated with r (assuming r is not using a hardcoded IP address) is classified as C&C domain and labeled with the malware family associated to τ . Furthermore, the host from which the request r originated is labeled as compromised with τ 's malware family.

4.7 Similarity Functions

4.7.1 CPT matching score

To determine if a new HTTP request r matches a CPT τ , we compute a *matching score* $S(r, \tau)$ as follows:

$$S(r, \tau) = \frac{\sum_k w_k(s_k, \sigma_k) \cdot s_k(r_k, \tau_k)}{\sum_k w_k(s_k, \sigma_k)} \cdot \sigma_d \quad (1)$$

where s_k is a similarity function that compares each element τ_k of τ (Section 4.4) with its respective counterpart r_k of r , and where w_k is a *dynamic weight* (whose definition is given below) that is a function of both the similarity s_k and the specificity σ_k of the k -th component of τ . The denominator scales $S(r, \tau)$ between zero and one.

The factor σ_d is the *specificity of the destination domain* d of request r , which is computed as $\sigma_d = 1 - \frac{m_d}{\max_i\{m_{d_i}\}}$, where m_d is the number of hosts in the deployment network's traffic that queried domain d , and $\max_i\{m_{d_i}\}$ is the number of hosts that queried the most “popular” domain in the traffic. Accordingly, we use σ_d to decrease the matching score $S(r, \tau)$ for low-specificity domains (i.e., domains queried by a large number of hosts). The intuition is that infections of a specific malware family often affect a relatively limited fraction of all hosts in an enterprise network, as most modern malware propagate relatively “slowly” via drive-by downloads or social engineering attacks. In turn, it is unlikely that a new C&C domain will be queried by a very large fraction (e.g., > 50%) of all hosts in the monitored network, within a limited amount of time (e.g., one day).

In the following, we describe the details of the similarity functions $s_k(\cdot)$ used in Equation 1. In addition, we further detail how the specificity value of each component is selected, once the value of $s_k(\cdot)$ has been computed (for the definition of specificity, we refer the reader to Section 4.5).

s_1 - Given the path of the URL associated with r , we measure the normalized edit distance between the path and the CPT's median URL path τ_1 . The URL path specificity σ_1 is computed as outlined in Section 4.5.

s_{2a} - We measure the Jaccard similarity⁴ between the set of parameter names in the URL query-string of r

⁴ $J = \frac{|A \cap B|}{|A \cup B|}$

and the set of names in τ_2 . The specificity of the parameter names σ_{2a} is equal to σ_2 (see Section 4.5).

s_{2b} - We compare the data types and lengths of the values in the generalized URL query-string parameters (see Section 4.2). For each element of the query string, we assign a score of one if its data type in r matches the data type recorded in τ_2 . Furthermore, we compute the ratio between the value length in r and in τ_2 . Finally, s_{2b} is computed by averaging all these scores, whereby the more data types and lengths that match, the higher the similarity score. As in s_{2a} , we set $\sigma_{2b} = \sigma_2$.

s_3 - We compute the normalized edit distance between the (generalized) user-agent string in r , and each of the strings in the set τ_3 . Let d_m be the smallest of such distances, where m is the closest of the template's user-agent strings. We define $s_3 = 1 - d_m$, and set the specificity $\sigma_3 = \sigma_{3,m}$.

s_4 - Given the remaining request header fields in r , we measure the similarity from different perspectives. First, we compute the Jaccard similarity j between the set of headers in r and the set τ_4 . Furthermore, we consider the order of the headers as they appear in r and in the requests from which τ was derived. If the order matches, we set a variable $o = 1$, otherwise we set $o = 0$. Finally, for each header, we compare the ratio between the length of its value as it appears in r and in τ_5 , respectively. The similarity s_4 is defined as the average of all these partial similarity scores (i.e., of j , o , and the length ratios). We set the specificity score $\sigma_5 = \min_l\{\sigma_{5,hd_l}\}$, where the hd_l are the request headers.

s_5 - Let ρ be the destination IP of request r . If ρ resides within any of the /24 network prefixes in τ_5 , we set $s_5 = 1$, otherwise we assign $s_5 = 0$. Assume ρ is within prefix $n \in \tau_5$ (in which case $s_5 = 1$). In this case, we set the specificity $\sigma_5 = \sigma_{5,n}$.

The dynamic weights $w_k(\cdot)$ are computed as follows:

$$w_k(s_k, \sigma_k) = \hat{w}_k \cdot \left(1 + \frac{1}{(2 - s_k \cdot \sigma_k)^n}\right) \quad (2)$$

where \hat{w}_k is a *static weight* (i.e., it takes a fixed value), and n is a configuration parameter. Notice that $w_k \in [\hat{w}_k(1 + \frac{1}{2^n}), 2\hat{w}_k]$, and that these weights are effectively normalized by the denominator of Equation 1, thus resulting in $S(r, \tau) \in [0, 1]$ (since $s_k \in [0, 1], \forall k$, and $\sigma_d \in [0, 1]$, by definition).

The intuition for the dynamic weights $w_k(\cdot)$ is that we want to give higher weight to components of a request r that match their respective counterpart in a CPT τ with

both high similarity and high specificity. In fact, the weight will be maximum when both the similarity and specificity are equal to one, and will tend to the minimum when either the similarity or specificity (or both) tend to zero.

In summary, *similarity measures the likeness of two values*, whereas *specificity measures their uniqueness* in the underlying network traffic. The dynamic weights allow us to highlight the *rare structural elements* that are common between a CPT and a request, so that we can leverage them as the dominant features for detection. Because rare structural elements differ in their importance across malware families, by emphasizing these “unique features” we are able to detect and distinguish between different malware families.

4.7.2 Similarity function for clustering phase

In Section 4.3, we have described the C&C request clustering process. In this section we define the function used to compute the similarity between pairs of HTTP requests, which is needed to perform the clustering.

Given two HTTP requests r_1 and r_2 , we compute their similarity using Equation 1. At this point, the reader may notice that Equation 1 is defined to compare an HTTP request to a CPT, rather than two requests. The reason why we can use Equation 1, is that we can think of a request as a CPT derived from only one HTTP request. Furthermore, if we want to include the specificity scores, which are used to make the weights w_k dynamic, we can use a dataset of traffic previously collected from one or more networks (see dashed arrow in Figure 2).

5 Evaluation

In this section, we describe the data used to evaluate ExecScent (Section 5.1), how the system was setup to conduct the experiments (Section 5.2), and present the experimental results in different live networks (Section 5.3). Furthermore, we quantify the advantage of modeling entire HTTP requests, rather than only considering URLs, and of using adaptive templates over “static” C&C models (Section 5.4). In addition, we show the benefits obtained by deploying new C&C domains discovered by ExecScent into large ISP networks (Section 5.5).

5.1 Evaluation Data

5.1.1 Malware Network Traces

We obtained access to a commercial feed of malware intelligence data (provided to us by a well known security

company), which we used to generate the control protocol templates (CPTs). Through this feed, we collected about 8,000 malware-generated network traces per day that contained HTTP traffic. Each network trace was marked with a hash of the malware executable that generated the network activity, and (if known) by the related malware family name.

5.1.2 Live Network Traffic

To evaluate ExecScent, we had access to the *live* traffic of three large production networks, which we refer to as UNETA, UNETB, and FNET. Networks UNETA and UNETB are two different academic networks based in the US, while FNET is the computer network of a large North-American financial institution. Table 1 reports statistics with respect to the network traffic observed in these three networks. For example, in UNetA we observed an average of 7,893 distinct active source IP addresses per day. In average, these network hosts generated more than 34.8M HTTP requests per day, destined to 149,481 different domain names (in average, per day).

Table 1: Live Network Traffic Statistics (Avg. per day)

	UNETA	UNETB	FNET
Distinct Src IPs	7,893	27,340	7,091
HTTP Requests	34,871,003	66,298,395	58,019,718
Distinct Domains	149,481	238,014	113,778

5.1.3 Ground Truth

To estimate true and false positives, we rely on the following data:

- CCBL: we obtained a large black-list containing hundreds of thousands of C&C domains provided by a well known security company, which we refer to as CCBL. It is worth noting that CCBL is different from most publicly available domain blacklists for two reasons: 1) the C&C domains are carefully vetted by professional threat analysts; 2) the domains are labeled with their respective malware families and, when available, a malware operator name (i.e., an identifier for the cyber-criminal group that operates the C&C).
- ATWL: we derived a large white-list of *benign* domain names from Alexa’s top 1 million global domains list (alexa.com). From these 1M domains, we filtered out domains that can be considered as *effective* top level domains⁵ (TLDs), such as domains related to dynamic DNS services (e.g., dyn-dns.org, no-ip.com, etc.). Next, we discarded domains that have not been in the top 1M list for at

⁵<http://publicsuffix.org>

least 90% of the time during the entire past year. To this end, we collected an updated top domains list every day for the past year, and only considered as benign those domains that have *consistently* appeared in the top 1M domains list. The purpose of this filtering process is to remove possible noise due to malicious domains that may became popular for a limited amount of time. After this pruning operations, we were left with about 450,000 popular domain names⁶.

- **PKIP:** we also maintain a list of parking IPs, PKIP. Namely, IP addresses related to *domain parking* services (e.g., IPs pointed to by expired or unused domains which have been temporarily taken over by a registrar). We use this list to prune ExecScent’s templates. In fact, CPTs are automatically derived from HTTP requests in malware-generated network traces that are labeled as C&C communications due to their associated domain name being in the CCBL list (Section 4). However, some of the domains in CCBL may be expired, and could be currently pointing to a parking site. This may cause some of the HTTP requests in the malware traces to be erroneously labeled as C&C requests, thus introducing noise in ExecScent’s CPTs. We use the PKIP to filter out this noise.
- **Threat Analysis:** clearly, it is not feasible to obtain complete ground truth about all traffic crossing the perimeter of the live networks where we evaluated ExecScent. To compensate for this and obtain a better estimate of the false and true positives (compared to only using CCBL and ATWL), we performed an extensive manual analysis of our experimental results with the help of professional threat analysts.

5.2 System Setup

To conduct our evaluation, we have implemented and deployed a Python-based proof-of-concept version of ExecScent. In this section we discuss how we prepared the system for live network deployment.

5.2.1 Clustering Parameters

As discussed in Section 4.3, to generate the CPTs, we first apply a request clustering step. The main purpose of this step is to improve the efficiency of the CPT learning process. The clustering phase relies on a hierarchical clustering algorithm that takes in as input the height at which the dendrogram (i.e., the “distance tree” generated

⁶More precisely, second level domains (2LDs).

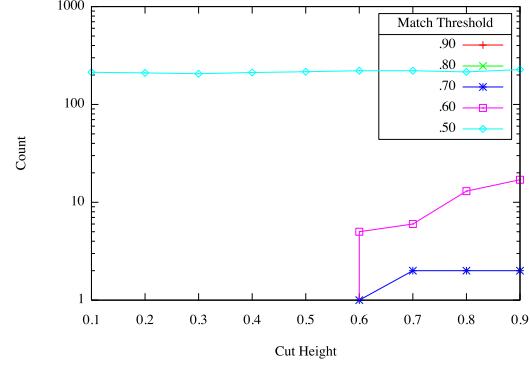


Figure 5: Effect of the dendrogram cut height (FPs).

by the clustering algorithm) needs to be cut to partition the HTTP requests into request clusters.

To select the dendrogram *cut height*, we proceeded as follows. We considered one day of malware traces collected from our malware intelligence feed (about 8,000 different malware traces). We then applied the clustering process to these traces, and produced different clustering results by cutting the dendrogram at different heights. For each of these different clustering results, we extracted the related set of CPTs, and we tested these CPTs over the next day of malware traces from our feed with varying matching thresholds. The obtained number of false positives, i.e., misclassified benign domains (measured using ATWL), and true positives, i.e., new correctly classified C&C domains (measured using CCBL), are summarized in Figure 5 and Figure 6, respectively. Notice that although in this phase we tested the CPTs over malware-generated network traces, we can still have false positives due to the fact that some malware query numerous benign domain names, along with C&C domains.

As Figures 5 and 6 show, per each fixed CPT matching threshold, varying the dendrogram cut height does not significantly change the false positives and true positives. In other words, the CPT matching results are not very sensitive to the specific value of the clustering parameter. We decided to finally set the value of the cut height to 0.38, which we use during all remaining experiments, because this provided good efficiency during the CPT generation process, while maintaining high CPT quality.

5.2.2 CPT Generation

To generate the CPTs used for the evaluation of ExecScent on live network traffic (Section 5.3), we initially used two weeks of malware traces collected from our malware intelligence feed. To label the *seed* of C&C HTTP requests in the malware traces, we used the CCBL black-list. We also use the list of parking IPs PKIP to

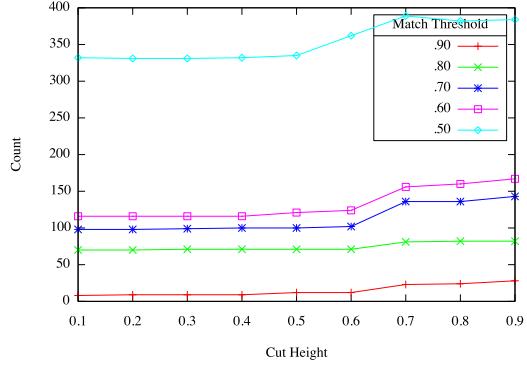


Figure 6: Effect of the dendrogram cut height (TPs).

prune CPTs related to parked C&C domains, as mentioned in Section 5.1.3. Once this initial set of CPTs was deployed, we continued to collect new malware traces from the feed, and updated the CPT set daily by adding new CPTs derived from the additional malware traces. More precisely, let D_1 be the day when the initial set of CPTs was first deployed in a live network, and let C_1 be this initial CPT set. C_1 is generated from the malware traces collected during a two-week period immediately before day D_1 . The CPTs set C_1 was then used to detect new C&C domains during the entire day D_1 . At the same time, during D_1 we generated additional CPTs from the malware traces collected on that day, and added them to set C_1 . Therefore, at the end of day D_1 we had an expanded set C_2 of CPTs, which we deployed on day D_2 , and so on. At the end of the deployment period we had just over 4,000 distinct CPTs.

To adapt the CPTs to the traffic of each deployment network (see Section 4.5), we proceeded in a similar way. We built a background traffic profile based on all HTTP traffic observed at each deployment network during the two days immediately before day D_1 , and used this profile to adapt the initial set of CPTs C_1 . Then, every day we updated the traffic profile statistics based on the new live traffic observed on that day, and used this information to further adapt all the CPTs. Notice that the set of CPTs deployed to different networks are different, in that they adapt differently to each deployment network (using that network’s background traffic profile).

5.3 Live Network Deployment Results

To evaluate ExecScent, we deployed it in three different large networks, UNETA, UNETB, and FNEN, for a period of two weeks. We generated the set of adaptive CPTs as explained above (Section 5.2.2), using a total of four weeks of malware-generated network traces (two weeks before deployment, plus daily updates during the

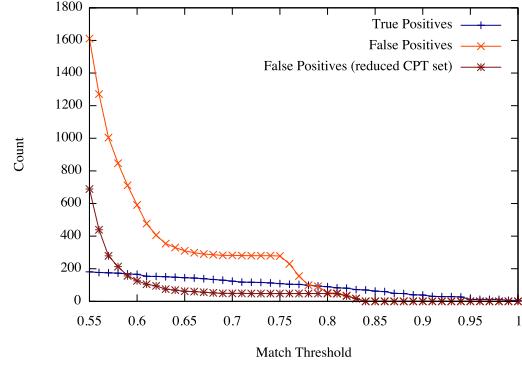


Figure 7: CPT detection results for varying detection thresholds.

two-week deployment period). The CPT matching engine was deployed at the edge of each network.

The detection phase proceeded as follows. For each network, we logged all HTTP requests that matched any of the adapted CPTs with a matching score $S \geq 0.5$, along with information such as the destination IP address of the request, the related domain name, the source IP address of the host that generated the request, and the actual value of the score S . This allowed us to compute the trade-off between the number of true and false positives for varying values of the detection threshold θ . Specifically, let h be a request whose matching score S_h is above the detection threshold θ , and let d be the domain name related to h . Consequently, we label h as a C&C request, and classify d as a C&C domain. We then rely on the CCBL and ATWL lists and on manual analysis (with the help of professional threat analysis) to confirm whether the detection of d represents a true positive, i.e., if d is in fact a C&C domain, or a false positive, in which case d is not a C&C domain.

Figure 7 summarizes the overall number of true positives and false positives obtained during the two-week deployment period over the three different live networks, while Table 2 shows a breakdown of the results on the different networks for a set of representative detection thresholds. For example, in Table 2, consider UNETA with a detection threshold of 0.65. During the two-week deployment period, we detected a total of 66 C&C domains, of which 34 are new, previously unknown C&C domains that were not present in our commercial black-list, CCBL. The 66 C&C domains were related to 17 distinct malware families. Overall, we detected 105 infected hosts, 90 of which were new infections related to the 34 previously unknown C&C domains. This means that 90 ($\approx 86\%$) of the infected hosts could not be detected by simply relying on the CCBL black-list.

The CPTs generated 118 false positives, namely domain names that we misclassified as C&C domains. We

Table 2: Live network results over a two-week deployment period

Detection Threshold	UNETA				UNETB				FNET			
	.62	.65	.73	.84	.62	.65	.73	.84	.62	.65	.73	.84
All C&C Domains	68	66	46	25	36	32	24	10	2	2	2	1
New C&C Domains	35	34	26	13	21	18	15	4	2	2	2	1
Distinct Malware Families	17	17	14	8	14	12	10	4	1	1	1	1
Number of Infected Hosts	114	105	98	37	185	150	147	21	7	7	7	7
Number of New Infected Hosts	91	90	86	25	145	135	133	11	7	7	7	7
FP Domains	133	118	114	0	152	117	105	0	109	63	49	0
FP Domains (reduced CPT set)	25	13	10	0	40	26	22	0	30	23	16	0

noticed that most of these false positives were generated by only two CPTs (the same two CPTs generated most false positives in all networks). By subtracting the false positives due to these two “noisy” CPTs, we were left with only 13 false positives, as shown in the last row of Table 2. The false positives marked with “*reduced CPT set*” in Figure 7 are also related to results without these two CPTs. Overall, within the entire two-week test period ExecScent generated a quite manageable number of false positives, in that a professional threat analyst could analyze and filter out the false C&C domains in a matter of hours.

Notice that the low number (only two) of new C&C domains found in the FNET network was expected. In fact, FNET is a very sensitive financial institution, where many layers of network security mechanisms are already in use to prevent malware infections. However, our findings confirm that even well guarded networks remain vulnerable.

5.3.1 Pushdo Downloader

It is worth clarifying that all results reported in Figure 7 and Table 2 have been obtained after discounting the domains detected through a single CPT that was causing hundreds of misclassifications. Through a manual investigation, we easily found that ExecScent had correctly learned this CPT, which actually models the HTTP-based C&C communications of a PUSHDO downloader variant [28]. This particular variant purposely replicates its C&C requests, and sends them to a large number of *decoy* benign domain names. The malware does this to try to hide the true C&C domain in plain sight, among a large set of benign domains. However, while this makes it somewhat harder to find the true C&C among hundreds or even thousands of benign domains (which requires some manual analysis effort), it makes it very easy to identify the fact that the source hosts of these requests, which matched our PUSHDO CPT, are infected with that specific malware variant.

We further discuss the implications of similar types of *noisy* or *misleading* malware behaviors in Section 6.

5.3.2 UNETB Deployment Results

The results we obtained for the UNETB deployment have been obtained in a slightly different way, compared to UNETA and FNET. Because of the higher volume of traffic in UNETB our proof-of-concept implementation of the CPT match engine could not easily keep pace with the traffic. This was due especially to the fact that our match engine software was sharing hardware resources with other production software that have to be given a much higher priority. A few weeks after conducting the experiments reported here, we implemented an optimized version (written in C, rather than Python) that is almost 8x faster; thus, it can easily keep up with the traffic on UNETB.

To compensate for the performance problems of our prototype implementation, during the two-week deployment period we only considered the traffic for every other day. That is, we only matched the CPTs over about seven days of traffic in UNETB, effectively cutting in half the traffic volume processed by ExecScent.

5.4 “Static” and URL-Only Models

In this section we compare the results of ExecScent’s *adaptive templates*, to “static” (i.e., non-adaptive) templates, which only learn from malware-generated traces and do not take into account the traffic profile of the deployment network, and to URL-based C&C request models, which only use information extracted from URLs.

To obtain the “static” models, we simply took ExecScent’s CPTs and “turned off” the specificity parameters. In other words, we set the specificity scores in Equation 1 to zero (with the exception of σ_d , which is set to one), essentially turning the dynamic waits w_k into their static counterparts \hat{w}_k (see Section 4.7). In the following, we refer to these static (non-adaptive) templates as “Specificity-Off” models.

To obtain the URL-based models, again we “turn-off” the specificity information, and also ignore all components of ExecScent’s CPT apart from URL-related components. Effectively, in Equation 1 we only use the similarity functions s_1 , s_{2a} , and s_{2b} defined in Section 4.7. We refer to these templates as “URL-Only” models.

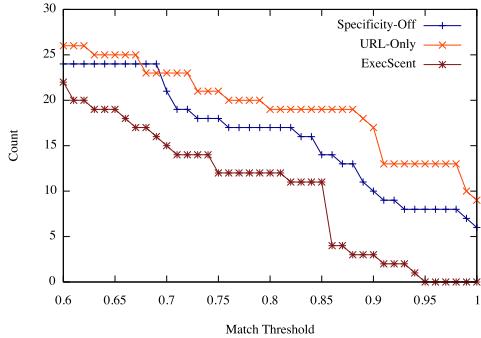


Figure 8: Comparing C&C Models - True Positives

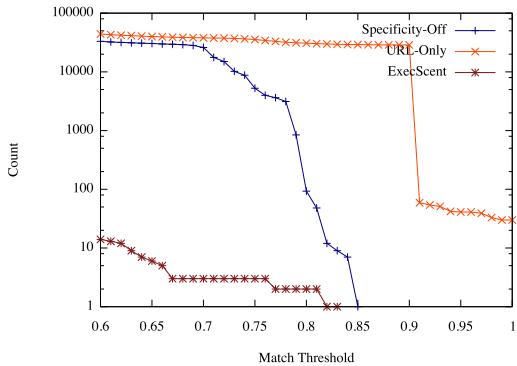


Figure 9: Comparing C&C Models - False Positives

To perform a comparison, we deployed the ExecScent CPTs and their related “Specificity-Off” and “URL-Only” models to UNETA, UNETB, and FNET for a period of 4 days. Figure 8 and 9 summarize the overall true and false positives, respectively, obtained by varying the detection threshold $\theta \in [0.6, 1]$. As can be seen from the figures, ExecScent’s adaptive templates outperform the two alternative models, for detection thresholds $\theta < 0.85$. Unless we are willing to sacrifice a large fraction of all true positives, compared to the numbers obtained at $\theta = 0.6$, the “Specificity-Off” and “URL-Only” models will generate a very large, likely unsustainable, number of false positives (notice the log scale on the y axes of Figure 9).

5.5 Deployment in ISP Networks

We were also able to evaluate the results of ExecScent over six large ISP networks serving several million hosts. We proceeded as follows: given 65 new C&C domains discovered by ExecScent during the live network deployment described in Section 5.3, we deployed the domains to the six ISPs for an entire week, during which we monitored all DNS traffic. Each day, we counted the number

of distinct source IP addresses that queried any of the 65 C&C domains. We found a maximum of 25,584 of distinct source IPs that in any given day queried these C&C domains. In other words, the new C&C domains discovered by ExecScent allowed us to identify 25,584 new potential malware infections across the six ISP networks.

6 Limitations

An attacker who gains knowledge of how ExecScent works may try to avoid detection by mutating her botnet’s C&C protocol every time the C&C server is relocated to a new domain. One possible approach would be to implement a new protocol that can be deployed on all the clients (i.e., malware agents) and servers (i.e., malware controllers) before switching to the new domain. However, this would substantially increase the complexity of managing the botnet and hurt its *agility*. Furthermore, for moderate to large botnets the updates would take time to deploy and a mistake in the update procedure could result in losing parts of or the entire botnet.

Another evasion approach may consist in injecting noise into the C&C protocol to make it appear “different”. For example, an attacker may randomly generate the C&C URL path or name-value pairs in the query-string, when making a request. However, if a malware agent needs to convey enough information to (loosely) authenticate itself to the C&C server, then at least one request component must have some form of “structured” data. Since ExecScent measures similarity by protocol structure and gives more weight to the shared unique components, it is non-trivial for an attacker to avoid detection on all deployment networks. In fact, several malware families we detect during our evaluation of ExecScent use such types of techniques to try to avoid detection via regular expressions.

An attacker may also try to “mislead” the detector by injecting noise into the domain name matches. For instance, an attacker may send requests to many decoy benign domains using the same malware C&C requests sent to the true C&C server. This is the approach used by the PUSHDO malware variant we discovered during our evaluation. This type of noisy malware is actually easy to identify, because of the number of unique destination domains contacted by a single host that match one particular CPT within a short period of time. Thus, detecting the infected hosts is easy. However, this makes it somewhat more difficult to determine the true C&C domains among all other domains. In this case, a threat analyst must review the domains, before they can be added to a blacklist; but at the same time, a security administrator can be immediately alerted regarding the infected hosts, thus enabling a prompt remediation.

Blending into the background traffic is another technique that may be used to avoid detection. For example, an attacker may choose “common” data types and values for their C&C protocol components. For some components such as the URL path it may be easy to select a popular value (e.g., “index.html”). However for many of the components, the “commonality” is relative to the deployment network’s traffic profile. Therefore, an attacker would need to customize the protocol based on the infected machine’s network. This may be difficult to do, because most network hosts have limited or no visibility into the traffic produced by other hosts in the same network. Therefore, although a C&C protocol may carry some “common” components, ExecScent’s adaptive CPTs may still be able to use those components that are *specific* (i.e., non-popular) in the deployment network to detect the C&C requests.

Finally, ExecScent’s CPTs depends on the malware traces and labeled C&C requests from which they are derived. Thus, ExecScent requires at least one or a few malware samples from a malware family, before its C&C protocol can be modeled and detected. In this case, though, malware code reuse plays to our advantage. A few samples of a malware family whose code has been reused elsewhere (because it was sold or leaked) will in fact facilitate the detection of future malware strains. Note that ExecScent in principle requires only a single sample to generate a CPT, thanks in particular to the request generalization process (Section 4.2). That being said, the quality of a CPT can be significantly improved when more than one sample sharing the same C&C protocol are available.

7 Related Work

Malware Clustering and Signature Generation: Grouping malware based on features extracted from HTTP requests has been studied for example in [7, 22, 23, 25]. Specifically, Perdisci et al. [22, 23] proposed a system for clustering malware samples that request similar sets of URLs. In addition, token-subsequences are extracted from the URLs, and used to detect infected hosts on live networks. In [7], information about HTTP request methods and URL parameters are used to cluster similar malware samples. The authors describe their clustering technique as a manual process and mention replacing it with an automated system in the future.

A recently proposed system FRIMA [25] clusters malware samples into families based on protocol features (e.g., same URL path) and for each family creates a set of network signatures. The network signatures are token-sets created from byte strings that are common to a large percentage of the network traffic within a clus-

ter. To reduce false positives, network signatures are pruned by removing the ones that match any communication in the authors’ benign traffic pool. Automated network signature generation has also been studied for detecting worms [19, 21, 27]. The generated signatures typically consist of fixed strings or token subsequences that can be deployed in an intrusion detection system. AutoRE [30] extends the automated signature generation process to produce regular expressions that can be used to match URLs in emails for the purpose of detecting spam emails and group them into spam campaigns.

Our work focuses on automatic *template* generation for detecting C&C communications and attributing them to a known malware family. In particular, our main focus is not on clustering malware samples per se. Rather, we apply clustering techniques mainly as an optimization step to generate high quality control protocol templates. Furthermore, we do not limit ourselves to only considering URLs or to extracting sets of common tokens. More importantly, our C&C templates are *adaptive*, in that they learn from the traffic of the network where they are to be deployed, thus self-tuning and automatically yielding a better trade-off between true and false positives.

Botnet Detection and C&C Identification: A number of studies have addressed the problem of detecting botnet traffic, for example [15, 16, 29]. BotSniffer [16] and BotMiner [15] are anomaly-based botnet detection systems that look for similar network behavior across hosts. The idea is that hosts infected with the same bot malware have common C&C communication patterns. Furthermore, BotMiner [15] leverages the fact that bots respond to commands in a coordinated way, producing similar malicious network activities. This type of systems require multiple infected hosts on the same monitored network for detection. In addition, being anomaly-based, they are not capable of attributing the infections to a specific malware family, and tend to suffer from relatively high false positive rates.

Our work is different, because ExecScent can detect botnets’ C&C even when only one bot is present in the monitored network. Furthermore, unlike previous work, ExecScent uses a *hybrid* detection approach, learning from both known C&C communications and the deployment network’s traffic to generated *adaptive* templates that can detect new C&C domains with high true positives and low false positives.

Wurzinger et al. [29] propose to isolate C&C traffic from mixed malicious and legitimated traffic generated by executing malware samples in a controlled environment. They propose to first identify malicious network activities (e.g., scanning, spamming, etc.), and then analyze the network traffic going back in time until a network flow is found that is likely to represent the command sent to the malware that caused the previously

identified malicious activities to be initiated. However, finding commands in malware network traces is not always possible. In fact, most datasets of malware network traces are obtained by running thousands of malware samples, with only a few minutes of execution time allocated to each sample. Therefore, the chances of witnessing a valid command being sent to a sample within such a small amount of time is intuitively small. On the other hand, malware samples typically attempt to contact the C&C server as soon as they run, even though no command to perform malicious activities may be issued at first contact. For this reason, ExecScent does not focus on identifying malicious network activities performed by the malware, and the related commands. Rather, ExecScent leverages any type of (HTTP-based) communication with a C&C server to learn control protocol templates that can be later used to identify new C&C communications and related C&C domains, even when malicious activities are not directly observable.

Jackstraws [17], executes malware in an instrumented sandbox [13] to generate behavior graphs of the system calls related to network communications. These system-level behavior graphs are then compared to C&C graph templates to find new C&C communications. ExecScent is different because it relies only on network information, and does not require malware to be executed in an instrumented sandbox (e.g., it can use traces collected from “bare metal” execution or live networks) to learn the templates. Furthermore, unlike Jackstraws [17], ExecScent learns *adaptive* templates, which allow us to identify new C&C domains in live networks.

Malicious Domains: Recently, a number of approaches for identifying malicious domains by monitoring DNS traffic have been proposed [2–4, 6]. These systems classify domains as malicious or benign, but do not attribute them to a specific malware family. Also, [2, 6] are mainly domain reputation system, and may assign a low reputation score to generic malicious domains, not only C&C domains, without providing any explicit distinction. On the other hand, [4] focuses only on malware that use pseudo-random domain generation algorithms. Kopis [3] is the only system that focuses explicitly on generic malware domains, but it requires the ability to monitor DNS traffic at the upper DNS hierarchy, which is difficult to obtain.

Unlike the DNS-based systems mentioned above, ExecScent focuses on detecting new C&C domains in live enterprise networks by inspecting HTTP(S) traffic, and using adaptive C&C protocol templates.

8 Conclusion

We presented ExecScent, a novel system that can discover new C&C domain names in *live* enterprise network traffic. ExecScent learns *adaptive* control protocol templates (CPTs) from both examples of known C&C communications and the “background traffic” of the network where the templates are to be deployed, yielding a better trade-off between true and false positives for a given network environment.

We deployed a prototype version of ExecScent in three large networks for a period of two weeks, discovering many new C&C domains and hundreds of new infected machines, compared to using a large up-to-date commercial C&C domain blacklist. We also compared ExecScent’s adaptive templates to “static” (non-adaptive) C&C traffic models. Our results show that ExecScent outperforms models that do not take the deployment network’s traffic into account. Furthermore, we deployed the new C&C domains we discovered using ExecScent to six large ISP networks, finding over 25,000 new malware-infected machines.

Acknowledgments

We thank the anonymous reviewers for their helpful comments. This material is based in part upon work supported by the National Science Foundation under Grant No. CNS-1149051. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References

- [1] ANTONAKAKIS, M., DEMAR, J., STEVENS, K., AND DAGON, D. Unveiling the network criminal infrastructure of tdds/tdl4. https://www.damballa.com/downloads/r_pubs/Damballa_tdss_tdl4_case_study_public.pdf.
- [2] ANTONAKAKIS, M., PERDISCI, R., DAGON, D., LEE, W., AND FEAMSTER, N. Building a dynamic reputation system for dns. In *Proceedings of the 19th USENIX conference on Security* (Berkeley, CA, USA, 2010), USENIX Security’10, USENIX Association, pp. 18–18.
- [3] ANTONAKAKIS, M., PERDISCI, R., LEE, W., VASILOGLOU, II, N., AND DAGON, D. Detecting malware domains at the upper dns hierarchy. In *Proceedings of the 20th USENIX conference on Security* (Berkeley, CA, USA, 2011), SEC’11, USENIX Association, pp. 27–27.
- [4] ANTONAKAKIS, M., PERDISCI, R., NADJI, Y., VASILOGLOU, N., ABU-NIMEH, S., LEE, W., AND DAGON, D. From throw-away traffic to bots: detecting the rise of dga-based malware. In *Proceedings of the 21st USENIX conference on Security symposium* (Berkeley, CA, USA, 2012), Security’12, USENIX Association, pp. 24–24.
- [5] BERNERS-LEE, T., FIELDING, R., AND MASINTER, L. RFC3986 - Uniform Resource Identifier (URI): Generic Syntax, 2005.

- [6] BILGE, L., KIRDA, E., KRUEGEL, C., AND BALDUZZI, M. Exposure: Finding malicious domains using passive dns analysis. In *NDSS* (2011), The Internet Society.
- [7] CABALLERO, J., GRIER, C., KREIBICH, C., AND PAXSON, V. Measuring pay-per-install: the commoditization of malware distribution. In *Proceedings of the 20th USENIX conference on Security* (Berkeley, CA, USA, 2011), SEC'11, USENIX Association, pp. 13–13.
- [8] CHANG, C.-C., AND LIN, C.-J. LIBSVM: A library for support vector machines. *ACM Transactions on Intelligent Systems and Technology* 2 (2011), 27:1–27:27. Software available at <http://www.csie.ntu.edu.tw/~cjlin/libsvm>.
- [9] CRISTIANINI, N., AND SHAWE-TAYLOR, J. *An introduction to support Vector Machines: and other kernel-based learning methods*. Cambridge University Press, New York, NY, USA, 2000.
- [10] DANCHEV, D. Leaked DIY malware generating tool spotted in the wild, 2013. <http://blog.webroot.com/2013/01/18/leaked-diy-malware-generating-tool-spotted-in-the-wild/>.
- [11] DE LA HIGUERA, C., AND CASACUBERTA, F. Topology of strings: Median string is np-complete. *Theoretical computer science* 230, 1 (2000), 39–48.
- [12] EDMONDS, R. ISC Passive DNS Architecture, 2012. <https://kb.isc.org/getAttach/30/AA-00654/passive-dns-architecture.pdf>.
- [13] EGELE, M., SCHOLTE, T., KIRDA, E., AND KRUEGEL, C. A survey on automated dynamic malware-analysis techniques and tools. *ACM Comput. Surv.* 44, 2 (Mar. 2008), 6:1–6:42.
- [14] FISHER, D. Zeus source code leaked. http://threatpost.com/en_us/blogs/zeus-source-code-leaked-051011.
- [15] GU, G., PERDISCI, R., ZHANG, J., AND LEE, W. Botminer: clustering analysis of network traffic for protocol- and structure-independent botnet detection. In *Proceedings of the 17th conference on Security symposium* (Berkeley, CA, USA, 2008), SS'08, USENIX Association, pp. 139–154.
- [16] GU, G., ZHANG, J., AND LEE, W. BotSniffer: Detecting botnet command and control channels in network traffic. In *Proceedings of the 15th Annual Network and Distributed System Security Symposium (NDSS'08)* (February 2008).
- [17] JACOB, G., HUND, R., KRUEGEL, C., AND HOLZ, T. Jackstraws: picking command and control connections from bot traffic. In *Proceedings of the 20th USENIX conference on Security* (Berkeley, CA, USA, 2011), SEC'11, USENIX Association, pp. 29–29.
- [18] JANG, J., BRUMLEY, D., AND VENKATARAMAN, S. Bitshred: feature hashing malware for scalable triage and semantic analysis. In *Proceedings of the 18th ACM conference on Computer and communications security* (New York, NY, USA, 2011), CCS '11, ACM, pp. 309–320.
- [19] KIM, H.-A., AND KARP, B. Autograph: toward automated, distributed worm signature detection. In *Proceedings of the 13th conference on USENIX Security Symposium - Volume 13* (Berkeley, CA, USA, 2004), SSYM'04, USENIX Association, pp. 19–19.
- [20] KOLTER, J. Z., AND MALOOF, M. A. Learning to detect and classify malicious executables in the wild. *J. Mach. Learn. Res.* 7 (Dec. 2006), 2721–2744.
- [21] NEWSOME, J., KARP, B., AND SONG, D. Polygraph: Automatically generating signatures for polymorphic worms. In *Proceedings of the 2005 IEEE Symposium on Security and Privacy* (Washington, DC, USA, 2005), SP '05, IEEE Computer Society, pp. 226–241.
- [22] PERDISCI, R., ARIU, D., AND GIACINTO, G. Scalable fine-grained behavioral clustering of http-based malware. *Computer Networks* 57, 2 (2013), 487 – 500. Botnet Activity: Analysis, Detection and Shutdown.
- [23] PERDISCI, R., LEE, W., AND FEAMSTER, N. Behavioral clustering of http-based malware and signature generation using malicious network traces. In *Proceedings of the 7th USENIX conference on Networked systems design and implementation* (Berkeley, CA, USA, 2010), NSDI'10, USENIX Association, pp. 26–26.
- [24] PLATT, J. Probabilistic outputs for support vector machines and comparisons to regularized likelihood methods. *Advances in large margin classifiers* 10, 3 (1999), 61–74.
- [25] RAFIQUE, M. Z., AND CABALLERO, J. Firma: Malware clustering and network signature generation with mixed network behaviors. In *Proceedings of the 16th international conference on Research in Attacks, Intrusions, and Defenses* (2013), RAID'13, Springer-Verlag. To be published. Research conducted concurrently and independently of ExecScent.
- [26] SANTORELLI, S. Developing botnets - an analysis of recent activity, 2010. <http://www.team-cymru.com/ReadingRoom/Whitepapers/2010/developing-botnets.pdf>.
- [27] SINGH, S., ESTAN, C., VARGHESE, G., AND SAVAGE, S. Automated worm fingerprinting. In *Proceedings of the 6th conference on Symposium on Operating Systems Design & Implementation - Volume 6* (Berkeley, CA, USA, 2004), OSDI'04, USENIX Association, pp. 4–4.
- [28] STONE-GROSS, B. Pushdo downloader variant generating fake HTTP requests, 2012. http://www.secureworks.com/cyber-threat-intelligence/threats/Pushdo_Downloader_Variant_Generating_Fake_HTTP_Requests/.
- [29] WURZINGER, P., BILGE, L., HOLZ, T., GOEBEL, J., KRUEGEL, C., AND KIRDA, E. Automatically generating models for botnet detection. In *Proceedings of the 14th European conference on Research in computer security* (Berlin, Heidelberg, 2009), ESORICS'09, Springer-Verlag, pp. 232–249.
- [30] XIE, Y., YU, F., ACHAN, K., PANIGRAHY, R., HULTEN, G., AND OSIPKOV, I. Spamming botnets: signatures and characteristics. In *Proceedings of the ACM SIGCOMM 2008 conference on Data communication* (New York, NY, USA, 2008), SIGCOMM '08, ACM, pp. 171–182.
- [31] ZHOU, Y., AND JIANG, X. Dissecting android malware: Characterization and evolution. In *Security and Privacy (SP), 2012 IEEE Symposium on* (2012), IEEE, pp. 95–109.

ZMap: Fast Internet-Wide Scanning and its Security Applications

Zakir Durumeric

University of Michigan

zakir@umich.edu

Eric Wustrow

University of Michigan

ewust@umich.edu

J. Alex Halderman

University of Michigan

jhalderm@umich.edu

Abstract

Internet-wide network scanning has numerous security applications, including exposing new vulnerabilities and tracking the adoption of defensive mechanisms, but probing the entire public address space with existing tools is both difficult and slow. We introduce ZMap, a modular, open-source network scanner specifically architected to perform Internet-wide scans and capable of surveying the entire IPv4 address space in under 45 minutes from user space on a single machine, approaching the theoretical maximum speed of gigabit Ethernet. We present the scanner architecture, experimentally characterize its performance and accuracy, and explore the security implications of high speed Internet-scale network surveys, both offensive and defensive. We also discuss best practices for good Internet citizenship when performing Internet-wide surveys, informed by our own experiences conducting a long-term research survey over the past year.

1 Introduction and Roadmap

Internet-scale network surveys collect data by probing large subsets of the public IP address space. While such scanning behavior is often associated with botnets and worms, it also has proved to be a valuable methodology for security research. Recent studies have demonstrated that Internet-wide scanning can help reveal new kinds of vulnerabilities, monitor deployment of mitigations, and shed light on previously opaque distributed ecosystems [10, 12, 14, 15, 25, 27]. Unfortunately, this methodology has been more accessible to attackers than to legitimate researchers, who cannot employ stolen network access or spread self-replicating code. Comprehensively scanning the public address space with off-the-shelf tools like Nmap [23] requires weeks of time or many machines.

In this paper, we introduce ZMap, a modular and open-source network scanner specifically designed for performing comprehensive Internet-wide research scans. A single

mid-range machine running ZMap is capable of scanning for a given open port across the entire public IPv4 address space in under 45 minutes—over 97% of the theoretical maximum speed of gigabit Ethernet—without requiring specialized hardware [11] or kernel modules [8, 28]. ZMap’s modular architecture can support many types of single-packet probes, including TCP SYN scans, ICMP echo request scans, and application-specific UDP scans, and it can interface easily with user-provided code to perform follow-up actions on discovered hosts, such as completing a protocol handshake.

Compared to Nmap—an excellent general-purpose network mapping tool, which was utilized in recent Internet-wide survey research [10, 14]—ZMap achieves much higher performance for Internet-scale scans. Experimentally, we find that ZMap is capable of scanning the IPv4 public address space over 1300 times faster than the most aggressive Nmap default settings, with equivalent accuracy. These performance gains are due to architectural choices that are specifically optimized for this application:

Optimized probing While Nmap adapts its transmission rate to avoid saturating the source or target networks, we assume that the source network is well provisioned (unable to be saturated by the source host), and that the targets are randomly ordered and widely dispersed (so no distant network or path is likely to be saturated by the scan). Consequently, we attempt to send probes as quickly as the source’s NIC can support, skipping the TCP/IP stack and generating Ethernet frames directly. We show that ZMap can send probes at gigabit line speed from commodity hardware and entirely in user space.

No per-connection state While Nmap maintains state for each connection to track which hosts have been scanned and to handle timeouts and retransmissions, ZMap forgoes any per-connection state. Since it is intended to target random samples of the address space, ZMap can avoid storing the addresses it has already scanned or needs to scan and instead selects addresses according to a random permutation generated by a cyclic

multiplicative group. Rather than tracking connection timeouts, ZMap accepts response packets with the correct state fields for the duration of the scan, allowing it to extract as much data as possible from the responses it receives. To distinguish valid probe responses from background traffic, ZMap overloads unused values in each sent packet, in a manner similar to SYN cookies [4].

No retransmission While Nmap detects connection timeouts and adaptively retransmits probes that are lost due to packet loss, ZMap (to avoid keeping state) always sends a fixed number of probes per target and defaults to sending only one. In our experimental setup, we estimate that ZMap achieves 98% network coverage using only a single probe per host, even at its maximum scanning speed. We believe this small amount of loss will be insignificant for typical research applications.

We further describe ZMap’s architecture and implementation in Section 2, and we experimentally characterize its performance in Section 3. In Section 4, we investigate the implications of the widespread availability of fast, low-cost Internet-wide scanning for both defenders and attackers, and we demonstrate ZMap’s performance and flexibility in a variety of security settings, including:

Measuring protocol adoption, such as the transition from HTTP to HTTPS. We explore HTTPS adoption based on frequent Internet-wide scans over a year.

Visibility into distributed systems, such as the certificate authority (CA) ecosystem. We collect and analyze TLS certificates and identify misissued CA certs.

High-speed vulnerability scanning, which could allow attackers to widely exploit vulnerabilities within hours of their discovery. We build a UPnP scanner using ZMap through which we find 3.4 million UPnP devices with known vulnerabilities [25].

Uncovering unadvertised services, such as hidden Tor bridges. We show that ZMap can locate 86% of hidden Tor bridges via comprehensive enumeration.

High-speed scanning can be a powerful tool in the hands of security researchers, but users must be careful not to cause harm by inadvertently overloading networks or causing unnecessary work for network administrators. In Section 5, we discuss our experiences performing numerous large-scale scans over the past year, we report on the complaints and other reactions we have received, and we suggest several guidelines and best practices for good Internet citizenship while scanning.

Internet-wide scanning has already shown great potential as a research methodology [10, 12, 14, 25], and we hope ZMap will facilitate a variety of new applications by drastically reducing the costs of comprehensive network surveys and allowing scans to be performed with very fine time granularity. To facilitate this, we are releasing ZMap as an open source project that is documented and packaged for real world use. It is available at <https://zmap.io/>.

2 ZMap: The Scanner

ZMap uses a modular design to support many types of probes and integration with a variety of research applications, as illustrated in Figure 1. The *scanner core* handles command line and configuration file parsing, address generation and exclusion, progress and performance monitoring, and reading and writing network packets. Extensible *probe modules* can be customized for different kinds of probes, and are responsible for generating probe packets and interpreting whether incoming packets are valid responses. Modular *output handlers* allow scan results to be piped to another process, added directly to a database, or passed on to user code for further action, such as completing a protocol handshake.

We introduced the philosophy behind ZMap’s design in Section 1. At a high level, one of ZMap’s most important architectural features is that sending and receiving packets take place in separate threads that act independently and continuously throughout the scan. A number of design choices were made to ensure that these processes share as little state as possible.

We implemented ZMap in approximately 8,900 SLOC of C. It was written and tested on GNU/Linux.

2.1 Addressing Probes

If ZMap simply probed every IPv4 address in numerical order, it would risk overloading destination networks with scan traffic and produce inconsistent results in the case of a distant transient network failure. To avoid this, ZMap scans addresses according to a random permutation of the address space. To select smaller random samples of the address space, we simply scan a subset of the full permutation.

ZMap uses a simple and inexpensive method to traverse the address space, which lets it scan in a random permutation while maintaining only negligible state. We iterate over a multiplicative group of integers modulo p , choosing p to be a prime slightly larger than 2^{32} . By choosing p to be a prime, we guarantee that the group is cyclic and will reach all addresses in the IPv4 address space except 0.0.0.0 (conveniently an IANA reserved address) once per cycle. We choose to iterate over $(\mathbb{Z}/4,294,967,311\mathbb{Z})^\times$, the multiplicative group modulo p for the smallest prime larger than 2^{32} : $2^{32} + 15$.

To select a fresh random permutation for each scan, we generate a new primitive root of the multiplicative group and choose a random starting address. Because the order of elements in a group is preserved by an isomorphism, we efficiently find random primitive roots of the multiplicative group by utilizing the isomorphism $(\mathbb{Z}_{p-1}, +) \cong (\mathbb{Z}_p^*, \times)$ and mapping roots of $(\mathbb{Z}_{p-1}, +)$ into the multiplicative group via the function $f(x) = n^x$ where n is a known primitive root of $(\mathbb{Z}/p\mathbb{Z})^\times$. In our

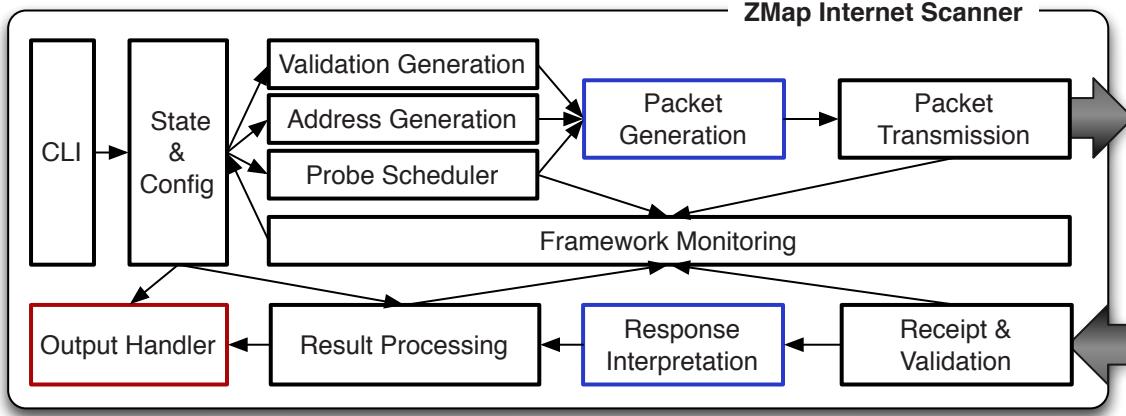


Figure 1: **ZMap Architecture**—ZMap is an open-source network scanner optimized for efficiently performing Internet-scale network surveys. Modular packet generation and response interpretation components (blue) support multiple kinds of probes, including TCP SYN scans and ICMP echo scans. Modular output handlers (red) allow users to output or act on scan results in application-specific ways. The architecture allows sending and receiving components to run asynchronously and enables a single source machine to comprehensively scan every host in the public IPv4 address space for a particular open TCP port in under 45 mins using a 1 Gbps Ethernet link.

specific case, we know that 3 is a primitive root of $(\mathbb{Z}/4,294,967,311\mathbb{Z})^\times$.

Because we know that the generators of $(\mathbb{Z}_{p-1}, +)$ are $\{s | (s, p - 1) = 1\}$, we can efficiently find the generators of the additive group by precalculating and storing the factorization of $p - 1$ and checking addresses against the factorization at random until we find one that is coprime with $p - 1$ and then map it into (\mathbb{Z}_p^*, \times) . Given that there exist approximately 10^9 generators, we expect to make four tries before finding a primitive root. While this process introduces complexity at the beginning of a scan, it adds only a small amount of one-time overhead.

Once a primitive root has been generated, we can easily iterate through the address space by applying the group operation to the current address (in other words, multiplying the current address by the primitive root modulo $2^{32} + 15$). We detect that a scan has completed when we reach the initially scanned IP address. This technique allows the sending thread to store the selected permutation and progress through it with only three integers: the primitive root used to generate the multiplicative group, the first scanned address, and the current address.

Excluding Addresses Since ZMap is optimized for Internet-wide scans, we represent the set of targets as the full IPv4 address space minus a set of smaller excluded address ranges. Certain address ranges need to be excluded for performance reasons (e.g., skipping IANA reserved allocations [16]) and others to honor requests from their owners to discontinue scanning. We efficiently support address exclusion through the use of radix trees, a trie specifically designed to handle ranges and frequently

used by routing tables [32, 34]. Excluded ranges can be specified through a configuration file.

2.2 Packet Transmission and Receipt

ZMap is optimized to send probes as quickly as the source’s CPU and NIC can support. The packet generation component operates asynchronously across multiple threads, each of which maintains a tight loop that sends Ethernet-layer packets via a raw socket.

We send packets at the Ethernet layer in order to cache packet values and reduce unnecessary kernel overhead. For example, the Ethernet header, minus the packet checksum, will never change during a scan. By generating and caching the Ethernet layer packet, we prevent the Linux kernel from performing a routing lookup, an arpcache lookup, and netfilter checks for every packet. An additional benefit of utilizing a raw socket for TCP SYN scans is that, because no TCP session is established in the kernel, upon receipt of a TCP SYN-ACK packet, the kernel will automatically respond with a TCP RST packet, closing the connection. ZMap can optionally use multiple source addresses and distribute outgoing probes among them in a round-robin fashion.

We implement the receiving component of ZMap using libpcap [17], a library for capturing network traffic and filtering the received packets. Although libpcap is a potential performance bottleneck, incoming response traffic is a small fraction of outgoing probe traffic, since the overwhelming majority of hosts are unresponsive to typical probes, and we find that libpcap is easily capable of handling response traffic in our tests (see Section 3).

Upon receipt of a packet, we check the source and destination port, discard packets clearly not initiated by the scan, and pass the remaining packets to the active probe module for interpretation.

While the sending and receiving components of ZMap operate independently, we ensure that the receiver is initialized prior to sending probes and that the receiver continues to run for a period of time (by default, 8 seconds) after the sender has completed in order to process any delayed responses.

2.3 Probe Modules

ZMap probe modules are responsible for filling in the body of probe packets and for validating whether incoming packets are responsive to the probes. Making these tasks modular allows ZMap to support a variety of probing methods and protocols and simplifies extensibility. Out of the box, ZMap provides probe modules to support TCP port scanning and ICMP echo scanning.

At initialization, the scanner core provides an empty buffer for the packet and the probe module fills in any static content that will be the same for all targets. Then, for each host to be scanned, the probe module updates this buffer with host-specific values. The probe module also receives incoming packets, after high-level validation by the scanner core, and determines whether they are positive or negative responses to scan probes. Users can add new scan types by implementing a small number of callback functions within the probe module framework.

For example, to facilitate TCP port scanning, ZMap implements a probing technique known as *SYN scanning* or *half-open scanning*. We chose to implement this specific technique instead of performing a full TCP handshake based on the reduced number of exchanged packets. In the dominant case where a host is unreachable or does not respond, only a single packet is used (a SYN from the scanner); in the case of a closed port, two packets are exchanged (a SYN answered with a RST); and in the uncommon case where the port is open, three packets are exchanged (a SYN, a SYN-ACK reply, and a RST from the scanner).

Checking Response Integrity ZMap’s receiving components need to determine whether received packets are valid responses to probes originating from the scanner or are part of other background traffic. Probe modules perform this validation by encoding host- and scan-invocation-specific data into mutable fields of each probe packet, utilizing fields that will have recognizable effects on fields of the corresponding response packets in a manner similar to SYN cookies [4].

For each scanned host, ZMap computes a MAC of the destination address keyed by a scan-specific secret. This MAC value is then spread across any available fields by

the active probe module. We chose to use the UMAC function for these operations, based on its performance guarantees [5]. In our TCP port scan module, we utilize the source port and initial sequence number; for ICMP, we use the ICMP identifier and sequence number. These fields are checked on packet receipt by the probe module, and ZMap discards any packets for which validation fails.

These inexpensive checks prevent the incorrect reporting of spurious response packets due to background traffic as well as responses triggered by previous scans. This design ultimately allows the receiver to validate responses while sharing only the scan secret and the initial configuration with the sending components.

2.4 Output Modules

ZMap provides a modular output interface that allows users to output scan results or act on them in application-specific ways. Output module callbacks are triggered by specific events: scan initialization, probe packet sent, response received, regular progress updates, and scan termination. ZMap’s built-in output modules cover basic use, including simple text output (a file stream containing a list of unique IP addresses that have the specified port open), extended text output (a file stream containing a list of all packet responses and timing data), and an interface for queuing scan results in a Redis in-memory database [29].

Output modules can also be implemented to trigger network events in response to positive scan results, such as completing an application-level handshake. For TCP SYN scans, the simplest way to accomplish this is to create a fresh TCP connection with the responding address; this can be performed asynchronously with the scan and requires no special kernel support.

forge_socket Some ZMap users may wish to complete the TCP handshake begun during a TCP SYN scan and exchange data with the remote host without the extra overhead of establishing a new connection. While the initial SYN/SYN-ACK exchange has established a connection from the destination’s perspective, ZMap bypasses the local system’s TCP stack and as such the kernel does not recognize the connection.

In order to allow the scanning host to communicate over ZMap-initiated TCP sessions, we implemented `forge_socket`, a kernel module that allows user processes to pass in session parameters (e.g. initial sequence number) using `setsockopt`. This allows application-level handshakes to be performed using the initial ZMap handshake and does not require the unnecessary transmission of a RST, SYN, or SYN-ACK packet that would be required to close the existing connection and initiate a new kernel-recognized session. We are releasing `forge_socket` along with ZMap.

3 Validation and Measurement

We performed a series of experiments to characterize the performance of ZMap. Under our test setup, we find that a complete scan of the public IPv4 address space takes approximately 44 minutes on an entry-level server with a gigabit Ethernet connection. We estimate that a single-packet scan can detect approximately 98% of simultaneously listening hosts, and we measure a 1300 x performance improvement over Nmap for Internet-wide scanning, with equivalent coverage.

We performed the following measurements on an HP ProLiant DL120 G7 with a Xeon E3-1230 3.2 GHz processor and 4 GB of memory running a stock install of Ubuntu 12.04.1 LTS and the 3.2.0-32-generic Linux kernel. Experiments were conducted using the onboard NIC, which is based on the Intel 82574L chipset and uses the stock e1000e network driver, or a quad-port Intel Ethernet adapter based on the newer Intel 82580 chipset and using the stock igb network driver. For experiments involving complete TCP handshakes, we disabled kernel modules used by iptables and conntrack. Experiments comparing ZMap with Nmap were conducted with Nmap 5.21.

These measurements were conducted using the normal building network at the University of Michigan Computer Science & Engineering division. We used a gigabit Ethernet uplink (a standard office network connection in our facility); we did not arrange for any special network configuration beyond static IP addresses. The access layer of the building runs at 10 gbps, and the building uplink to the rest of the campus is an aggregated 2×10 gigabit port channel. We note that ZMap's performance on other source networks may be worse than reported here due to local congestion.

3.1 Scan Rate: How Fast is Too Fast?

In order to determine whether our scanner and our upstream network can handle scanning at gigabit line speed, we examine whether the *scan rate*, the rate at which ZMap sends probe packets, has any effect on the *hit rate*, the fraction of probed hosts that respond positively (in this case, with a SYN-ACK). If libpcap, the Linux kernel, our institutional network, or our upstream provider are unable to adequately handle the traffic generated by the scanner at full speed, we would expect packets to be dropped and the hit rate to be lower than at slower scan rates.

We experimented by sending TCP SYN packets to random 1% samples of the IPv4 address space on port 443 at varying scan rates. We conducted 10 trials at each of 16 scan rates ranging from 1,000 to 1.4 M packets per second. The results are shown in Figure 2.

We find no statistically significant correlation between scan rate and hit rate. This shows that our ZMap setup is capable of handling scanning at 1.4 M packets per

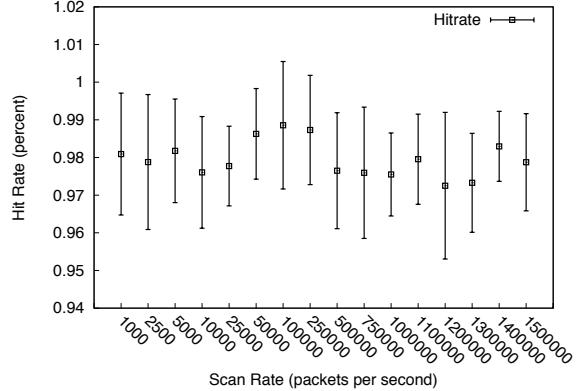


Figure 2: **Hit rate vs. Scan rate** — We find no correlation between hit rate (positive responses/hosts probed) and scan rate (probes sent/second). Shown are means and standard deviations over ten trials. This indicates that slower scanning does not reveal additional hosts.

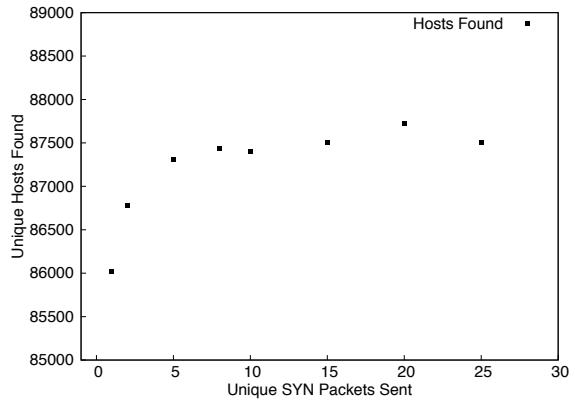


Figure 3: **Coverage for Multiple Probes** — Discovered hosts plateau after ZMap sends about 8 SYNs to each. If this plateau represents the true number of listening hosts, sending just 1 SYN will achieve about 98% coverage.

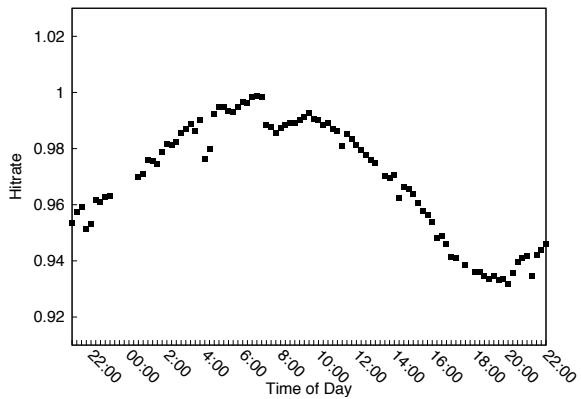


Figure 4: **Diurnal Effect on Hosts Found** — We observed a $\pm 3.1\%$ variation in ZMap's hit rate depending on the time of day the scan was performed. (Times EST.)

second and that scanning at lower rates provides no benefit in terms of identifying additional hosts. From an architectural perspective, this validates that our receiving infrastructure based on libpcap is capable of processing responses generated by the scanner at full speed and that kernel modules such as PF_RING [8] are not necessary for gigabit-speed network scanning.

3.2 Coverage: Is One SYN Enough?

While scanning at higher rates does not appear to result in a lower hit rate, this does not tell us what *coverage* we achieve with a single scan—what fraction of target hosts does ZMap actually find using its default single-packet probing strategy?

Given the absence of ground truth for the number of hosts on the Internet with a specific port open, we cannot measure coverage directly. This is further complicated by the ever changing state of the Internet; it is inherently difficult to detect whether a host was not included in a scan because it was not available at the time or because packets were dropped between it and the scanner. Yet, this question is essential to understanding whether performing fast, single-packet scans is an accurate methodology for Internet-wide surveys.

To characterize ZMap’s coverage, we estimate the number of hosts that are actually listening by sending multiple, distinct SYN packets to a large address sample and analyzing the distribution of the number of positive responses received compared to the number of SYNs we send. We expect to eventually see a plateau in the number of hosts that respond regardless of the number of additional SYNs we send. If this plateau exists, we can treat it as an estimate of the real number of listening hosts, and we can use it as a baseline against which to compare scans with fewer SYN packets.

We performed this experiment by sending 1, 2, 5, 8, 10, 15, 20, and 25 SYN packets to random 1% samples of the IPv4 address space on port 443 and recording the number of distinct hosts that sent SYN-ACK responses in each scan. The results indicate a clear plateau in the number of responsive hosts after sending 8 SYN packets, as shown in Figure 3.

Based on the level of this plateau, we estimate that our setup reaches approximately 97.9% of live hosts using a single packet, 98.8% of hosts using two packets, and 99.4% of hosts using three packets. The single packet round-trip loss rate of about 2% is in agreement with previous studies on random packet drop on the Internet [12].

These results suggest that single-probe scans are sufficiently comprehensive for typical research applications. Investigators who require higher coverage can configure ZMap to send multiple probes per host, at the cost of proportionally longer running scans.

3.3 Variation by Time of Day

In previous work, Internet-wide scans took days to months to execute, so there was little concern over finding the optimal time of day to perform a scan. However, since ZMap scans can take less than an hour to complete, the question as to the “right time” to perform a scan arises. Are there certain hours of the day or days of the week that are more effective for scanning than others?

In order to measure any diurnal effects on scanning, we performed continuous scans of TCP port 443 targeting a random 1% sample of the Internet over a 24-hour period. Figure 4 shows the number of hosts found in each scan.

We observed a $\pm 3.1\%$ variation in hit rate dependent on the time of day scans took place. The highest response rates were at approximately 7:00 AM EST and the lowest response rates were at around 7:45 PM EST.

These effects may be due to variation in overall network congestion and packet drop rates or due to a diurnal pattern in the aggregate availability of end hosts that are only intermittently connected to the network. In less formal testing, we did not notice any obvious variation by day of the week or day of the month.

3.4 Comparison with Nmap

We performed several experiments to compare ZMap to Nmap in Internet-wide scanning applications, focusing on coverage and elapsed time to complete a scan. Nmap and ZMap are optimized for very different purposes. Nmap is a highly flexible, multipurpose tool that is frequently used for probing a large number of open ports on a smaller number of hosts, whereas ZMap is optimized to probe a single port across very large numbers of targets. We chose to compare the two because recent security studies used Nmap for Internet-wide surveys [10, 14], and because, like ZMap, Nmap operates from within user space on Linux [23].

We tested a variety of Nmap settings to find reasonable configurations to compare. All performed a TCP SYN scan on port 443 (-Ss -p 443). Nmap provides several defaults known as *timing templates*, but even with the most aggressive of these (labeled “insane”), an Internet-wide scan would take over a year to complete. To make Nmap scan faster in our test configurations, we started with the “insane” template (-T5), disabled host discovery and DNS resolutions (-Pn -n), and set a high minimum packet rate (--min-rate 10000). The “insane” template retries each probe once after a timeout; we additionally tested a second Nmap configuration with retries disabled (--max-retries 0).

We used ZMap to select a random sample of 1 million IP addresses and scanned them for hosts listening on port 443 with Nmap in the two configurations described above and with ZMap in its default configuration and in a

Scan Type	Coverage (normalized)	Duration (mm:ss)	Est. Time for Internet-wide Scan
Nmap, max 2 probes (default)	0.978	45:03	116.3 days
Nmap, 1 probe	0.814	24:12	62.5 days
ZMap, 2 probes	1.000	00:11	2:12:35
ZMap, 1 probe (default)	0.987	00:10	1:09:45

Table 1: **ZMap vs. Nmap Comparison**— We scanned 1 million hosts on TCP port 443 using ZMap and Nmap and averaged over 10 trials. Despite running hundreds of times faster, ZMap finds more listening hosts than Nmap, due to Nmap’s low host timeout. Times for ZMap include a fixed 8 second delay to wait for responses after the final probe.

second configuration that sends two SYN probes to each host (-P 2). We repeated this process for 10 trials over a 12 hour period and report the averages in Table 1.

The results show that ZMap scanned much faster than Nmap and found more listening hosts than either Nmap configuration. The reported durations for ZMap include time sent sending probes as well as a fixed 8-second delay after the sending process completes, during which ZMap waits for late responses. Extrapolating to the time required for an Internet-wide scan, the fastest tested ZMap configuration would complete approximately 1300 times faster than the fastest Nmap configuration.¹

Coverage and Timeouts To investigate why ZMap achieved higher coverage than Nmap, we probed a random sample of 4.3 million addresses on TCP port 80 and measured the latency between sending a SYN and receiving a SYN-ACK from responsive hosts. Figure 5 shows the CDF of the results. The maximum round-trip time was 450 seconds, and a small number of hosts took more than 63 seconds to respond, the time it takes for a TCP

¹The extrapolated 1-packet Internet-wide scan time for ZMap is longer than the 44 minutes we report elsewhere for complete scans, because this test used a slower NIC based on the Intel 82574L chipset.

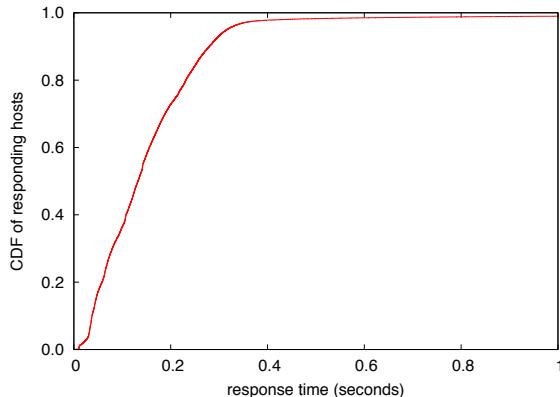


Figure 5: **SYN to SYN-ACK time**— In an experiment that probed 4.3 million hosts, 99% of SYN-ACKs arrived within about 1 second and 99.9% within 8.16 seconds.

connection attempt to timeout on Linux. 99% of hosts that responded within 500 seconds did so within about 1 second, and 99.9% responded within 8.16 seconds.

As ZMap’s receiving code is stateless with respect to the sending code, a valid SYN-ACK that comes back any time before the scan completes will be recorded as a listening host. To assure a high level of coverage, the default ZMap settings incorporate an empirically derived 8-second delay after the last probe is sent before the receiving process terminates.

In contrast, Nmap maintains timeouts for each probe. In the Nmap “insane” timing template we tested, the timeout is initially 250 ms, by which time fewer than 85% of responsive hosts in our test had responded. Over the course of a scan, Nmap’s timeout can increase to 300 ms, by which time 93.2% had responded. Thus, we would expect a single-probe Nmap scan with these timing values to see 85–93% of the hosts that ZMap finds, which is roughly in line with the observed value of 82.5%.

With Nmap’s “insane” defaults, it will attempt to send a second probe after a timeout. A response to either the first or second SYN will be considered valid until the second times out, so this effectively raises the overall timeout to 500–600 ms, by which time we received 98.2–98.5% of responses. Additional responses will likely be generated by the second SYN. We observed that the 2-probe Nmap scan found 99.1% of the number of hosts that a 1-probe ZMap scan found.

3.5 Comparison with Previous Studies

Several groups have previously performed Internet-wide surveys using various methodologies. Here we compare ZMap to two recent studies that focused on HTTPS certificates. Most recently, Heninger et al. performed a distributed scan of port 443 in 2011 as part of a global analysis on cryptographic key generation [14]. Their scan used Nmap on 25 Amazon EC2 instances and required 25 hours to complete, with a reported average of 40,566 hosts scanned per second. A 2010 scan by the EFF SSL Observatory project used Nmap on 3 hosts and took 3 months to complete [10].

Scan	Date	Port 443 Open	TLS Servers	All Certs	Trusted Certs
EFF SSL Observatory [10]	2010/12	16.2 M	7.7 M	4.0 M	1.46 M
Mining Ps and Qs [14]	2011/10	28.9 M	12.8 M	5.8 M	1.96 M
ZMap + certificate fetcher	2012/06	31.8 M	19.0 M	7.8 M	2.95 M
ZMap + certificate fetcher	2013/05	34.5 M	22.8 M	8.6 M	3.27 M

Table 2: **Comparison with Prior Internet-wide HTTPS Surveys**—Due to growth in HTTPS deployment, ZMap finds almost three times as many TLS servers as the SSL Observatory did in late 2010, yet this process takes only 10 hours to complete from a single machine using a ZMap-based workflow, versus three months on three machines.

To compare ZMap’s performance for this task, we used it to conduct comprehensive scans of port 443 and used a custom certificate fetcher based on libevent [24] and OpenSSL [37] to retrieve TLS certificates from each responsive host. With this methodology, we were able to discover hosts, perform TLS handshakes, and collect and parse the resulting certificates in under 10 hours from a single machine.

As shown in Table 2, we find significantly more TLS servers than previous work—78% more than Heninger et al. and 196% more than the SSL Observatory—likely due to increased HTTPS deployment since those studies were conducted. Linear regression shows an average growth in HTTPS deployment of about 540,000 hosts per month over the 29 month period between the SSL Observatory scan and our most recent dataset. Despite this growth, ZMap is able to collect comprehensive TLS certificate data in a fraction of the time and cost needed in earlier work. The SSL Observatory took roughly 650 times as much machine time to acquire the same kind of data, and Heninger et al. took about 65 times as much.

4 Applications and Security Implications

The ability to scan the IPv4 address space in under an hour opens an array of new research possibilities, including the ability to gain visibility into previously opaque distributed systems, understand protocol adoption at a new resolution, and uncover security phenomenon only accessible with a global perspective [14]. However, high-speed scanning also has potentially malicious applications, such as finding and attacking vulnerable hosts en masse. Furthermore, many developers have the preconceived notion that the Internet is far too large to be fully enumerated, so the reality of high speed scanning may disrupt existing security models, such as by leading to the discovery of services previously thought to be well hidden. In this section, we use ZMap to explore several of these applications.

4.1 Visibility into Distributed Systems

High-speed network scanning provides researchers with the possibility for a new real-time perspective into pre-

Organization	Certificates
GoDaddy.com, Inc.	913,416 (31.0%)
GeoTrust Inc.	586,376 (19.9%)
Comodo CA Limited	374,769 (12.7%)
VeriSign, Inc.	317,934 (10.8%)
Thawte, Inc.	228,779 (7.8%)
DigiCert Inc	145,232 (4.9%)
GlobalSign	117,685 (4.0%)
Starfield Technologies	94,794 (3.2%)
StartCom Ltd.	88,729 (3.0%)
Entrust, Inc.	76,929 (2.6%)

Table 3: **Top 10 Certificate Authorities**—We used ZMap to perform regular comprehensive scans of HTTPS hosts in order gain visibility into the CA ecosystem. Ten organizations control 86% of browser trusted certificates.

viously opaque distributed systems on the Internet. For instance, e-commerce and secure web transactions inherently depend on browser trusted TLS certificates. However, there is currently little oversight over browser trusted certificate authorities (CAs) or issued certificates. Most CAs do not publish lists of the certificates they have signed, and, due to delegation of authority to intermediate CAs, it is unknown what set of entities have the technical ability to sign browser-trusted certificates at any given time.

To explore this potential, we used ZMap and our custom certificate fetcher to conduct regular scans over the past year and perform analysis on new high-profile certificates and CA certificates. Between April 2012 and June 2013, we performed 1.81 billion TLS handshakes, ultimately collecting 33.6 million unique X.509 certificates of which 6.2 million were browser trusted. We found and processed an average of 220,000 new certificates, 15,300 new browser trusted certificates, and 1.2 new CA certificates per scan. In our most recent scan, we identified 1,832 browser trusted signing certificates from 683 organizations and 57 countries. We observed 3,744 distinct browser-trusted signing certificates in total. Table 3 shows the most prolific CAs by leaf certificates issued.

Wide-scale visibility into CA behavior can help to identify security problems [10, 18]. We found two cases of misissued CA certificates. In the first case, we found a CA certificate that was accidentally issued to a Turkish transit provider. This certificate, C=TR, ST=ANKARA, L=ANKARA, O=EGO, OU=EGO BILGI ISLEM, CN=*.EGO.GOV.TR, was later found by Google after being used to sign a Google wildcard certificate and has since been revoked and blacklisted in common web browsers [20].

In the second case, we found approximately 1,300 CA certificates that were misissued by the Korean Government to government sponsored organizations such as schools and libraries. While these certificates had been issued with rights to sign additional certificates, a length constraint on the grandparent CA certificate prevented these organizations from signing new certificates. We do not include these Korean certificates in the CA totals above because they are unable to sign valid browser-trusted certificates.

4.2 Tracking Protocol Adoption

Researchers have previously attempted to understand the adoption of new protocols, address depletion, common misconfigurations, and vulnerabilities through active scanning [2, 10, 12, 14, 15, 27]. In many of these cases, these analyses have been performed on random samples of the IPv4 address space due to the difficulty of performing comprehensive scans [15, 27]. In cases where full scans were performed, they were completed over an extended period of time or through massive parallelization on cloud providers [10, 14]. ZMap lowers the barriers to entry and allows researchers to perform studies like these in a comprehensive and timely manner, ultimately enabling much higher resolution measurements than previously possible.

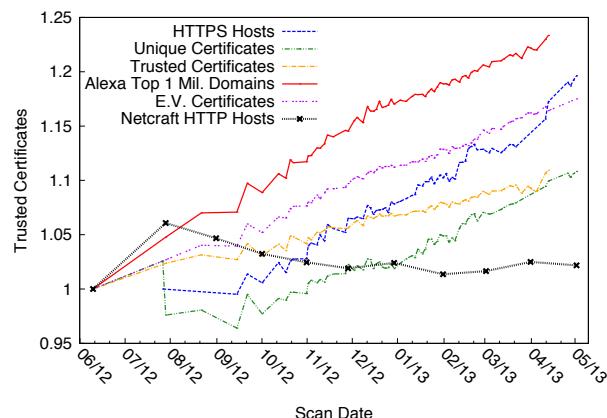


Figure 6: **HTTPS Adoption**—Data we collected using ZMap show trends in HTTPS deployment over one year. We observed 19.6% growth in hosts serving HTTPS.

Port	Service	Hit Rate (%)
80	HTTP	1.77
7547	CWMP	1.12
443	HTTPS	0.93
21	FTP	0.77
23	Telnet	0.71
22	SSH	0.57
25	SMTP	0.43
3479	2-Wire RPC	0.42
8080	HTTP-alt/proxy	0.38
53	DNS	0.38

Table 4: **Top 10 TCP ports**—We scanned 2.15 million hosts on TCP ports 0–9175 and observed what fraction were listening on each port. We saw a surprising number of open ports associated with embedded devices, such as ports 7547 (CWMP) and 3479 (2-Wire RPC).

To illustrate this application, we tracked the adoption of HTTPS using 158 Internet-wide scans over the past year. Notably, we find a 23% increase in HTTPS use among Alexa Top 1 Million websites and a 10.9% increase in the number of browser-trusted certificates. During this period, the Netcraft Web Survey [26] finds only a 2.2% increase in the number of HTTP sites, but we observe an 8.5% increase in sites using HTTPS. We plot these trends in Figure 6.

We can also gain instantaneous visibility into the deployment of multiple protocols by performing many ZMaps scans of different ports. We scanned 0.05% samples of the IPv4 address space on each TCP port below 9175 to determine the percentage of hosts that were listening on each port. This experiment requires the same number of packets as over 5 Internet-wide scans of a single port, yet we completed it in under a day using ZMap. Table 4 shows the top 10 open ports we observed.

4.3 Enumerating Vulnerable Hosts

With the ability to perform rapid Internet-wide scans comes the potential to quickly enumerate hosts that suffer from specific vulnerabilities [2]. While this can be a powerful defensive tool for researchers—for instance, to measure the severity of a problem or to track the application of a patch—it also creates the possibility for an attacker with control of only a small number of machines to scan for and infect all public hosts suffering from a new vulnerability within minutes.

UPnP Vulnerabilities To explore these applications, we investigated several recently disclosed vulnerabilities in common UPnP frameworks. On January 29, 2013, HD Moore publicly disclosed several vulnerabilities in common UPnP libraries [25]. These vulnerabilitiesulti-

mately impacted 1,500 vendors and 6,900 products, all of which can be exploited to perform arbitrary code execution with a single UDP packet. Moore followed responsible disclosure guidelines and worked with manufacturers to patch vulnerable libraries, and many of the libraries had already been patched at the time of disclosure. Despite these precautions, we found that at least 3.4 million devices were still vulnerable to the problem in February 2013.

To measure this, we created a custom ZMap probe module that performs a UPnP discovery handshake. We were able to develop this 150-SLOC module from scratch in approximately four hours and performed a comprehensive scan of the IPv4 address space for publicly available UPnP hosts on February 11, 2013, which completed in under two hours. This scan found 15.7 million publicly accessible UPnP devices, of which 2.56 million (16.5%) were running vulnerable versions of the Intel SDK for UPnP Devices, and 817,000 (5.2%) used vulnerable versions of MiniUPnPd.²

Given that these vulnerable devices can be infected with a single UDP packet [25], we note that these 3.4 million devices could have been infected in approximately the same length of time—much faster than network operators can reasonably respond or for patches to be applied to vulnerable hosts. Leveraging methodology similar to ZMap, it would only have taken a matter of hours from the time of disclosure to infect every publicly available vulnerable host.

Weak Public Keys As part of our regular scans of the HTTPS ecosystem, we tracked the mitigation of the 2008 Debian weak key vulnerability [3] and the weak and shared keys described by Heninger et al. in 2012 [14]. Figure 7 shows several trends over the past year.

In our most recent scan, we found that 44,600 unique certificates utilized factorable RSA keys and are served on 51,000 hosts, a 20% decrease from 2011 [14]. Four of these certificates were browser trusted; the last was signed in August 2012. Similarly, we found 2,743 unique certificates that contained Debian weak keys, of which 96 were browser trusted, a 34% decrease from 2011 [14]. The last browser trusted certificate containing a Debian weak key was signed in January 2012. We also observed a 67% decrease in the number of browser-trusted certificates that contained default public keys used for Citrix remote access products [14].

We created an automated process that alerts us to the discovery of new browser-trusted certificates containing factorable RSA keys, Debian weak keys, or default Citrix keys as soon as they are found, so that we can attempt to notify the certificate owners about the vulnerability.

²Moore reported many more UPnP hosts [25] but acknowledges that his scans occurred over a 5 month period and did not account for hosts being counted multiple times due to changing IP addresses.

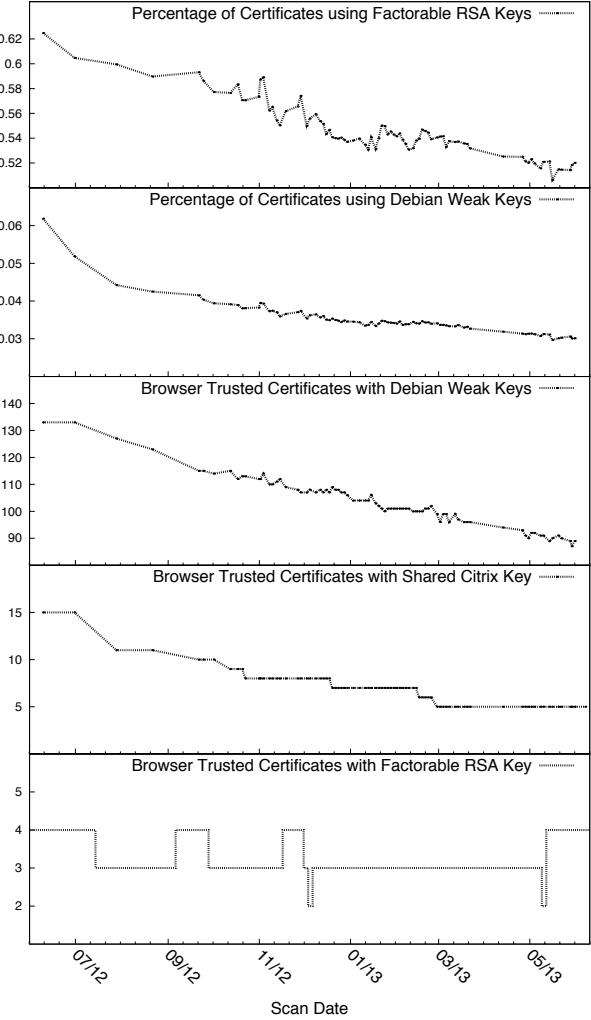


Figure 7: **Trends in HTTPS Weak Key Usage**—To explore how ZMap can be used to track the mitigation of known vulnerabilities, we monitored the use of weak HTTPS public keys from May 2012 through June 2013.

4.4 Discovering Unadvertised Services

The ability to perform comprehensive Internet scans implies the potential to uncover unadvertised services that were previously only accessible with explicit knowledge of the host name or address. For example, Tor bridges are intentionally not published in order to prevent ISPs and government censors from blocking connections to the Tor network [35]. Instead, the Tor Project provides users with the IP addresses of a small number of bridges based on their source address. While Tor developers have acknowledged that bridges can in principle be found by Internet-wide scanning [9], the set of active bridges is constantly changing, and the data would be stale by the time a long running scan was complete. However, high-speed scanning might be used to mount an effective attack.

To confirm this, we performed Internet wide-scans on ports 443 and 9001, which are common ports for Tor bridges and relays, and applied a set of heuristics to identify likely Tor nodes. For hosts with one of these ports open, we performed a TLS handshake using a specific set of cipher suites supported by Tor’s “v1 handshake.” When a Tor relay receives this set of cipher suites, it will respond with a two-certificate chain. The signing (“Certificate Authority”) certificate is self-signed with the relay’s identity public key and uses a subject name of the form “CN=www.X.com”, where X is a randomized alphanumeric string. This pattern matched 67,342 hosts on port 443, and 2,952 hosts on port 9001.

We calculated each host’s identity fingerprint and checked whether the SHA1 hash appeared in the public Tor metrics list for bridge pool assignments. Hosts we found matched 1,170 unique bridge fingerprints on port 443 and 419 unique fingerprints on port 9001, with a combined total of 1,534 unique fingerprints (some were found on both ports). From the bridge pool assignment data, we see there have been 1,767–1,936 unique fingerprints allocated at any given time in the recent past, which suggests that we were able to identify 79–86% of allocated bridges at the time of the scan. The unmatched fingerprints in the Tor metrics list may correspond to bridges we missed, offline bridges, or bridges configured to use a port other than 9001 or 443.

In response to other discovery attacks against Tor bridges [38], the Tor project has started to deploy obfsproxy [36], a wrapper that disguises client–bridge connections as random data in order to make discovery by censors more difficult. Obfsproxy nodes listen on randomized ports, which serves as a defense against discovery by comprehensive scanning.

4.5 Monitoring Service Availability

Active scanning can help identify Internet outages and disruptions to service availability without an administrative perspective. Previous studies have shown that active surveying (ICMP echo request scans) can help track Internet outages, but they have either scanned small subsets of the address space based on preconceived notions of where outages would occur or have performed random sampling [9, 13, 31]. High speed scanning allows scans to be performed at a high temporal resolution through sampling or comprehensively. Similarly, scanning can help service providers identify networks and physical regions that have lost access to their service.

In order to explore ZMap’s potential for tracking service availability, we performed continuous scans of the IPv4 address space during Hurricane Sandy to track its impact on the East Coast of the United States. We show a snapshot of outages caused by the hurricane in Figure 8.

4.6 Privacy and Anonymous Communication

The advent of comprehensive high-speed scanning raises potential new privacy threats, such as the possibility of tracking user devices between IP addresses. For instance, a company could track home Internet users between dynamically assigned IP addresses based on the HTTPS certificate or SSH host key presented by many home routers and cable modems. This would allow tracking companies to extend existing IP-based tracking beyond the length of DHCP leases.

In another scenario, it may be possible to track travelers. In 2006 Scholz et al. presented methods for fingerprinting SIP devices [30] and other protocols inadvertently expose unique identifiers such as cryptographic keys. Such features could be used to follow a specific mobile host across network locations. These unique fingerprints, paired with publicly available network data and commercial geolocation databases, could allow an attacker to infer relationships and travel patterns of a specific individual.

The ability to rapidly send a single packet to all IPv4 addresses could provide the basis for a system of anonymous communication. Rather than using the scanner to send probes, it could be used to broadcast a short encrypted message to every public IP address. In this scenario, it would be impossible to determine the desired destination host. If the sender is on a network that does not use ingress filtering, it could also spoof source addresses to obscure the sender’s identity. This style of communication could be of particular interest to botnet operators, because it would allow infected hosts to remain dormant indefinitely while waiting for instructions, instead of periodically checking in with command and control infrastructure and potentially revealing their existence.

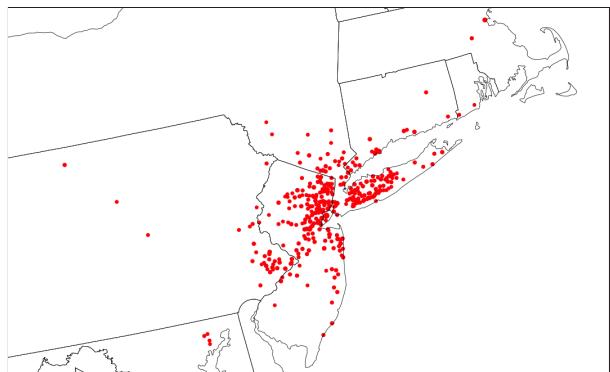


Figure 8: Outages in the Wake of Hurricane Sandy — We performed scans of port 443 across the entire IPv4 address space every 2 hours from October 29–31, 2013 to track the impact of Hurricane Sandy on the East Coast of the United States. Here, we show locations with more than a 30% decrease in the number of listening hosts.

5 Scanning and Good Internet Citizenship

We worked with senior colleagues and our local network administrators to consider the ethical implications of high-speed Internet-wide scanning and to develop a series of guidelines to identify and reduce any risks. Such scanning involves interacting with an enormous number of hosts and networks worldwide. It would be impossible to request permission in advance from the owners of all these systems, and there is no IP-level equivalent of the HTTP robots exclusion standard [19] to allow systems to signal that they desire not to be scanned. If we are to perform such scanning at all, the most we can do is try to minimize any potential for harm and give traffic recipients the ability to opt out of further probes.

High-speed scanning uses a large amount of bandwidth, so we need to ensure that our activities do not cause service degradation to the source or target networks. We confirmed with our local network administrators that our campus network and upstream provider had sufficient capacity for us to scan at gigabit speeds. To avoid overwhelming destination networks, we designed ZMap to scan addresses according to a random permutation. This spreads out traffic to any given destination network across the length of the scan. In a single probe TCP scan, an individual destination address receives one 40 byte SYN packet. If we scan at full gigabit speed, each /24 network block will receive a packet about every 10.6 seconds (3.8 bytes/s), each /16 network every 40 ms (1000 bytes/s), and each /8 network every 161 μ s (250,000 bytes/s) for the 44 minute duration of the scan. These traffic volumes should be negligible for networks of these sizes.

Despite these precautions, there is a small but nonzero chance that any interaction with remote systems might cause operational problems. Moreover, users or network administrators who observe our scan traffic might be alarmed, in the mistaken belief that they are under attack. Many may be unable to recognize that their systems are not being uniquely targeted and that these scans are not malicious in nature, and might waste resources responding. Some owners of target systems may simply be annoyed and want our scans to cease. To minimize the risks from these scenarios, we took several steps to make it easy for traffic recipients to learn why they were receiving probes and to have their addresses excluded from scanning if so desired.

First, we configured our source addresses to present a simple website on port 80 that describes the nature and purpose of the scans. The site explains that we are not targeting individual networks or attempting to obtain access to private systems, and it provides a contact email address to request exclusion from future scans. Second, we set reverse DNS records for our source addresses to “researchscanx.eecs.umich.edu” in order to signal that traffic

-
1. Coordinate closely with local network admins to reduce risks and handle inquiries.
 2. Verify that scans will not overwhelm the local network or upstream provider.
 3. Signal the benign nature of the scans in web pages and DNS entries of the source addresses.
 4. Clearly explain the purpose and scope of the scans in all communications.
 5. Provide a simple means of opting out, and honor requests promptly.
 6. Conduct scans no larger or more frequent than is necessary for research objectives.
 7. Spread scan traffic over time or source addresses when feasible.
-

Table 5: **Recommended Practices** — We offer these suggestions for other researchers conducting fast Internet-wide scans as guidelines for good Internet citizenship.

fic from these hosts was part of an academic research study. Third, we coordinated with IT teams at our institution who might receive inquiries about our scan traffic.

For our ongoing Internet-wide HTTPS surveys (our largest-volume scanning effort), we took additional steps to further reduce the rate of false alarms from intrusion detection systems. Rather than scanning at full speed, we conducted each of these scans over a 12 hour period. We also configured ZMap to use a range of 64 source addresses and spread out probe traffic among them. We recognize that there is a difficult balance to strike here: we do not want to conceal our activities from system administrators who would want to know about them, but we also do not want to divert IT support resources that would otherwise be spent dealing with genuine attacks.

We provide a summary of the precautions we took in Table 5 as a starting point for future researchers performing Internet-wide scans. It should go without saying that scan practitioners should refrain from exploiting vulnerabilities or accessing protected resources, and should comply with any special legal requirements in their jurisdictions.

5.1 User Responses

We performed approximately 200 Internet-wide scans over the course of a year, following the practices described above. We received e-mail responses from 145 scan traffic recipients, which we classify in Table 6. In most cases, these responses were informative in nature, notifying us that we may have had infected machines, or were civil requests to be excluded from future scans. The vast majority of these requests were received at our institution’s WHOIS abuse address or at the e-mail address published on the scan source IP addresses, but we also received

Small/Medium Business	41
Home User	38
Other Corporation	17
Academic Institution	22
Government/Military	15
Internet Service Provider	2
Unknown	10
Total Entities	145

Table 6: **Responses by Entity Type**

responses sent to our institution’s help desk, our chief security officer, and our departmental administrator.

We responded to each inquiry with information about the purpose of our scans, and we immediately excluded the sender’s network from future scans upon request. In all, we excluded networks belonging to 91 organizations or individuals, totaling 3,753,899 addresses (0.11% of the public IPv4 address space). About 49% of the blacklisted addresses resulted from requests from two Internet service providers. We received 15 actively hostile responses that threatened to retaliate against our institution legally or to conduct a denial-of-service (DOS) attack against our network. In two cases, we received retaliatory DOS traffic, which was blacklisted by our upstream provider.

6 Related Work

Many network scanning tools have been developed, the vast majority of which have been optimized to scan small network segments. The most popular and well respected is Nmap (“Network Mapper”) [23], a versatile, multipurpose tool that supports a wide variety of probing techniques. Unlike Nmap, ZMap is specifically designed for Internet-wide scanning, and it achieves much higher performance in this application.

Leonard and Loguinov introduced IRLscanner, an Internet-scale scanner with the demonstrated ability to probe the advertised IPv4 address space in approximately 24 hours, ultimately scanning at 24,421 packets per second [22]. IRLscanner is able to perform scanning at this rate by utilizing a custom Windows network driver, IRLstack [33]. However, IRLscanner does not process responses, requires a custom network driver and a complete routing table for each scan, and was never released to the research community. In comparison, we developed ZMap as a self-contained network scanner that requires no custom drivers, and we are releasing it to the community under an open source license. We find that ZMap can scan at 1.37 million packets per second, 56 times faster than IRLScanner was shown to operate.

Previous work has developed methods for sending and receiving packets at fast network line speeds, including PF_RING [8], PacketShader [11], and netmap [28], all of which replace parts of the Linux kernel network stack. However, as discussed in Section 3.1, we find that the Linux kernel is capable of sending probe packets at gigabit Ethernet line speed without modification. In addition, libpcap is capable of processing responses without dropping packets as only a small number of hosts respond to probes. The bottlenecks in current tools are in the scan methodology rather than the network stack.

Many projects have performed Internet-scale network surveys (e.g., [10, 12, 14, 15, 25, 27]), but this has typically required heroic effort on the part of the researchers. In 2008, Heidemann et al. presented an Internet census in which they attempted to determine IPv4 address utilization by sending ICMP packets to allocated IP addresses; their scan of the IPv4 address space took approximately three months to complete and claimed to be the first Internet-wide survey since 1982 [12]. Two other recent works were motivated by studying the security of HTTPS. In 2010, the Electronic Frontier Foundation (EFF) performed a scan of the public IPv4 address space using Nmap [23] to find hosts with port 443 (HTTPS) open as part of their SSL Observatory Project [10]; their scans were performed on three Linux servers and took approximately three months to complete. Heninger et al. performed a scan of the IPv4 address space on port 443 (HTTPS) in 2011 and on port 22 (SSH) in 2012 as part of a study on weak cryptographic keys [14]. The researchers were able to perform a complete scan in 25 hours by concurrently performing scans from 25 Amazon EC2 instances at a cost of around \$300. We show that ZMap could be used to collect the same data much faster and at far lower cost.

Most recently, an anonymous group performed an illegal “Internet Census” in 2012, using the self-named Carna Botnet. This botnet used default passwords to log into thousands of telnet devices. After logging in, the botnet scanned for additional vulnerable telnet devices and performed several scans over the IPv4 space, comprising over 600 TCP ports and 100 UDP ports over a 3-month period [1]. With this distributed architecture, the authors claim to have been able to perform a single-port scan survey over the IPv4 space in about an hour. ZMap can achieve similar performance without making use of stolen resources.

7 Future Work

While we have demonstrated that efficiently scanning the IPv4 address space at gigabit line speeds is possible, there remain several open questions related to performing network surveys over other protocols and at higher speeds.

Scanning IPv6 While ZMap is capable of rapidly scanning the IPv4 address space, brute-force scanning methods will not suffice in the IPv6 address space, which is far too large to be fully enumerated [7]. This places current researchers in a window of opportunity to take advantage of fast Internet-wide scanning methodologies before IPv6-only services become common place. New methodologies will need to be developed specifically for performing surveys of the IPv6 address space.

10gigE Surveys ZMap is currently limited by the speed of widely available gigabit networks, and we have not tested how well its architecture will scale as 10gigE and faster networks become available. There is motivation to perform the fastest scans possible as they will provide the truest sense of a snapshot of the Internet at a given point in time. However, these faster rates also open questions of overloading destination networks and hosts. The dynamics of performing scans at 10gigE have not yet been explored.

Server Name Indication Server Name Indication (SNI) is a TLS protocol extension that allows a server to present multiple certificates on the same IP address [6]. SNI has not yet been widely deployed, primarily because Internet Explorer does not support it on Windows XP hosts [21]. However, its inevitable growth will make scanning HTTPS sites more complicated, since simply enumerating the address space will miss certificates that are only presented with the correct SNI hostname.

Scanning Exclusion Standards If Internet-wide scanning becomes more widespread, it will become increasingly burdensome for system operators who do not want to receive such probe traffic to manually opt out from all benign sources. Further work is needed to standardize an exclusion signaling mechanism, akin to HTTP’s robots.txt [19]. For example, a host could use a combination of protocol flags to send a “do-not-scan” signal, perhaps by responding to unwanted SYNs with the SYN and RST flags, or a specific TCP option set.

8 Conclusion

We are living in a unique period in the history of the Internet: typical office networks are becoming fast enough to exhaustively scan the IPv4 address space, yet IPv6 (with its much larger address space) has not yet been widely deployed. To help researchers make the most of this window of opportunity, we developed ZMap, a network scanner specifically architected for performing fast, comprehensive Internet-wide surveys.

We experimentally showed that ZMap is capable of scanning the public IPv4 address space on a single port in under 45 minutes, at 97% of the theoretical maximum

speed for gigabit Ethernet and with an estimated 98% coverage of publicly available hosts. We explored the security applications of high speed scanning, including the ability to track protocol adoption at Internet scale and to gain timely insight into opaque distributed systems such as the certificate authority ecosystem. We further showed that high-speed scanning also provides new attack vectors that we must consider when defending systems, including the ability to uncover hidden services, the potential to track users between IP addresses, and the risk of infection of vulnerable hosts en masse within minutes of a vulnerability’s discovery.

We hope ZMap will elevate Internet-wide scanning from an expensive and time-consuming endeavor to a routine methodology for future security research. As Internet-wide scanning is conducted more routinely, practitioners must ensure that they act as good Internet citizens by minimizing risks to networks and hosts and being responsive to inquiries from traffic recipients. We offer the recommendations we developed while performing our own scans as a starting point for further conversations about good scanning practice.

Acknowledgments

The authors thank the exceptional sysadmins at the University of Michigan for their help and support throughout this project. This research would not have been possible without Kevin Cheek, Laura Fink, Paul Howell, Don Winsor, and others from ITS, CAEN, and DCO. We thank Michael Bailey for advice on many aspects of the work and Oguz Durumeric for his discussion of generating permutations of the IPv4 address space. We also thank Brad Campbell, Peter Eckersley, James Kasten, Pat Pannuto, Amir Rahmati, Michael Rushanan, and Seth Schoen. This work was supported in part by NSF grant CNS-1255153 and by an NSF Graduate Research Fellowship.

References

- [1] Anonymous. Internet census 2012. <http://census2012.sourceforge.net/paper.html>, March 2013.
- [2] G. Bartlett, J. Heidemann, and C. Papadopoulos. Understanding passive and active service discovery. In *7th ACM SIGCOMM conference on Internet measurement (IMC)*, pages 57–70, 2007.
- [3] L. Bello. DSA-1571-1 OpenSSL—Predictable random number generator, 2008. Debian Security Advisory. <http://www.debian.org/security/2008/dsa-1571>.
- [4] D. J. Bernstein. SYN cookies. <http://cr.yp.to/syncookies.html>, 1996.
- [5] J. Black, S. Halevi, H. Krawczyk, T. Krovetz, and P. Rogaway. UMAC: Fast and secure message authentication. In *Advances in Cryptology—CRYPTO ’99*, 1999.

- [6] S. Blake-Wilson, M. Nystrom, D. Hopwood, J. Mikkelsen, and T. Wright. Transport Layer Security (TLS) Extensions. RFC 3546 (Proposed Standard), June 2003.
- [7] T. Chown. IPv6 Implications for Network Scanning. RFC 5157 (Informational), March 2008.
- [8] L. Deri. Improving passive packet capture: Beyond device polling. In *4th International System Administration and Network Engineering Conference (SANE)*, 2004.
- [9] R. Dingledine. Research problems: Ten ways to discover Tor bridges. <http://blog.torproject.org/blog/research-problems-ten-ways-discover-tor-bridges>, October 2011.
- [10] P. Eckersley and J. Burns. An observatory for the SSLiverse. Talk at Defcon 18 (2010). <https://www.eff.org/files/DefconSSLiverse.pdf>.
- [11] S. Han, K. Jang, K. Park, and S. Moon. PacketShader: A GPU-accelerated software router. In *ACM SIGCOMM*, September 2010.
- [12] J. Heidemann, Y. Pradkin, R. Govindan, C. Papadopoulos, G. Bartlett, and J. Bannister. Census and survey of the visible Internet. In *8th ACM SIGCOMM conference on Internet measurement (IMC)*, 2008.
- [13] J. Heidemann, L. Quan, and Y. Pradkin. A preliminary analysis of network outages during hurricane sandy. Technical Report ISI-TR-2008-685b, USC/Information Sciences Institute, November 2012.
- [14] N. Heninger, Z. Durumeric, E. Wustrow, and J. A. Halderman. Mining your Ps and Qs: Detection of widespread weak keys in network devices. In *21st USENIX Security Symposium*, August 2012.
- [15] R. Holz, L. Braun, N. Kammenhuber, and G. Carle. The SSL landscape: A thorough analysis of the X.509 PKI using active and passive measurements. In *11th ACM SIGCOMM conference on Internet measurement (IMC)*, pages 427–444, 2011.
- [16] IANA. IPv4 address space registry. <http://www.iana.org/assignments/ipv4-address-space/ipv4-address-space.xml>.
- [17] V. Jacobson, C. Leres, and S. McCanne. libpcap. Lawrence Berkeley National Laboratory, Berkeley, CA. Initial release June 1994.
- [18] J. Kasten, E. Wustrow, and J. A. Halderman. Cage: Taming certificate authorities by inferring restricted scopes. In *17th International Conference on Financial Cryptography and Data Security (FC)*, 2013.
- [19] M. Koster. A standard for robot exclusion. <http://www.robotstxt.org/orig.html>, 1994.
- [20] A. Langley. Enhancing digital certificate security. Google Online Security Blog, <http://googleonlinesecurity.blogspot.com/2013/01/enhancing-digital-certificate-security.html>, January 2013.
- [21] E. Law. Understanding certificate name mismatches. <http://blogs.msdn.com/b/ieinternals/archive/2009/12/07/certificate-name-mismatch-warnings-and-server-name-indication.aspx>, December 2009.
- [22] D. Leonard and D. Loguinov. Demystifying service discovery: Implementing an Internet-wide scanner. In *10th ACM SIGCOMM conference on Internet measurement (IMC)*, pages 109–122, 2010.
- [23] Gordon Fyodor Lyon. *Nmap Network Scanning: The Official Nmap Project Guide to Network Discovery and Security Scanning*. Insecure, USA, 2009.
- [24] N. Mathewson and N. Provos. libevent—An event notification library. <http://libevent.org>.
- [25] HD Moore. Security flaws in universal plug and play. Unplug. Don't Play, January 2013. <http://community.rapid7.com/servlet/JiveServlet/download/2150-1-16596/SecurityFlawsUPnP.pdf>.
- [26] Netcraft, Ltd. Web server survey. <http://news.netcraft.com/archives/2013/05/03/may-2013-web-server-survey.html>, May 2013.
- [27] N. Provos and P. Honeyman. ScanSSH: Scanning the Internet for SSH servers. In *16th USENIX Systems Administration Conference (LISA)*, 2001.
- [28] Luigi Rizzo. netmap: A novel framework for fast packet I/O. In *2012 USENIX Annual Technical Conference*, 2012.
- [29] S. Sanfilippo and P. Noordhuis. Redis. <http://redis.io>.
- [30] H. Scholz. SIP stack fingerprinting and stack difference attacks. Talk at Blackhat 2006. <http://www.blackhat.com/presentations/bh-usa-06/BH-US-06-Scholz.pdf>.
- [31] A. Schulman and N. Spring. Pingin' in the rain. In *11th ACM SIGCOMM conference on Internet measurement (IMC)*, pages 19–28, 2011.
- [32] K. Sklower. A tree-based packet routing table for Berkeley Unix. In *Winter USENIX Conference*, 1991.
- [33] M. Smith and D. Loguinov. Enabling high-performance Internet-wide measurements on Windows. In *11th International Conference on Passive and Active Measurement (PAM)*, pages 121–130. Springer, 2010.
- [34] W. R. Stevens and G. R. Wright. *TCP/IP Illustrated: The Implementation*, volume 2. Addison-Wesley, 1995.
- [35] Tor Project. Tor Bridges. <https://www.torproject.org/docs/bridges>, 2008.
- [36] Tor Project. obfsproxy. <https://www.torproject.org/projects/obfsproxy.html.en>, 2012.
- [37] J. Viega, M. Messier, and P. Chandra. *Network Security with OpenSSL: Cryptography for Secure Communications*. O'Reilly, 2002.
- [38] T. Wilde. Great Firewall Tor probing. <https://gist.github.com/twilde/da3c7a9af01d74cd7de7>, 2012.

Eradicating DNS Rebinding with the Extended Same-Origin Policy

Martin Johns

SAP Research

martin.johns@sap.com

Sebastian Lekies

SAP Research

sebastian.lekies@sap.com

Ben Stock

FAU Erlangen-Nuremberg

ben.stock@cs.fau.de

Abstract

The Web’s principal security policy is the Same-Origin Policy (SOP), which enforces origin-based isolation of mutually distrusting Web applications. Since the early days, the SOP was repeatedly undermined with variants of the DNS Rebinding attack, allowing untrusted script code to gain illegitimate access to protected network resources. To counter these attacks, the browser vendors introduced countermeasures, such as DNS Pinning, to mitigate the attack. In this paper, we present a novel DNS Rebinding attack method leveraging the HTML5 Application Cache. Our attack allows reliable DNS Rebinding attacks, circumventing all currently deployed browser-based defense measures. Furthermore, we analyze the fundamental problem which allows DNS Rebinding to work in the first place: The SOP’s main purpose is to ensure security boundaries of Web servers. However, the Web servers themselves are only indirectly involved in the corresponding security decision. Instead, the SOP relies on information obtained from the domain name system, which is not necessarily controlled by the Web server’s owners. This mismatch is exploited by DNS Rebinding. Based on this insight, we propose a light-weight extension to the SOP which takes Web server provided information into account. We successfully implemented our extended SOP for the Chromium Web browser and report on our implementation’s interoperability and security properties.

1 Introduction

The Web has won. No other platform for distributed applications can rival the Web’s ubiquity and flexibility. The functionality demands of the ever-expanding Web application paradigm caused the browser to evolve from a simple program to display hypertext documents into a full-fledged runtime environment for sophisticated, networked applications. This evolution is still in full effect, with HTML5 and related JavaScript APIs being the latest

addition to the browser model. In the context of Web applications, fundamental security properties are governed by the Same-Origin Policy (SOP): The SOP is the Web’s principal security policy. It provides origin-based isolation of Web applications.

In the recent past, low-level vulnerabilities have become considerably harder to find and exploit. Hence, the ever growing capabilities of the Web browser make it an increasingly interesting offensive tool for attackers [8]: The Web browser runs behind the firewall within the boundaries of the internal network and executes code that was retrieved from the Internet. Thus, the SOP constitutes the only barrier between attacker provided code and the crown jewels in the internal network. Unfortunately, the SOP is far from bulletproof: Soon after the introduction of the policy in 1996, clever students at Princeton university found a way to utilize attacker controlled DNS settings to subvert the policy [25]. The underlying attack is today known as “DNS Rebinding” [14]. Since then, DNS Rebinding remained a constant problem of the SOP that was (re)discovered multiple times and, subsequently, attempted to be fixed.

In this paper, we demonstrate how the HTML5 Offline Application Cache can be misused to conduct reliable DNS Rebinding attacks. Our attack works with all major browsers, circumvents all current browser-based countermeasures, and affects most browser-based scripting runtime environments (JavaScript, Flash, Silverlight). Furthermore, we revisit the underlying problem of the SOP and propose a light-weight but powerful extension to the policy, which tackles the root cause of the problem.

Contribution and paper organization: After covering the required technical background (see Sec. 2) and the history of DNS Rebinding (see Sec. 3), we make the following contributions:

- *DNS Rebinding and the AppCache (Section 4):* We present a novel attack technique, capable of circumventing any existing browser-based countermeasure

against DNS Rebinding. In our attack, we utilize the HTML5 Offline AppCache to persist a malicious script until any domain-to-IP information is lost. In theory, caching-based attack scenarios are already known. However, the unpredictable and short-lived nature of the browser’s caching behavior rendered them fragile to a level of unfeasibility. In this paper, we show how the unique characteristics of the AppCache can be leveraged by the attacker to create highly reliable DNS Rebinding attacks.

- *Vulnerability demonstration (Section 5):* To validate our attack method and to demonstrate its severity, we present two practical attacks on real-world applications utilizing Web interfaces. For our experiments, we chose the light-weight proxy server Polipo, and the Unix-based printing system CUPS. The effects of our demonstration exploits range from simple information leakage to remote code execution.
- *Extended Same-Origin Policy (Section 6):* We analyze the fundamental problem that causes DNS Rebinding to work. Thereby we identify a mismatch between the semantics and the implementation of the Same-Origin Policy: The SOP’s main purpose is to ensure security boundaries of Web servers. However, the Web servers themselves are only indirectly involved in the corresponding security decision. In order to overcome this mismatch, we propose a light-weight extension to the Same-Origin Policy that considers server-provided origin information. Our extended SOP reliably defeats DNS Rebinding attacks while increasing interoperability with mechanisms that rely on flexible DNS setups, such as DNS-based load-balancing or Content Distribution Networks.
- *Implementation for the Chromium browser (Section 7):* To demonstrate the practical applicability of our approach, we implemented it for the open-source browser Chromium. The implementation required in total 34 lines of code and does not cause a perceivable performance overhead.

We end the paper with a review of related work (Sec. 8) and a conclusion (Sec. 9).

2 Technical Background

In this section, we briefly cover selected topics that are necessary to discuss the paper’s technical content.

2.1 The Same-Origin Policy

The Same-Origin Policy (SOP) was designed to enforce origin-based isolation of mutually distrusting Web applications. Several variants of the policy exist [38]. In this section, we focus the SOP for JavaScript [31].

In general, the Same-Origin Policy [14] is the main security policy for all active content that is executed in a

Web browser within the context of a Web page. This policy restricts all client-side interactions to objects which share the same origin. In this context, an object’s origin is defined by the domain, port, and protocol, which were utilized to obtain the object. Hence, a JavaScript snippet is only allowed to access a resource if its own origin exactly matches the origin of the resource. The SOP for plug-in based script content, such as Flash or Silverlight, enforces similar rules.

Developers can adjust a JavaScript snippet’s origin slightly by modifying the `document.domain` DOM property: The value of this property can be set to omit the values of subdomains up to the second level domain value (e.g., relaxing `www.example.org` to `example.org`). This process is known under the term “domain relaxation”.

2.2 The HTML5 AppCache

Modern Web applications have one crucial disadvantage compared to desktop applications: Such applications can only be used when a network connection is available. In order to eradicate this disadvantage the HTML5 Offline Application Cache (*AppCache*) was introduced [10]. The AppCache is a mechanism that can be utilized to store resources (such as HTML documents, images, etc) within the browser for offline usage. In order to employ the Application Cache, a Web site may provide a manifest file containing a list of resources. The manifest file’s location can be specified within the `manifest` attribute of a document’s `HTML` tag as shown in Listing 1.

Listing 1: HTML5 Manifest attribute

```
1 <html manifest="manifest.mf">
2 [...]
```

When a browser discovers this attribute, it fetches the file and caches the listed resources within the AppCache. Listing 2 shows an exemplary manifest file that advises the browser to cache `index.php` as well as a flash applet named `flash.swf`. As soon as a cached resource is requested again, the Application Cache returns the cached HTTP response even if an Internet connection is available. After each access to the AppCache, the browser downloads the manifest file again to check whether it has changed. The resources within the AppCache are only updated if the manifest has changed - otherwise the resources reside within the cache even if their server-side counterparts have changed.

Listing 2: Exemplary manifest file (excerpt)

```
1 CACHE MANIFEST
2
3 http://example.org/index.php
4 http://example.org/flash.swf
```

3 DNS Rebinding

DNS Rebinding is a term introduced by [14], which describes a class of Web browser-based attacks that undermine the SOP through sophisticated mapping of DNS entries to restricted network resources. In Section 3.1 we give a full account on the historical development of these attack methods. In the remainder of this section, we briefly revisit the basic attack pattern.

The decision if a given JavaScript is granted access to a certain resource (e.g., browser window, or network location) is governed by the SOP. As explained earlier, the SOP relies on the *domain* property of the respective entity’s origins. However, the HTTP protocol does not require any information about the requested *domain*. The actual HTTP connections are made using the server’s IP.

An attacker can exploit this fact an attacker issues a very short-lived DNS entry for an attacker controlled web page. Whenever a victim visits this particular Web site, the victim’s browser fetches the DNS entry, connects to the provided IP address and downloads attacker controlled JavaScript or plug-in code. This code is only capable of creating network connection to same-domain hosts due to the SOP. In the meantime, the DNS entry expired and therefore, as soon as another request is conducted towards the same domain, a new DNS entry has to be fetched. The attacker is able to exploit this behavior by altering the domain-to-IP-mapping. By providing an IP of the victim’s intranet, the browser connects to the intranet IP as soon as the JavaScript conducts a same-domain request (see Fig. 1). As the IP is not a part of the Same-Origin check, the policy is still fulfilled and, therefore, the attacker controlled script is granted access to the response of the intranet host. Thereby, potential offensive scenarios are not limited to information leakage attacks on internal servers. DNS Rebinding can, for instance, also be used to conduct click fraud, defeating IP-based authentication, or hijacking of IP addresses (refer to [14] for a comprehensive overview). Nonetheless, for readability reasons, from now on we will use the information leakage attack as the motivational example.

3.1 The History of DNS Rebinding

As we will show in this section, the history of DNS Rebinding reaches back in time to the early days in which the SOP just started to emerge. Over the years, the attack was discussed under several different names, including “anti-DNS pinning” [7] and “Quick-swap DNS” [21]. In this time, several variants of the rebinding attack have been developed, either with focus on different browser-based technologies [18, 25, 29], with new techniques to circumvent the implemented mitigation measures [3, 15, 28, 34], or with focus on novel attack targets [9, 14]. Nonetheless, the general technique re-

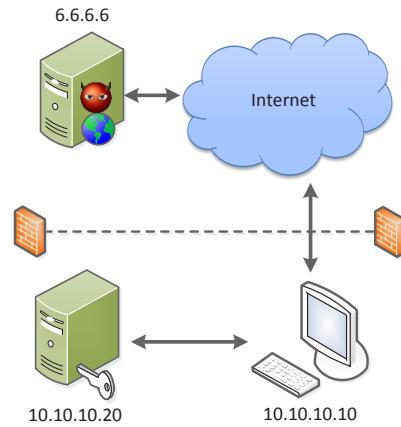


Figure 1: Intranet attack scenario

mained stable: Mapping an attacker-controlled DNS entry to a restricted network resource and subsequently using active browser content to access the resource.

In this section, we give a brief overview on the developments of the past years. In general, the history of DNS Rebinding can be divided into three distinct time spans, each starting with the (re)discovery of the basic issue for a separate browser-based technology: 1996 (Java applets), 2002 (JavaScript), 2006 (Flash, JavaScript, Java).

3.1.1 1996 - Java Applets

Princeton’s Secure Internet Programming group first mentioned the attack method in 1996 [25]. Back then, JavaScript networking capabilities were rather limited, while Java Applets already allowed comparatively sophisticated networking functionality [4, 25].

To be precise, the Princeton attack did not rely on DNS Rebinding per se. Instead, the attack utilized DNS records, which returned two IP addresses for the adversary’s domain: The IP of the attacker’s server, from which the applet was loaded and another IP pointing to the target of the attack. As the adversary controls the order of the values in the DNS response, the applet could be tricked to connect to the target system. To mitigate the issue, Java’s vendor SUN introduced strict IP based access control [24]: After the initial loading of an applet, the only IP the applet is allowed to access is the IP address it was originally obtained from, regardless of information provided by DNS. This restriction is maintained for the entire lifespan of the applet.

3.1.2 2002 - JavaScript

The Princeton attack was extended by Adam Megacz to JavaScript in 2002 [21]. Megacz presented two variants

of the attack. For one, he utilized domain relaxation. In this case, the malicious JavaScript was hosted on a sub-domain of the adversary’s server, e.g., `sub.attacker.org`. The DNS entry for the father domain `attacker.org` pointed to the internal host. After being loaded in the victim’s browser, the script relaxed its `document.domain` value to the father domain and, thus, was subsequently granted access to the internal server. The second attack variant, named “Quick-swap DNS” was roughly equivalent to the general attack scheme presented in Section 3.

In response to Megacz’s security advisory, Netscape implemented explicit “pinning” of the domain-to-IP mapping for the lifetime of the Web page. In addition, to mitigate the domain relaxation based attack, a patch was created that required both parties in a domain relaxation scenario to assign the `document.domain` property to the same value. Versions of Internet Explorer that followed Megacz disclosure, exposed behavior similar to Netscape’s browser. However, in 2007 Microsoft’s Dave Ross gave to record that the observed DNS pinning was incidental and not introduced as a security measure [30].

3.1.3 2006 - The full browser experience

In 2006, Martin Johns discovered a technique to reliably cause Firefox and Internet Explorer to drop any domain-to-IP mapping, which in turn re-enabled the rebinding attack for JavaScript [15]. In the following months, several additional DNS Rebinding attack methods were disclosed: Kanatoko showed that Flash applets were also susceptible to the attack [18]. Also, Johns and Kanatoko documented a method to use the LiveConnect JavaScript-to-Java bridge to utilize Java methods in rebinding attacks [16]. Moreover, two further methods were discovered which allowed DNS Rebinding attacks on Java Applets: Rios and McFeters [28] tricked Java’s applet cache by using multiple instances of the Java VM and David Byrne leveraged a mismatch in communication channels, in case the Java VM was configured to access the network with a Web proxy [3]. Finally, Dafydd Stuttard examined the effects of Web proxies on DNS pinning [34].

The susceptibility of the plug-in technologies Flash and Java enabled the usage of low-level socket communication in rebinding. This expanded the resulting attack surface towards non-HTTP network services. Furthermore, socket connections could be utilized to circumvent HTTP-based countermeasures, such as host-header checking [7].

Additionally, multiple public demonstrations on the capabilities of the attack vector have been given. Notable in this context are the experiments by Jackson et al. [14]: Using a specifically crafted Flash advertisement delivered by a major advertising network, the group was able to take over 27,480 unique IPs for a total amount

of less than 30 US dollars. In response to the disclosed attacks, the vendors of Flash and Java introduced further restrictions on their socket-level network capabilities.

3.2 Capabilities and limitations of available countermeasures

Over the years, several practical and experimental countermeasures to protect against DNS Rebinding attacks have been introduced.

3.2.1 DNS Pinning

As previously discussed, most browser and plug-in vendors primarily reacted to DNS Rebinding by introducing DNS Pinning. When DNS Pinning is used, a Web resource’s IP-to-DNS mapping is maintained for a prolonged timespan, ideally exceeding the lifetime of the resource. While being able to provide basic protection properties, DNS Pinning has security and functionality drawbacks: For one, DNS pinning is inherently incompatible with all technical measures that rely on dynamic and potentially changing DNS answers, such as load balancing, active failover, disaster recovery [1], or Content Distribution Networks. Also, DNS Pinning is unable to protect if Web proxies are part of the communication path to the server [21, 34] or in content caching scenarios (more on this in Section 4).

3.2.2 Limiting internal IP ranges

Due to the specific nature of DNS Rebinding, internal servers are the prime target of the attack. Hence, several techniques have been presented that protect internal network resources against external scripts. In general, these approaches primarily protect resources hosted on the “private” netblocks of the IPv4 space, as defined by RFC 1918 [26]. For one, such protection can be implemented on the DNS level: DNSWall [2] is a daemon that is designed to be used in conjunction with an existing recursive DNS resolver. It filters out RFC 1918 addresses in DNS responses. Also, the OpenDNS service offers a similar option [36]. Furthermore, similar protection can be achieved within the browser: Opera refuses script code which was obtained from an external source to access internal RFC 1918 IP ranges. The Firefox extensions NoScript [20] and LocalRodeo [17] can be configured to do the same.

The attempt to provide protection by restricting access to private IP ranges is necessarily incomplete. For one, network based access control is not limited to RFC 1918 ranges. In addition, bigger organizations, such as large companies or universities, do not always use

RFC 1918 addresses for their internal networks. Furthermore, with the growing support for IPv6 many use cases for RFC 1918 addresses cease to exist, as there is no shortage of IPv6 addresses. Finally, Craig Heffner has demonstrated [9], that even in cases where access to the private IP ranges is protected against DNS Rebinding attacks, under certain conditions the adversary can use rebinding to gain privileged access to local network resources, if these resource listens both on a private and a public IP address.

3.2.3 Application-layer protection of servers

Servers can implement active protection against the attack. A straight forward choice is requiring authentication before an internal server can be accessed. As the rebinding attack utilizes the adversary’s domain, pre-existing authentication credentials, such as session cookies, cannot be abused by the attacker and, hence, the restricted data should be safe. Additionally, servers can implement host-header checking: The attacker’s HTTP requests carry the domain name of the attacker’s server in their host-header. Hence, the attack can be spotted and the access can be stopped, which usually is done by throwing a 400/500 server error or responding with a standard error message. However, this measure does not resolve the issue completely. The browser still allows the script to omit the request and receive the response. So even though, the server’s data cannot be obtained, the attack vector may still leak valuable information to the attacker, such as validation that the server exists and material to do server-type and software fingerprinting. Also, while sounding straight-forward, host-header checking can be error-prone, as our experiments with CUPS has shown (see Sec. 5.2): Even though CUPS implements the check, the implementation is incomplete and grants an attacker access to a subset of the tool’s data. Both techniques have in common, that they have to be introduced manually for each server on the application layer.

4 DNS rebinding using HTML5 AppCache

In the previous section we explained the basic mechanisms of DNS Rebinding. In order to counter these attacks browser vendors introduced a technique called DNS Pinning. In this section we show how this technique can be circumvented to reliably conduct DNS Rebinding attacks using the HTML5 Offline Application Cache.

4.1 Rebinding HTML/JavaScript content

pinning is to avoid the interaction of content that is served via the same origin, but received from different hosts. As soon as a DNS query is conducted, the browser

pins the received domain-to-IP mapping. Subsequent requests conducted towards this origin are then exclusively sent to the host utilizing the “pinned” IP. Thus, while DNS pinning is active, content fetched from one origin always corresponds to the same host. Ideally, the pinning information should be stored as long as a resource resides within the browser. However, as mentioned already, DNS Pinning interferes with techniques such as load-balancing, active failover and disaster recovery [5]. The longer the pinning times, the bigger is the negative effect on these techniques. In the worst case, if the domain-to-IP mapping information are stored by the browser for an unlimited amount of time, these techniques would be more or less useless. Therefore, pinning durations differ substantially from browser to browser. However, all major browsers have one thing in common: As soon as the user closes the browser, the pinning information is automatically deleted. This also affects Web content which ended up in the browser’s cache. Hence, a hunch about potential DNS Rebinding issues through cached content existed for some time [32].

The basic attack via cached content is similar to the general DNS Rebinding attack as described in Section 3. This time, however, we assume that DNS Pinning is in place and therefore the basic attack does not work as described. When caching comes into play, an attacker can re-enable the attack. This advanced attack, thereby, consists of two separate steps. In the first step the attacker lures the victim onto a prepared Web site and forces the browser to cache the attacker controlled contents. As DNS pinning is active this content is not yet able to launch a DNS Rebinding attack. However, browsers do not persist the domain-to-IP mapping and dispose it eventually. In the second step, at some later point in time, the attacker again lures the victim onto the Web page. This time the content is fetched from cache and therefore no DNS Queries or TCP connections are created. Only the origin information (protocol, domain, port) and the resources are retrieved from cache. When the cached resources attempt to create network connections to its own origin, no domain-to-IP mapping is available and therefore a fresh DNS Query is conducted opening up a vector for DNS rebinding.

Until today, it was difficult to launch such an attack as a browser’s caching behavior is rather unpredictable and the adversary has only limited means to influence which content actually gets cached. The browser cache has a fixed size and in general handles cached content in a first-in-first-out fashion. Given the size of current Web sites, even a moderately used browser’s cache fills up quickly and even recently cached content often gets discarded quickly [11]. Hence, depending on the given circumstances, the chances of keeping the attack script in the cache long enough for a successful attack tend

to be small. This changes with the introduction of the HTML5 Offline Application Cache. Compared to a traditional cache the AppCache provides an attacker with two novel capabilities that make attacks feasible:

- *Controllable caching behavior*: Using the AppCache manifest, the attacker can advise the browser to cache certain resources in a reliable way. As soon as the resources are stored within the AppCache, they reside in the browser for a potentially unlimited amount of time (until the attacker's application or the user decide to empty the cache manually).
- *JavaScript API*: The AppCache provides an API that allows JavaScript to identify whether it was loaded from cache or via the network.

Using these two ingredients, an attacker can conduct reliable DNS Rebinding attacks: In the first step the attacker lures the victim onto his Web site. The Web site uses a manifest file to cache an adversary controlled Web page within the Application Cache. After the browser deleted the DNS Pinning information, the adversary waits until the user visits the same site again. This time the Web page is loaded from the AppCache and no domain-to-IP mapping is available. Using the AppCache's JavaScript API, scripts contained in the page can verify that they indeed have been retrieved without network interaction. Hence, the cached script can now conduct same-origin requests towards the IP returned in the second DNS query (which the attacker controls completely). After the attacker's payload was loaded from cache, the AppCache revalidates the manifest file by downloading it from the attacker's domain. As this domain now points to the victim's IP address, the manifest will not be found and the cache will automatically be deleted (including the evidence for the attack). However, the attack has already taken place. In Section 4.2 we demonstrate, how an attacker is able to avoid the deletion of its content, in case he wants to conduct multiple attacks upon the same victim.

The attack demonstrated in this section only targets one specific victim. Nevertheless, the attack scheme can be extended to conduct large-scale attacks. Instead of conducting a rebinding attack directly on the main domain, the attacker could simply forward each user onto a distinct subdomain that can be rebound separately. As soon as one DNS query arrived at the attacker's DNS server for a specific subdomain, the DNS server could rebind the IP immediately. In the first step the user's browser pins the IP and therefore only sends one initial DNS request. Thus, if a second request arrives, the user's browser must have deleted the pinning information and is in need to refresh the information (opening the DNS Rebinding vector). The only challenge the attacker has

to solve in step 2 is to forward the user to the same sub-domain as utilized for this specific user in step 1. To identify whether the user has already conducted step 1 the attacker could simply utilize cookies that store the subdomain information on the victim's computer until the next visit.

4.2 Utilizing multiple domains for reliable DNS Rebinding attacks

The previously described attack has one major weakness: As explained in Section 2.2, the AppCache revalidates the cache manifest after each access. If the manifest changed, files in the cache will be updated/deleted accordingly. Hence, in the last step of the attack, after the malicious script was fetched from the Application Cache, the browser revalidates the manifest file from the attacker's domain. Since the domain is, at this point in time, bound to the intranet host's IP, the browser requests the manifest file from the intranet host. As the file will typically not be available on the rebound server, the browser deletes the cached content. Nevertheless, the attacker is able to execute the malicious script at least once, as the cache validation takes place after the access to the cache. However, if the attack fails, e.g. because the user closed the browser before the script was executed completely, the attacker has to start the whole process of rebinding from scratch. For large-scale, automated attacks this is not a feasible solution. In order to overcome this issue, a more sophisticated attack scenario can be used. In this scenario, we are able to prevent the deletion of cached content after the rebinding step has taken place by utilizing two distinct domain names. Thereby, we are able to reliably repeat an attack multiple times without the need for rebinding a domain name over and over again. The attack thereby works as follows:

1. An attacker is in control of two domains (*attacker1.org* and *attacker2.org*) and the corresponding DNS server. In order to set up a DNS Rebinding attack, the attacker deploys an HTML document and an offline manifest to *attacker1.org*. The HTML document embeds (via frame, object or embed tags) active content (JavaScript, SVGs, Flash or Silverlight applets, etc) served by *attacker2.org*.
2. The attacker lures a user onto *attacker1.org*. Consequently, the user's browser renders the malicious HTML document and interprets the corresponding manifest file. Due to the instructions contained within the manifest, the browser caches the HTML document as well as the active elements.
3. By closing the browser, the user deletes the DNS pinning information. In the mean time, the attacker rebinds *attacker2.org* to the IP of an intranet host.

4. The attacker again lures the user onto *attacker1.org*. The Web page and the active elements are loaded directly from cache. As the page utilizes embed, frame or object tags for embedding the active elements, these elements are executed within the origin of *attacker2.org*. Due to the fact that *attacker2.org* is bound to the intranet IP, the active content is now able to communicate with intranet applications.

Analysis: In this scenario, as opposed to the first attack, the manifest file resides on a domain that is not subject to rebinding. Hence, when the cache validation takes place, the manifest is still available. Consequently, the browser does not delete the cached content. This is an important fact as it simplifies the attack a lot. If we take, e.g., a corporate wiki containing a multitude of information, the extraction and transfer of the data to the attacker would consume a large amount of time. However, the attacker can only extract the data while the user still visits the malicious Web site. If the user leaves the Web site before all parts of the data were extracted, the attacker is able to again lure the user onto the vulnerable page to continue the extraction process instead of needing to re-iterate the first rebinding step.

4.3 Caching of plug-in content

As mentioned before, the AppCache can be used to store cross-domain resources for offline usage, which is a key enabler for the attack described in the previous section. However, the browser implementations differ in the way they utilize the cache when it comes to cross-domain caching and in the way they defend against rebinding attacks. In this section we shed light on these differences and explain how an attacker can make use of them.

HTML/SVG documents Caching of HTML and SVG documents works across all browsers in the same-domain scenario. However, when it comes to cross-domain caching the behaviors of browsers differ substantially. For the second attack, a distinct document embeds an HTML or SVG file from a second domain via `frame` or `object` tags. The manifest file resides on the first domain, hence referencing the HTML/SVG file across domain boundaries. While WebKit-based browsers (e.g. Safari, Chrome) and Internet Explorer do not fetch such embedded cross-domain resources from cache, Firefox and Opera expose a different behavior: Opera fetches both, content embedded via `frame` and via `object` tags, from the AppCache. Firefox, however, only fetches HTML/SVG documents from cache when they are embedded via `object` tags. Therefore, the advanced attack does not work within Safari or Chrome when utilized in combination with JavaScript. To overcome this issue an attacker can utilize plug-ins such as Flash or Silverlight.

Silverlight All popular desktop browsers except Internet Explorer 10 support the cross-domain caching of Silverlight applets within the offline application cache. This behavior can be abused to conduct DNS Rebinding attacks within these browsers. A Silverlight applet is, similar to JavaScript, able to conduct requests and read the corresponding responses. Hence, the abilities are similar to the HTML/SVG case, but the desktop browser support for the complex attack is better. Mobile browsers, however, are not able to execute Silverlight applets.

In earlier versions of Silverlight, it was possible to also create arbitrary socket connections to same-domain hosts. Fortunately, those capabilities are nowadays severely limited by the underlying security model which only allows opening of a socket connection when the receiving host explicitly grants this connection by setting up a whitelisting policy on port 943. If port 943 is closed, the Silverlight plug-in attempts to download the policy file from the Web server's root directory. Using the HTML5 Offline Cache, an attacker is able to cache such a cross-domain policy at the Web server level. This allows an attacker to open arbitrary socket connections to the rebound IP. As this behavior was already misused in earlier rebinding attacks, Microsoft limited the connection capabilities of Silverlight to a very restricted port range (4502-4534), effectively reducing the impact of such attacks.

Flash Similar to Silverlight, browsers also cache Flash applets within the AppCache. Hence, Flash can be used as an alternative to Silverlight when conducting a DNS Rebinding attack with multiple domains. Thereby, Flash also has the ability to create HTTP requests towards same-origin resources without restrictions. However, Flash has two major advantages over Silverlight:

1. *Widespread adoption:* Although its market share decreases, Flash is still present in about 95% of all browsers [27] (including some mobile browsers).
2. *Less restrictive SOP for HTTP requests:* Flash only includes the protocol and the domain into its cross-domain decision making process [35]. Hence, a Flash applet is able to send requests to any same-domain port and receive the corresponding responses. This behavior can be used to conduct DNS Rebinding attacks on non-HTTP-based intranet services.

Java Java applets do not utilize the browser's AppCache. Instead, Java uses its own caching mechanism that defends against DNS Rebinding by storing the IP address of the host that served the applet. When conducting a HTTP or Socket connection the applet is only allowed to connect back to the same IP address.

Browser	SD	TD SVG	TD F	TD SL
IE 10	✓	-	-	-
Firefox 14.0.1	✓	✓	✓	✓
Chrome 21	✓	-	✓	✓
Safari 5.1	✓	-	✓	✓
Opera 12	✓*	✓*	✓*	✓*

*: Opera prevents access to RFC 1918 addresses.

Table 1: Desktop browser & Attack Overview

Other plug-ins Beside Flash, Silverlight and Java there is a multitude of other plug-ins which can potentially be abused to conduct the presented attacks. If a plug-in applet can be cached within the browser’s Offline Application Cache, it is very likely that it can also be used for the outlined rebinding attacks.

4.4 Summary

As seen in this section, there are a lot of technologies that can be abused by an attacker to gain novel capabilities in the context of a rebinding attack. In order to summarize our findings, Tables 1 and 2 outline which desktop and mobile browsers are vulnerable to the presented attacks.

As seen within the tables, the attack including a single domain (denoted as SD) works within every browser. The attacks comprising two distinct domain names (denoted as TD) affect mainly desktop browsers. The reason for this is the missing plug-in and SVG support within mobile browsers. Furthermore, the mobile versions seem to be more error-prone: The mobile version of Chrome was not able to render our SVG test case (it showed a 404 page, although the server logs indicated that the resource was properly requested), Android’s standard browser even crashed every time it loaded a Flash file from cache.

5 Practical Attacks

To demonstrate the impact of the outlined vulnerabilities, we deployed a real-world setup including three distinct hosts (depicted in Figure 1). In this setup we investigated the susceptibility of two applications (Polipo and CUPS) by conducting the attack described in Sec. 4.2

5.1 Polipo

Our first attack targets a light-weight proxy server called Polipo, which can be used to connect to the TOR anonymizing network. To simplify the handling, Polipo offers a Web interface for configuration purposes. By default, this interface listens to port 8123 and does not defend against DNS Rebinding attacks. Via the Web inter-

Browser	SD	TD SVG	TD F
Mobile Safari	✓	-	<i>n.a.</i>
Android Browser	✓	<i>n.a.</i>	Crash
Mobile Chrome	✓	Error	<i>n.a.</i>
Mobile Firefox	✓	-	✓

Table 2: Mobile browser & Attack Overview

face, a user is able to configure the proxy settings, which are, obviously, security critical.

To evaluate Polipo’s resilience against our DNS Rebinding attack we successfully conducted an attack as described in Section 4.2. Due to the fact that Polipo does not implement any countermeasures against DNS Rebinding, our malicious requests were processed as if the Web application itself created it. Via this attack, we were able to remotely change the settings of the proxy server. Beside the standard proxy functionality, Polipo also offers Web server functionality that can be abused by an attacker to download arbitrary files from the attacked host. The Web server is by default only serving the configuration interface. However, the Web server’s configuration can also be changed via the configuration interface. In order to steal arbitrary files, an adversary could simply set the Web server’s root directory to the server’s root directory (“/” on Unix-based systems), effectively exposing all the files on the host to the outside world. For example, by requesting <http://attacker2.org:8123/etc/passwd> (where attacker2.org is already bound to the internal host) our malicious script was able to extract the information on all the registered user accounts.

5.2 CUPS

CUPS is a printing system for Unix-based operating systems. It offers a web-based administration interface running on port 631 (accessible via localhost only). Via this interface a user can administer the installed printer, monitor print jobs and configure the print server. Interestingly, the main administration panel of CUPS protects against DNS Rebinding attacks by checking the HTTP host header. Some features also require proper authorization, consequently, mitigating the risk of unauthorized access via DNS Rebinding. Nevertheless, it is still possible to extract valuable information out of the administration interface via a DNS Rebinding attack. The reason for this is an insufficient protection of log files that are accessible via the Web interface. While the main administrative functions are protected, the page and error log files can be accessed with arbitrary host headers. This allows an attacker to extract the log files containing sensitive information via DNS Rebinding attacks:

Error log: The error log contains information on failed print jobs, which can be used for reconnaissance of a corporate intranet. When a print job fails, technical

details are written into the logs, including the username of the creator, exact information on the printer addresses and the administrator of the printer. Furthermore, it contains information on the root directory of CUPS as well as the value of the current PATH variable of the machine CUPS is running on.

Page log: The page log gives an overview over the past print jobs sent to a printer. By extracting the page log, the adversary receives the names and dates of the documents that were printed via CUPS. On our test system, running Mac OS, we were able to extract the complete printing history of over one year. Thereby, the name of a document reveals a lot of information such as absence dates of the employee, data on intellectual property, etc.

6 Extending the Same-Origin Policy

As shown in Section 3.1, DNS Rebinding is a constant problem of the Web application paradigm (as witnessed in 1996, 2002, and 2006). Taking the attack method presented in this paper into account, this is the fourth time that wide-scale DNS Rebinding issues are discovered, even though the basic problem is known since 1996 and has received considerable attention. Hence, it is safe to conclude that DNS rebinding is a fundamental, protocol-layer flaw of the Same-Origin Policy, which is not solvable with the existing means. As discussed in Section 3.2, all currently available remedies are either incomplete (e.g., protecting specific IP ranges) and/or have to be implemented explicitly on the server-side’s application layer (e.g., host header checking).

In this section, we show how the Web interaction paradigm can be extended in a non-disruptive manner to enable a robust protection. For this purpose, we first state our design goals (Sec. 6.1) and conduct a root-cause analysis of DNS rebinding (Sec. 6.2). Then, we introduce the “Extended Same-Origin Policy (eSOP)”, starting with simple scenarios (Sec. 6.3) and then iteratively explaining how the policy handles non-trivial cases (Sec. 6.3.1 and Sec. 6.3.2). Finally, after stating the eSOP’s decision logic (Sec. 6.3.3), we show how the policy protects against DNS Rebinding attacks (Sec. 6.3.5).

6.1 Design goals

Before going into detail concerning our solution, we briefly discuss the goals which steered its design process. As stated above, we are not aiming to create band-aid solutions or incomplete protection measures. Instead, the goal is to introduce a fundamental solution that is capable of completely solving DNS Rebinding. In this context, our design goals were as follows:

(DG1) Client-side enforcement: The Same-Origin Policy is a client-side security policy. Hence, all aspects of the policy decision and enforcement process should be conducted in the Web browser.

(DG2) Protocol layer: It should be avoided that Web applications have to explicitly implement protection or decision logic on the server-side’s application layer. Instead, the designed solution should be capable of providing transparent protection by default purely on the protocol layer.

(DG3) Dedicated security functionality: The history and present of the Web is full of cases in which non-security features were (mis)used to realize security functionality. In many cases, the resulting security properties were fragile, often incomplete and not necessarily future proof. Therefore, we do not want to rely on non-security features (i.e., the host header). Instead, dedicated functionality shall be introduced where necessary.

(DG4) Non-disruptive: The solution should be backwards compatible. This means, if a given application scenario involves an entity (i.e., Web server or browser) that does not yet implement the solution, the Web application should not break and the security properties should transparently revert to the currently established state.

6.2 The three principals of Web interaction

As explained in Section 2.1, the Same-Origin Policy’s duty is to isolate unrelated Web servers. To do so, the SOP enforces access control in the browser, based on the “origins” of the corresponding resources. In this context, such origins are derived from the URLs that are associated with the interacting resources - usually the URLs of the enclosing document objects. Hence, the semantics of the SOP are built around two principals: The *browser* for enforcing the policy and the *server(s)* for providing the resources which are the subjects of the policy decision.

However, the entities involved in the implementation of the SOP differ: While the *browser* remains in charge of enforcement, the underlying informations are not provided by the involved Web *server(s)*. Instead, the *network* in the form of Domain Name System and IP addresses is utilized to associate the URL-values to the server resources. Hence, the principal that is central to the SOP’s purpose, the *server*, is not even involved in the actual policy decision. Even worse, security characteristics associated with the *server* are governed by *network* resources that are not necessarily controlled by the server’s owner. As a consequence, a crucial mismatch exists between the semantics and the implementation of the SOP. As seen above, DNS Rebinding takes advantage of this mismatch. In a rebinding scenario, the attacker utilizes *network* resources under his control to undermine the security characteristics of the *server*.

In summary, the Web application model actually spans three principals in total: The *browser*, the *server*, and the *network*. Hence, to address the currently existing mismatch between policy semantics and implementation, it is necessary to investigate approaches that involve the *server* in the policy decision process.

6.3 eSOP: Extending the SOP with explicit server-origin

When considering the SOP from an abstract point of view, a Web “origin” defines the trust boundaries of a Web application. Everything within the application’s origin is fully trusted, everything outside is completely distrusted. Additional browser capabilities, such as domain relaxation (see Sec. 2.1) and CORS [37], provide methods to selectively widen the application’s trust boundaries. In the last section, we observed that the Web server itself is left out of the equation in the SOP’s current implementation. This is counterintuitive, as among the involved parties, it is the Web server that should be able to set its own trust boundaries. However, the Web server can only indirectly influence the browser’s enforcement decisions. Hence, to resolve this shortcoming, we propose to extend the SOP to include Web server-provided input. For this purpose, our approach expands the current, triple-based SOP with a fourth component that is provided by the server. Simplified, our proposed extended Same-Origin Policy (eSOP) works as follows: All HTTP responses of a given server carry explicit, server-provided information of the server’s trust boundaries. From now on, we refer to this information as the *server-origin*. Thus, in the extended model, a Web origin consists of the quadruple $\{\text{protocol}, \text{domain}, \text{port}, \text{server-origin}\}$. In consequence, whenever the browser conducts an eSOP check, not only the classic protocol/domain/port triple has to match, but also the *server-origin* values.

Example 1 (standard behavior): For simple cases, a Web origin’s *domain* and *server-origin* values should not differ. Take for instance a script running under the origin $\{\text{http}, \text{example.org}, 80, \text{example.org}\}$. This script attempts to access a document in an iframe which also has the origin $\{\text{http}, \text{example.org}, 80, \text{example.org}\}$. All four elements of the respective Web origins match, thus, the eSOP is satisfied and the access is granted.

6.3.1 Multiple domains as server-origin

However, last section’s simplified policy decision logic is not sufficient to cover all application scenarios, that are allowed with the current SOP. This primarily concerns Web applications which can be accessed via multiple domain names. For instance, many Web applications do not distinguish between the main domain

name (e.g., example.org) and its “www” counterpart (i.e., www.example.org). Similar scenarios exist for applications accepting requests for multiple top-level domains (e.g., example.com and example.net). Hence, for resources served by such applications, it is not straight forward to decide what their corresponding *server-origin* is. As stated in design goal 6.1, our solution shall not require the implementation of application-layer decision logic on the server-side. In consequence, a solution is needed which allows server-side configuration on the protocol-layer. For this reason, the eSOP permits that the server specifies more than one domain value as its *server-origin*. This way, the *server-origin* precisely specifies a server’s trust boundaries, i.e, the set of domains which it grants access in a same-origin context. Furthermore, we adjust the criteria under which two Web origin quadruples comply to the eSOP: The eSOP is satisfied if and only if the classic *protocol/domain/port* values of both quadruples match and the domain value of the acting origin (i.e., the origin of the script) is included in the *server-origin* of the resource which the script tries to access.

Example 2 (multiple server-origins): A Web application available via example.org and www.example.org specifies its *server-origin* as a tuple of both domains: $\langle \text{example.org}, \text{www.example.org} \rangle$. A script running in a document under the origin $\{\text{http}, \text{example.org}, 80, \langle \text{example.org}, \text{www.example.org} \rangle\}$ tries to access a document in a iframe which also has the origin $\{\text{http}, \text{example.org}, 80, \langle \text{example.org}, \text{www.example.org} \rangle\}$. As the script’s domain value (example.org) is included in the target document’s *server-origin* list $\langle \text{example.org}, \text{www.example.org} \rangle$, the eSOP is satisfied and, thus, the access is granted.

6.3.2 Handling domain relaxation

The specific matching criterion for *server-origin* also allows simple and robust handling of domain relaxation via setting the document .domain property during client-side execution: As long as the newly set origin is still in the target resource’s list of domains, the eSOP allows access under the relaxed domain values. This even works in situations in which the individual subdomains are handled by separate Web servers with potentially different *server-origin* configurations.

Example 3 (domain relaxation): Take a Web application on example.org, which has multiple subdomains, including sub.example.org. The application’s subdomains are handled by dedicated Web servers. Furthermore, the example.org server hosts all resources that are shared among the subdomains. A script is executed under the extended Web origin $\{\text{http}, \text{sub.example.org}, 80, \langle \text{sub.example.org} \rangle\}$. Furthermore, the browser provides a reference to a resource from

the main application with the origin $\{http, example.org, 80, \langle example.org \rangle\}$. The script assigns the value *example.org* to the `document.domain` property, thus, effectively relaxes its domain value to the fathering domain. As a result, the script's effective origin is now $\{http, example.org, 80, \langle sub.example.org \rangle\}$. Consequently, the eSOP is now satisfied in respect to the referenced resource, as the script's domain value is included in the domain set of the resource's *server-origin*, and the access is granted.

6.3.3 The eSOP decision logic

To sum up, we now give a precise definition of the eSOP.

The eSOP is satisfied iff:

$$\begin{aligned} \{prot1, domain1, port1\} &== \{prot2, domain2, port2\} \\ \text{and} \\ domain1 &\in server\text{-}origin2 \end{aligned}$$

If the *server-origin2* property is empty, the second criterion always evaluates as “true”.

The last condition of the eSOP provides robustness and backwards compatibility with the old behavior. In addition, to facilitate flexible and easy configuration, we follow the example of the Content-Security Policy format [33], and allow the usage of wildcards for subdomain values within the set of domains in the *server-origin*, e.g., $\{\ast.\text{domain.com}\}$.

6.3.4 Communicating the server-origin

The final missing puzzle piece is the exact method, how the server communicates the *server-origin* property of his resources to the browser. We propose to introduce a dedicated HTTP response header, `X-Server-Origin`, that carries the *server-origin* property in the form of a comma-separated list.

Choosing this approach has several advantages: Foremost, it is compatible with the caching behavior of Web browsers. Web browsers are already required to cache HTTP response headers along with the actual resources, as they otherwise would not be able to properly interpret the cached content after retrieving it from storage. Also, unlike DNS or IP-based protection schemes, properties communicated via HTTP response headers are preserved when the browser accesses the network via a Web proxy. Finally, adding features using new response headers is non-disruptive, as older browsers simply ignore unknown response headers. Furthermore, implementing server-driven security functionality via

HTTP response headers is a proven technique. In the recent past, several security measures have successfully been introduced, that leverage response headers, such as Clickjacking protection via the `X-Frame-Options` header [23], protection against SSL-stripping attacks via the `Strict-Transport-Security` header [12], Content Security Policies, that are set using the `X-Content-Security-Policy` or `X-WebKit-CSP` headers [33], and cross-origin resource sharing which utilizes the `Allow-From`-header [37].

6.3.5 The eSOP and DNS Rebinding

In the previous sections, we discussed the semantics of the eSOP and the reasoning behind the corresponding design process. Now finally, we show that the eSOP is indeed capable of prevention DNS Rebinding attacks. To conduct a DNS Rebinding attack, the adversary maps the DNS setting of a *domain* to the IP address of the targeted Web server. However, the attacker controlled *domain* value is not in the Web server's trust boundary. In consequence, the value will not be included in the list of domain values in the server's *server origin* property. Therefore, the eSOP check will necessarily fail.

Example 4 (DNS Rebinding): The attacker controls the domain *attacker.org*. His goal is to access an internal wiki server under the domain *wiki.corp*, which sets a corresponding *server-origin*. In the first step of his attack, the adversary tricks the victim to access the *attacker.org*, which still is mapped to a Web server IP under his control. Hence, the script is handled by the browser under a Web origin of the form $\{http, attacker.org, 80, \dots\}$. Please note, that this Web origin's *server-origin* property is fully controlled by the attackers, as he creates the corresponding HTTP response. However, this does not cause any issues, as the *server-origin* of the acting script is irrelevant for the eSOP decision process. Then, the attacker conducts the DNS Rebinding step. Now, the DNS entry of *attacker.org* points to the IP address of the internal server. From this point on, the browser will interpret all resources from the server under the Web origin $\{http, attacker.org, 80, \langle wiki.corp \rangle\}$. Following the rebinding step, the attacker's script attempts to access Web resources that are provided by the internal server. However, as the attacker's script carries the *domain* property *attacker.org*, which is not included in the list of domains in the server's *server-origin*, the attack fails, even though the classic *protocol/domain/port* SOP is satisfied.

6.3.6 Invalid eSOP origins

In [13], Jackson and Barth examine a set of proposed SOP variants with finer-grained origins. Among other

techniques they discuss two approaches closely related to the eSOP: The Locked SOP and IP-based origins (for details on these techniques please refer to Sec. 8), which provide basic protection against DNS Rebinding attacks. For both techniques they uncover a loophole which re-enables DNS Rebinding attacks, even if the refined SOP variant is in place: Take a Web page on an internal host which intends to import a JavaScript file from the same host using a relative URL (see Lst. 3).

Listing 3: Direct script include using a relative URL

```
1 <script src="jquery.js"></script>
```

This Web page is retrieved by the browser using the adversary controlled hostname *attacker.org*, which resolves to the intranet IP 10.10.10.10. Then, before the `script` tag is interpreted the rebinding step takes place. *Attacker.org* now points to 6.6.6.6 which is owned by the adversary. Unlike JavaScript execution, HTML-based script includes are not subject to origin restrictions. Hence, a refined SOP has no direct effect here and the script code is retrieved from the adversary’s host, circumventing the protection of the refined policy. Fortunately, in the case of the eSOP such situations are reliably detectable. The following condition holds for all HTML documents with origin $\{prot, domain, port, server-origin\}$ that were retrieved from an attacked host:

$$domain \notin server-origin$$

This necessarily results from the fact that the adversary cannot control the *server-origin* of the internal host, which only contains domain values within the server’s trust boundaries (which obviously excludes the adversary’s sites). In such cases, we label the page’s Web origin as *invalid*. For Web documents with an *invalid origin* caching is disabled and strict DNS pinning is enforced for the whole browser session, effectively closing the loophole.

6.4 Security evaluation

As shown above the eSOP protects against DNS Rebinding attacks, without requiring additional server-side logic or specific actions on the client-side. As soon as the `X-Server-Origin` header is present, the browser is capable of transparently enforcing the policy, fulfilling design goals (DG1) and (DG2). Furthermore, due to communicating the *server-origin* in the form of an HTTP response header, the protection is robust in scenarios which caused other countermeasures to fail: HTTP response headers are cached alongside with the actual cached resources. Hence, the *server-origin* is maintained even in long-term caching scenarios, effectively closing the attack vector which is the subject of Section 4. In addition, currently problematic scenarios, in which the

browser has no control over the domain-to-IP mapping, e.g., through a Web proxy, can be handled conveniently. The `X-Server-Origin` header is preserved, even if Web proxies obstruct the link between domain name and server address. Hence, the attack scenario described in [34] (see also Sec. 3.1) is not feasible anymore. Finally, the eSOP is at least as strong as the currently implemented SOP: The *protocol/domain/port*-triple is still required to match, as it is by the classic SOP. Thus, it is a necessary condition that the access to a resource is granted under the SOP for the eSOP to be satisfied. Therefore, implementing the eSOP will never lead to security degradation.

6.5 Functional evaluation

The eSOP is fully backwards compatible to the classic SOP. In cases that either the browser does not implement the extended policy or the Web server does not provide a `X-Server-Origin` header, the enforced policy transparently reverts back to the standard behavior of matching *protocol/domain/port*, fulfilling design goal (DG4).

A major concern during designing the extended policy was the aspect of maintainability: Especially in large set-ups that span multiple Web servers, ensuring that all server installations provide the exact same values for the *server-origin* property, is an unrealistic hard requirement. Fortunately, the eSOP’s specific *server-origin* matching criterion (see Sec. 6.3.3) allows a robust and flexible handling of such situations. The eSOP does not require the *server-origin* values to match exactly. The only requirement is, that the acting domain is whitelisted in the receiving *server-origin*. Hence, even in situations of slightly different server configurations (much like in Example 3, Sec. 6.3.2), the functionality of the Web application remains undisturbed. Additionally, this robustness property also allows *server-origin* settings to change in long term caching scenarios. As long as the initial domain requirements of the cached resource remain fulfilled, the server’s *server-origin* setting can be extended or modified without causing interoperability problems.

Last but not least, an adaption of the eSOP would obliterate the requirement of DNS Pinning for security reasons completely. Hence, for servers that provide the `X-Server-Origin` header, the DNS TTL value can be as small as desired. No security degradation will occur, when browsers respect such small TTL values. This in turn allows easy setup of highly flexible load-balancing and error-correcting network setups with multiple, redundant servers.

7 Practical Implementation

In order to validate the feasibility, security and functionality properties of the eSOP, we implemented it for the Chromium Web browser [6]. Thereby, we enhanced the so-called Security-Origin which stores the "protocol, domain and port"-triple of a Web site by adding the proposed *Server-Origin*. Data stored within this data structure is provided by the X-Server-Origin response header. Our implementation allows the header to have two types of values. If the server does not send the header or sends an empty header, we assume that it does not implement our approach or wants to opt-out of the protection mechanism. In these cases, we allow access regardless of the acting *domain* value for backwards compatibility. Additionally, the header can be set to a list of comma-separated domains. Using the stored information we are able to successfully prevent rebinding scenarios. At this point, we need to distinguish between XMLHttpRequests (XHRs) and script access to a viewport, such as frames or popup windows.

Script access to a viewport For a viewport, we want to align our implementation to how browsers should handle cross-origin requests, thus allowing a popup or frame from any resource to be rendered but to deny script access if the origins do not match. This is also important towards keeping design goal (DG4), i.e., being downwards compatible. In the current implementation of Chromium we extended the origin check to verify the server-origin as well as the protocol, domain and port. If a Web application does not implement our suggested extended same-origin policy, the browser falls back to the normal SOP validation and renders the page properly.

XmlHttpRequests For XMLHttpRequests, we patched the functionality for same-origin requests to parse our response header field and to grant or revoke scripting access depending on the received value.

To be fully interoperable with the browser's XHR object, we had to ensure compatibility with its recently introduced cross-origin capabilities:

To allow XHRs to access cross-origin resources, the W3C specified cross-origin resource sharing (CORS) [37]. CORS allows the initiation of *simple* requests to a cross-origin resource and only checks the right to access the response after the request has been completed. In the context of CORS a request is considered to be *simple* if it also would be possible to create an equivalent request with other means, such as *IMG*-tags or HTML forms. because simple requests cannot change the state of a web application.

For *complex* requests, CORS requires that the browser sends a preflight request to the server to retrieve the

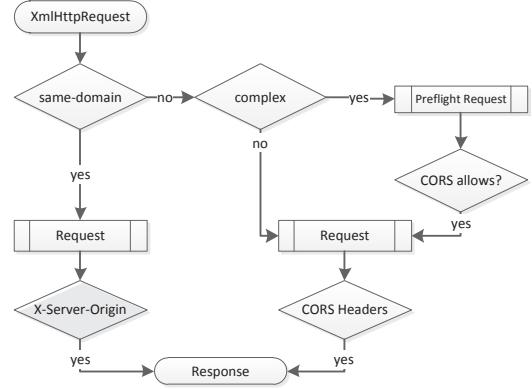


Figure 2: Implementation logic for XHR

CORS-relevant headers. Only if the retrieved headers allow access to the resource, the complex request is sent to the server to ensure that state-changing operations are only performed if explicitly allowed by the application.

In a sense, requests to rebound domain should also be treated as cross-origin requests. Thus, we can allow simple requests to be sent but need to verify the server-origin before allowing access to the response. For a complex request, we need to check the preflight response and only allow the actual request to be sent if the server-origin matches. To distinguish between simple and complex, we used the already existing check from the CORS implementation in Chromium. However, using the preflight functionality from CORS would break constraint DG3. If - for example - we request a same-domain resource on a server that does not implement CORS, the CORS headers would not be set and the check would fail. Therefore, we implemented a function that only checks the X-Server-Origin header.

The flow chart in Fig. 2 shows the resulting implementation logic of the XHR object. Our addition to the implementation is positioned on the lower left of the chart, whereas the right part of the figure depicts the original logic as implemented by Chromium. Note that as an XHR is not rendered by the browser, we can directly block access upon receiving the response from the server.

7.1 Implementation and performance

In total, we modified 34 lines of code in Chromium. As discussed earlier, the implementation manifests itself only as parsing and extraction of the HTTP headers, the allocation of a little amount of memory to store the server-origin and a string comparison of the domain and the stored value. The parsing of HTTP headers is executed for any request, thus the performance impact is

reduced to just one more array access. Thus, in our tests we had no noticeable overhead when accessing a Web application.

8 Related Work

Related offensive and protective techniques have already been the subject of Sections 3.1 and 3.2. Hence, in this section we focus on approaches that directly relate to the eSOP, as they propose modifications to the browser-server interaction to combat DNS Rebinding:

Conceptually closest to our protection approach is the “Strong Locked SOP” by Karlof et al. [19], which also proposes to include server-provided information into the SOP decision. In the case of the “Strong Locked SOP”, this information is derived from the TLS/SSL certificates of the involved Web servers in the form of the certificates’ public keys. Consequently, JavaScript is only granted access to resources that share the same public key. In the special case of “pharming” attacks (which is the approach’s main concern), where the attacker controls the DNS resolving process of the victim, Karlof’s approach is conceptually stronger than the eSOP. Furthermore, in a scenario in which all communication is done via HTTPS and all servers are outfitted with valid SSL certificates, the Strong Locked SOP would provide reliable protection against rebinding attacks. However, expecting the Web to go completely HTTPS appears unrealistic, especially regarding intranet Web resources which only in very rare cases have valid SSL certificates. In contrast, the eSOP only requires to configure a single response header and works well in plain HTTP scenarios.

In [13] it is mentioned that early versions of the HTML5 specification included “IP-based Origins”, which utilize the server’s IP as a fourth factor in the origin check. Compared to the eSOP, IP-based Origins are neither able to securely handle domain relaxation nor do they provide evidence of invalid origins (see Sec. 6.3.6), thus, making them susceptible to library include attacks.

Furthermore, Jackson et al. propose “Host Name Authorization”, a network based service [14], which announces the host names that are associated with a given IP address. Host Name Authorization relies on reverse DNS: Whenever the browser executes a DNS lookup, it also verifies that the requested domain is actually in the set of valid domains of the received IP address. This is done via querying the service under `auth.ip.in-addr.arpa`, with `ip` being the IP address which has been returned by the DNS server. Compared to our approach, Host Name Authorization has several drawbacks. For one, it requires considerable setup effort, as both reverse DNS as well as the actual service have to be enabled. Also, Host Name Authorization is realized within the DNS system, hence, the maintainer of the Web server

also needs administrative access to the corresponding DNS server. This requirement cannot always be satisfied, e.g., in shared hosting scenarios, for local machines, or for internal services in cooperate networks. In addition, the approach requires two additional DNS round trips for each DNS resolving process, which could lead to noticeable latency under certain circumstance, e.g., cellular networks. In comparison, our approach only requires Web server-provided functionality and does not add any network overhead.

Finally, for completeness sake, the Internet draft [22] proposes the HTTP request header `X-Request-Origin`. The purpose of the header is to transport the domain value or IP address of the browser-based component which was responsible for initiating the HTTP request within the browser. The draft lists DNS Rebinding attacks (in the form of “Quick-swap DNS”) as one of its motivational examples. However, in the context of DNS Rebinding situations, the header’s value will necessarily always equal the value of the HTTP `host` header, and hence, shares its protection properties and drawbacks.

9 Conclusion

For more than one and a half decades, DNS Rebinding continued to be a constant problem of the Web. Several attempts to mitigate the issue have been undertaken, but up to now no fundamental solution for the problem was introduced successfully. In this paper, we presented a novel attack variant, utilizing the HTML5 AppCache. We practically validated our attack and demonstrated that it affects all popular browsers and most plug-in technologies, while reliably circumventing currently existing browser-based countermeasures. Using our attack as motivation, we revisited the attack’s underlying problem and identified a mismatch between the SOP’s semantics and its implementation: The SOP’s main purpose is to ensure security boundaries of Web servers. However, the Web servers themselves are only indirectly involved in the security decision. Instead, the SOP relies on information obtained from the domain name system, which is not necessarily controlled by the Web server’s owners. This mismatch is exploited by DNS Rebinding.

To overcome this problematic inconsistency, we proposed a light-weight extension to the SOP (eSOP), which takes input from the Web server into account. The eSOP robustly defeats DNS Rebinding attacks while being backward compatible with user-agents that do not yet implement the extended policy. Our solution does not require additional network traffic and fully supports previously problematic scenarios, including domain relaxation, content caching, and communication over Web proxies. Additionally, the eSOP eradicates the need for DNS Pinning. Thus, browsers implementing the pol-

icy can better inter-operate with dynamic DNS settings, such as DNS based load-balancing or Content Distribution Networks (CDNs). In summary, adopting the eSOP comes with very little costs but leads to a significant security increase and additional benefits in functionality.

Acknowledgments

This work was in parts supported by the EU Project Web-Sand (FP7-256964), <http://www.websand.eu>. The support is gratefully acknowledged.

References

- [1] B. Anderson. Why Web Browser DNS Caching Can Be A Bad Thing. [online], <http://dyn.com/web-browser-dns-caching-bad-thing/>, last accessed 08/06/2012, 2011.
- [2] A. Bortz, A. Barth, and C. Jackson. Dnswall. [software], <http://code.google.com/p/google-dnswall/>.
- [3] D. Byrne. Anti-DNS Pinning and Java Applets. Posting to the Bugtraq mailing list, <http://seclists.org/fulldisclosure/2007/Jul/0159.html>, July 2007.
- [4] D. Dean, E. Felten, and D. Wallach. Java Security: From Hot Java to Netscape and Beyond. In *Proceedings of the 1996 IEEE Symposium on Security and Privacy*, SP '96, pages 190–, Washington, DC, USA, 1996. IEEE Computer Society.
- [5] S. Dutta. Client-side cross-domain security. Technical report, Microsoft, Dec. 2011. <http://msdn.microsoft.com/en-us/library/cc709423%28v=vs.85%29.aspx>.
- [6] Google Chromium Developers. The Chromium projects. [online] <http://www.chromium.org>.
- [7] J. Grossman, R. Hansen, P. Petkov, and A. Rager. *Cross Site Scripting Attacks: XSS Exploits and Defense*. Syngress, 2007.
- [8] J. Grossman and T. Niedzialkowski. Hacking Intranet Websites from the Outside. Talk at Black Hat USA, <http://www.blackhat.com/presentations/bh-usa-06/BH-US-06-Grossman.pdf>, 2006.
- [9] C. Heffner. How to Hack Millions of Routers. Talk at the Black Hat USA conference, 2010.
- [10] I. Hickson. HTML5. W3C working draft, W3C, May 2012. <http://www.w3.org/TR/html5/>.
- [11] J. Hirth. It's Time to Rethink the Default Cache Size of Web Browsers. [online], <http://kaioa.com/node/74>, last access 8/5/2012, 2008.
- [12] J. Hodges, C. Jackson, and A. Barth. HTTP Strict Transport Security (HSTS). [IETF draft], <http://tools.ietf.org/html/draft-ietf-websec-strict-transport-sec>, Version 11, July 2012.
- [13] C. Jackson and A. Barth. Beware of Finer-Grained Origins. In *In Web 2.0 Security and Privacy (W2SP 2008)*, 2008.
- [14] C. Jackson, A. Barth, A. Bortz, W. Shao, and D. Boneh. Protecting Browsers from DNS Rebinding Attacks. In *In Proceedings of ACM CCS 07*, 2007.
- [15] M. Johns. (somewhat) breaking the same-origin policy by undermining dns-pinning. Posting to the Bugtraq mailinglist, <http://www.securityfocus.com/archive/107/443429/30/180/threaded>, 2006.
- [16] M. Johns and Kanatoko. Using Java in anti DNS-pinning attacks (Firefox and Opera). [online], Security Advisory, <http://shampoo.antville.org/stories/1566124/>, (08/27/07), Februar 2007.
- [17] M. Johns and J. Winter. Protecting the Intranet Against "JavaScript Malware" and Related Attacks. In *Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA 2007)*. Springer, July 2007.
- [18] Kanatoko. Anti-DNS Pinning + Socket in Flash. [online], <http://www.jumperz.net/index.php?i=2&a=3&b=3>, (19/01/07), January 2007.
- [19] C. Karlof, U. Shankar, J. Tygar, and D. Wagner. Dynamic pharming attacks and the locked same-origin policies for web browsers. In *Proceedings of the 14th ACM Conference on Computer and Communication Security (CCS '07)*, October 2007.
- [20] G. Maone. NoScript Firefox Extension. [software], <http://www.noscript.net/whats>, 2012.
- [21] A. Megacz. Firewall circumvention possible with all browsers. Posting to the Bugtraq mailinglist, <http://seclists.org/bugtraq/2002/Jul/0362.html>, July 2002.
- [22] A. Megacz and D. Meketa. X-RequestOrigin. Internet Draft, <http://tools.ietf.org/html/draft-megacz-x-requestorigin-00>, June 2003.
- [23] Microsoft. IE8 Security Part VII: ClickJacking Defenses, 2009.
- [24] M. Mueller. Response to DNS spoofing attack. [Usenet posting], <http://sip.cs.princeton.edu/news/sun-02-2-2-96.html>, 1996.
- [25] Princeton University. DNS Attack Scenario. [online], <http://www.cs.princeton.edu/sip/news/dns-scenario.html>.
- [26] Y. Rekhter, B. Moskowitz, D. Karrenberg, G. J. de Groot, and E. Lear. Address Allocation for Private Internets. RFC 1918, <http://www.ietf.org/rfc/rfc1918.txt>, February 1996.
- [27] Rich internet application (ria) market share. http://www.statowl.com/custom_ria_market_penetration.php.
- [28] B. K. Rios and N. McFeters. Slipping Past The Firewall. Talk at the HITBSecConf2007 conference, <http://conference.hitb.org/hitbsecconf2007kl/agenda.htm>, 2007.
- [29] J. Roskind. Attacks Against the Netscape Browser. Talk at the RSA Conference, April 2001.
- [30] D. Ross. Notes on DNS Pinning. [online], <http://blogs.msdn.com/b/dross/archive/2007/07/09/notes-on-dns-pinning.aspx>, last accessed 8/4/12, July 2007.
- [31] J. Ruderman. The Same Origin Policy. [online], <http://www.mozilla.org/projects/security/components/same-origin.html> (01/10/06), August 2001.

- [32] J. Soref. DNS: Spoofing and Pinning. [online], <http://web.archive.org/web/20100211170613/http://viper.haque.net/~timeless/blog/11/>, (07/07/12), 2003.
- [33] B. Sterne and A. Barth. Content Security Policy. W3C Working Draft, <http://www.w3.org/TR/2011/WD-CSP-20111129/>, 2012.
- [34] D. Stuttard. DNS Pinning and Web Proxies. NISR whitepaper, <http://www.ngssoftware.com/research/papers/DnsPinningAndWebProxies.pdf>, 2007.
- [35] P. Uhley. Flash content and the same-origin policy. http://blogs.adobe.com/asset/2009/11/flash_content_and_the_same_ori.html, 2009.
- [36] D. Ulevitch. Finally, a real solution to DNS rebinding attacks. [online], <http://blog.opendns.com/2008/04/14/finally-a-real-solution-to-dns-rebinding-attacks/>, last accessed 08/06/2012, April 2008.
- [37] A. van Kesteren (Editor). Cross-Origin Resource Sharing. W3C Working Draft, Version WD-cors-20100727, <http://www.w3.org/TR/cors/>, July 2010.
- [38] W3C. Same Origin Policy. [online], http://www.w3.org/Security/wiki/Same_Origin_Policy, (08/01/2012, 2010).

Revolver: An Automated Approach to the Detection of Evasive Web-based Malware

Alexandros Kapravelos
UC Santa Barbara
kapravel@cs.ucsb.edu

Yan Shoshitaishvili
UC Santa Barbara
yans@cs.ucsb.edu

Marco Cova
University of Birmingham
m.cova@cs.bham.ac.uk

Christopher Kruegel
UC Santa Barbara
chris@cs.ucsb.edu

Giovanni Vigna
UC Santa Barbara
vigna@cs.ucsb.edu

Abstract

In recent years, attacks targeting web browsers and their plugins have become a prevalent threat. Attackers deploy web pages that contain exploit code, typically written in HTML and JavaScript, and use them to compromise unsuspecting victims. Initially, static techniques, such as signature-based detection, were adequate to identify such attacks. The response from the attackers was to heavily obfuscate the attack code, rendering static techniques insufficient. This led to dynamic analysis systems that execute the JavaScript code included in web pages in order to expose malicious behavior. However, today we are facing a new reaction from the attackers: *evasions*. The latest attacks found in the wild incorporate code that detects the presence of dynamic analysis systems and try to avoid analysis and/or detection.

In this paper, we present *Revolver*, a novel approach to automatically detect evasive behavior in malicious JavaScript. *Revolver* uses efficient techniques to identify similarities between a large number of JavaScript programs (despite their use of obfuscation techniques, such as packing, polymorphism, and dynamic code generation), and to automatically interpret their differences to detect evasions. More precisely, *Revolver* leverages the observation that two scripts that are similar should be classified in the same way by web malware detectors (either both scripts are malicious or both scripts are benign); differences in the classification may indicate that one of the two scripts contains code designed to evade a detector tool.

Using large-scale experiments, we show that *Revolver* is effective at automatically detecting evasion attempts in JavaScript, and its integration with existing web malware analysis systems can support the continuous improvement of detection techniques.

1 Introduction

In the last several years, we have seen web-based malware—malware distributed over the web, exploiting vulnerabilities

in web browsers and their plugins—becoming a prevalent threat. Microsoft reports that it detected web-based exploits against over 3.5 million distinct computers in the first quarter of 2012 alone [22]. In particular, drive-by-download attacks are the method of choice for attackers to compromise and take control of victim machines [12, 29]. At the core of these attacks are pieces of malicious HTML and JavaScript code that launch browser exploits.

Recently, a number of techniques have been proposed to detect the code used in drive-by-download attacks. A common approach is the use of honeyclients (specially instrumented browsers) that visit a suspect page and extract a number of features that help in determining if a page is benign or malicious. Such features can be based on static characteristics of the examined code [5, 7], on specifics of its dynamic behavior [6, 20, 25, 28, 32, 40], or on a combination of static and dynamic features [34].

Drive-by downloads initially contained only the code that exploits the browser. This approach was defeated by static detection of the malicious code using signatures. The attackers started to obfuscate the code in order to make the attacks impossible to be matched by signatures. Obfuscated code needs to be executed by a JavaScript engine to truly reveal the final code that performs the attack. This is why researchers moved to dynamic analysis systems which execute the JavaScript code, deobfuscating this way the attacks regardless of the targeted vulnerable browser or plugin. As a result, the attackers have introduced evasions: JavaScript code that detects the presence of the monitoring system and behaves differently at runtime. Any diversion from the original targeted vulnerable browser (e.g., missing functionality, additional objects, etc.) can be used as an evasion.

As a result, malicious code is not a static artifact that, after being created, is reused without changes. To the contrary, attackers have strong motivations to modify the code they use so that it is more likely to evade the defense mechanisms employed by end-users and security researchers, while continuing to be successful at exploiting vulnerable browsers. For example, attackers may obfuscate their code so that it

does not match the string signatures used by antivirus tools (a situation similar to the polymorphic techniques used in binary malware). Attackers may also mutate their code with the intent of evading a specific detection tool, such as one of the honeyclients mentioned above.

This paper proposes *Revolver*, a novel approach to automatically identify evasions in drive-by-download attacks. In particular, given a piece of JavaScript code, *Revolver* efficiently identifies scripts that are *similar* to that code, and automatically classifies the differences between two scripts that have been determined to be similar. *Revolver* first identifies syntactic-level differences in similar scripts (e.g., insertion, removal, or substitution of snippets of code). Then *Revolver* attempts to explain the semantics of such differences (i.e., their effect on page execution). We show that these changes often correspond to the introduction of evasive behavior (i.e., functionality designed to evade popular honeyclient tools).

There are several challenges that *Revolver* needs to address to make this approach feasible in practice. First, typical drive-by-download web pages serve malicious code that is heavily obfuscated. The code may be mutated from one visit to the page to the next by using simple polymorphic techniques, e.g., by randomly renaming variables and functions names. Polymorphism creates a multitude of differences in two pieces of code. From a superficial analysis, two functionally identical pieces of code will appear as very different. In addition, malicious code may be produced on-the-fly, by dynamically generating and executing new code (through JavaScript and browser DOM constructs such as the `eval()` and `setTimeout()` functions). Dynamic code generation poses a problem of coverage; that is, not all JavaScript code may be readily available to the analyzer. Therefore, a naive approach that attempts to directly compare two malicious scripts would be easily thwarted by these obfuscation techniques and would fail to detect their similarities. Instead, *Revolver* dynamically monitors the execution of JavaScript code in a web page so that it can analyze both the scripts that are statically present in the page and those that are generated at runtime. In addition, to overcome polymorphic mutations of code, *Revolver* performs its similarity matching by analyzing the Abstract Syntax Tree (AST) of code, thereby ignoring superficial changes to its source code.

Another challenge that *Revolver* must address is scalability. For a typical analysis of a web page, *Revolver* needs to compare several JavaScript scripts (more precisely, their ASTs) with a repository of millions of ASTs (potential matches) to identify similar ones. To make this similarity matching computationally efficient, we use a number of machine learning techniques, such as dimensionality reduction and clustering algorithms.

Finally, not all code changes are security-relevant. For example, a change in a portion of the code that is never executed is less interesting than one that causes a difference in

the runtime behavior of the script. In particular, we are interested in identifying code changes that cause detection tools to misclassify a malicious script as benign. To identify such evasive code changes, *Revolver* focuses on modifications that introduce control flow changes in the program. These changes may indicate that the modified program checks whether it is being analyzed by a detector tool (rather than an unsuspecting visitor) and exhibits a different behavior depending on the result of this check.

By automatically identifying code changes designed to evade drive-by-download detectors, one can improve detection tools and increase their detection rate. We also leverage *Revolver* to identify benign scripts (e.g., well-known libraries) that have been injected with malicious code, and, thus, display malicious behavior.

This paper makes the following contributions:

1. **Code similarity detection:** We introduce techniques to efficiently identify JavaScript code snippets that are similar to each other. Our tool is resilient to obfuscation techniques, such as polymorphism and dynamic code generation, and also pinpoints the precise differences (changes in their ASTs) between two different versions of similar scripts.
2. **Detection of evasive code:** We present several techniques to automatically classify differences between two similar scripts to highlight their purpose and effect on the executed code. In particular, *Revolver* has identified several techniques that attackers use to evade existing detection tools by continuously running in parallel with a honeyclient.

2 Background and Overview

To give the reader a better understanding of the motivation for our system and the problems that it addresses, we start with a discussion of malicious JavaScript code used in drive-by-download attacks. Moreover, we present an example of the kind of code similarities that we found in the wild.

Malicious JavaScript code. The web pages involved in drive-by-download attacks typically include malicious JavaScript code. This code is usually obfuscated, and it fingerprints the visitor’s browser, identifies vulnerabilities in the browser itself or the plugins that the browser uses, and finally launches one or more exploits. These attacks target memory corruption vulnerabilities or insecure APIs that, if successfully exploited, enable the attackers to execute arbitrary code of their choice.

Figure 1 shows a portion of the code used in a recent drive-by-download attack against users of the Internet Explorer browser. The code (slightly edited for the sake of clarity) instantiates a shellcode (Line 8) by concatenating the variables defined at Lines 1–7; a later portion of the code (not shown in the figure) triggers a memory corruption

```

1 var nop="%uyt9yt2yt9yt2";
2 var nop=(nop.replace(/yt/g,""));
3 var sc0="%ud5db%uc9c9%u87cd...";
4 var scl=%"+yutianu"+ByutianD+ ...;
5 var scl=(scl.replace(/yutian/g,""));
6 var sc2=%"+u"+54"+FF"+...+"8"+E"+E";
7 var sc2=(sc2.replace(/yutian/g,""));
8 var sc=unescape(nop+sc0+scl+sc2);

```

Figure 1: Malicious code that sets up a shellcode.

vulnerability, which, if successful, causes the shellcode to be executed.

A common approach to detect such attacks is to use honeyclients, which are tools that pose as regular browsers, but are able to analyze the code included in the page and the side-effects of its execution. More precisely, low-interaction honeyclients emulate regular browsers and use various heuristics to identify malicious behavior during the visit of a web page [6, 13, 25]. High-interaction honeyclients consist of full-featured web browsers running in a monitoring environment that tracks all modifications to the underlying system, such as files created and processes launched [28, 38, 40]. If any unexpected modification occurs, it is considered to be a manifestation of a successful exploit. Notice that this sample is difficult to detect with a signature, as strings are randomized on each visit to the compromised site.

Evasive code. Attackers have a vested interest in crafting their code to evade the detection of analysis tools, while remaining effective at exploiting regular users. This allows their pages to stay “under the radar” (and actively malicious) for a longer period of time, by avoiding being included in blacklists such as Google’s Safe Browsing [11] or being targeted by take-down requests.

Attackers can use a number of techniques to avoid detection [31]: for example, code obfuscation is effective against tools that rely on signatures, such as antivirus scanners; requiring arbitrary user interaction can undermine high-interaction honeyclients; probing for arcane characteristics of browser features (likely not correctly emulated in browser emulators) can thwart low-interaction honeyclients.

An effective way to implement this kind of circumventing techniques consists of adding some specialized “evasive code” whose only purpose is to cause detector tools to fail on an existing malicious script. Of course, the evasive code is designed in such a way that regular browsers (used by victims) effectively ignore it. Such evasive code could, for example, pack an exploit code in an obfuscation routine, check for human interaction, or implement a test for detecting browser emulators (such evasive code is conceptually similar to “red pills” employed in binary malware to detect and evade commonly-used analysis tools [10]).

Figure 2 shows an evasive modification to the original exploit of Figure 1, which we also found used in the wild. More precisely, the code tries to load a non-existent ActiveX

```

1 try {
2     new ActiveXObject("yutian");
3 } catch (e) {
4     var nop="%uyt9yt2yt9yt2";
5     var nop=(nop.replace(/yt/g,""));
6     var sc0="%ud5db%uc9c9%u87cd...";
7     var scl=%"+yutianu"+ByutianD+ ...;
8     var scl=(scl.replace(/yutian/g,""));
9     var sc2=%"+u"+54"+FF"+...+"8"+E"+E";
10    var sc2=(sc2.replace(/yutian/g,""));
11    var sc=unescape(nop+sc0+scl+sc2);
12 }

```

Figure 2: An evasion using non-existent ActiveX controls.

control, named `yutian` (Line 2). On a regular browser, this operation fails, triggering the execution of the `catch` branch (Lines 4–11), which contains an identical copy of the malicious code of Figure 1. However, low-interaction honeyclients usually emulate the ActiveX API by simulating the presence of *any* ActiveX control. In these systems, the loading of the ActiveX control does not raise any exception; as a consequence, the shellcode is not instantiated correctly, which stops the execution of the exploits and causes the honeyclient to fail to detect the malicious activity.

Detecting evasive code using code similarity. Code similarity approaches have been proposed in the past, but none of them has focused specifically on malicious JavaScript. There are several challenges involved when processing malicious JavaScript for similarities. Attackers actively try to trigger parsing issues in analyzers. The code is usually heavily obfuscated, which means that statically examining the code is not enough. The malicious code itself is designed to evade signature detection from antivirus products. This renders string-based and token-based code similarity approaches ineffective against malicious JavaScript. We will show later how regular code similarity tools, such as Moss [37], fail when analyzing obfuscated scripts. In *Revolver*, we extend tree-based code similarity approaches and focus on making our system robust against malicious JavaScript. We elaborate on our novel code similarity techniques in §3.4.

At a high-level overview, we use *Revolver* to detect and understand the similarity between two code scripts. Intuitively, *Revolver* is provided with the code of both scripts and their classification by one or more honeyclient tools. In our running example, we assume that the code in Figure 1 is flagged as malicious and the one in Figure 2 as benign. *Revolver* starts by extracting the Abstract Syntax Tree (AST) corresponding to each script. *Revolver* inspects the ASTs rather than the original code samples to abstract away possible superficial differences in the scripts (e.g., the renaming of variables). When analyzing the AST of Figure 2, it detects that it is similar to the AST of the code in Figure 1. The change is deemed to be interesting, since it introduces a difference (the try-catch statement) that may cause a change in the control flow of the original program. Our system also determines that the added code (the statement that tries to

load the ActiveX control) is indeed executed by tools visiting the page, thus increasing the relevance of the detected change (*execution bits* are described in more detail in §3.1). Finally, *Revolver* classifies the modification as a possible evasion attempt, since it causes the honeyclient to change its detection result (from malicious to benign).

Assumptions and limitations. Our approach is based on a few assumptions. *Revolver* relies on external detection tools to collect (and make available) a repository of JavaScript code, and to provide a classification of such code as either malicious or benign (i.e., *Revolver* is not a detection tool by itself). To obtain code samples and classification scores, we can rely on several publicly-available detectors [6, 13, 25].

Attackers might write a brand new attack with all components (evasion, obfuscation, exploit code) written from scratch. In such cases, *Revolver* will not be able to find any similarities the first time it analyzes these attacks. The lack of similarities though can be used to our advantage, since we can isolate brand-new attacks (provided that they can be identified by other means) based on the fact that we have never observed such code before.

In the same spirit, to detect evasions, *Revolver* needs to inspect two versions of a malicious script: the “regular” version, which does not contain evasive code, and the “evasive” version, which attempts to circumvent detection tools. Furthermore, if an evasion is occurring, we assume that a detection tool would classify these two versions differently. In particular, if only the evasive version of a JavaScript program is available, *Revolver* will not be able to detect this evasion. We consider this condition to be unlikely. In fact, trend results from a recent Google study on circumvention [31] suggest that malicious code evolves over time to incorporate more sophisticated techniques (including evasion). Thus, having a sufficiently large code repository should allow us to have access to both regular and evasive versions of a script. Furthermore, we have anecdotal evidence of malware authors creating different versions of their malicious scripts and submitting them to public analyzers, until they determine that their programs are no longer detected (this situation is reminiscent of the use of anti-antivirus services in the binary malware world [18]).

Revolver is not effective when server-side evasion (for example, IP cloaking) is used: in such cases, the malicious web site does not serve at all the malicious content to a detector coming from a blocked IP address, and, therefore, no analysis of its content is possible. This is a general limitation of all analysis tools and can be solved by means of a better analysis infrastructure (for example, by visiting malicious sites from IP addresses and networks that are not known to be associated with analysts and security researchers and cannot be easily fingerprinted by attackers).

3 Approach

In this section, we describe *Revolver* in detail, focusing on the techniques that it uses to find similarities between JavaScript files.

A high-level overview of *Revolver* is presented in Figure 3. First, we leverage an existing drive-by-download detection tool (an “Oracle”) to collect datasets of both benign and malicious web pages (§3.1). Second, *Revolver* extracts the ASTs (§3.2) of the JavaScript code contained in these pages and, leveraging the Oracle’s classification for the code that contains them, marks them as either benign or malicious. Third, *Revolver* computes a similarity score for each pair of ASTs, where one AST is malicious and the other one can be either benign or malicious (§3.3–§3.4). Finally, pairs that are found to have a similarity score higher than a given threshold are further analyzed to identify and classify their similarities (§3.5).

If *Revolver* finds similarities between two malicious scripts, then we classify this case as an instance of *evolution* (typically, an improvement of the original malicious code). On the other hand, if *Revolver* detects similarities between a malicious and a benign script, it performs an additional classification step. In particular, similarities can be classified by *Revolver* into one of four possible categories: *evasions*, *injections*, *data dependencies*, and *general evolutions*. We are especially interested in identifying evasions, which indicate changes that cause a script that had been found to be malicious before to be flagged as benign now.

It is important to note that, due to JavaScript’s ability to produce additional JavaScript code on the fly (which enables extremely complex JavaScript packers and obfuscators), performing this analysis statically would not be possible. *Revolver* works dynamically, by analyzing all JavaScript code that is compiled in the course of a web page’s execution. By including all these scripts, and the relationships between them (such as what code created what other code), *Revolver* is able to calculate JavaScript similarities among malicious web pages to an extent that is not, to our knowledge, possible with existing state-of-the-art code comparison tools.

3.1 Oracle

Revolver relies on existing drive-by-download detection tools for a single task: the classification of scripts in web pages as either malicious or benign. Notice that our approach is not tied to a specific detection technique or tool; therefore, we use the term “Oracle” to generically refer to any such detection system. In particular, several popular low- and high-interaction honeyclients (e.g., [6, 13, 25, 38]) or any antivirus scanner can readily be used for *Revolver*.

Revolver analyzes the Abstract Syntax Trees (ASTs) of individual scripts rather than examining web pages as a whole. Therefore, *Revolver* performs a refinement step, in which

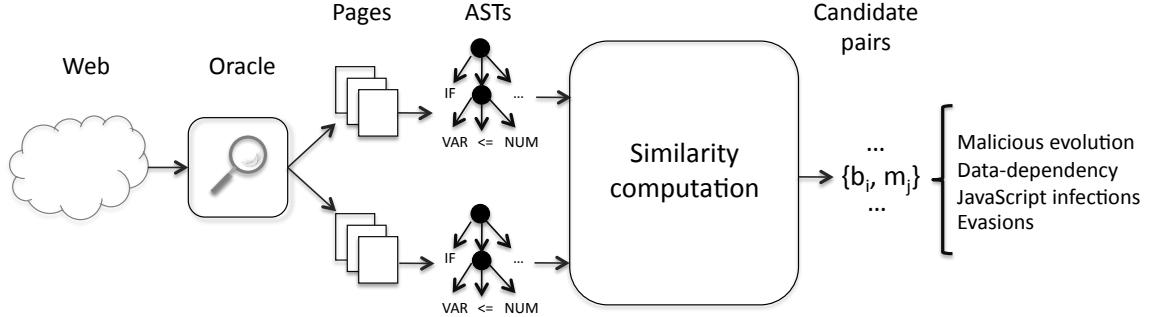


Figure 3: Architecture of *Revolver*.

i) individual ASTs are extracted from the web pages obtained from the Oracle, ii) their detection status is determined (that is, each AST is classified as either benign or malicious), based on the page classification provided by the Oracle, and iii) for each node in an AST, it is recorded whether the corresponding statement was executed. Of course, if an Oracle natively provides this fine-grained information, this step can be skipped.

More precisely, *Revolver* executes each web page using a browser emulator based on HtmlUnit [1]. The emulator parses the page and extracts all of its JavaScript content (e.g., the content of `script` tags and the body of event handlers). In particular, the ASTs of the JavaScript code are saved for later analysis. In addition, to obtain the AST of dynamically-generated code, *Revolver* executes the JavaScript code. At the end of the execution, for each node in the AST, *Revolver* keeps an **execution bit** to record whether the code corresponding to that node was executed. Whenever it encounters a function that generates new code (e.g., a call to the `eval()` or `setTimeout()` functions), *Revolver* analyzes the code that is generated by these functions. It also saves the parent-child relationship between scripts, i.e., which script is responsible for the execution of a dynamically-generated script. For example, the script containing the `eval()` call is considered the parent of the script that is evaluated. Similarly, *Revolver* keeps track of which script causes network resources to be fetched, for example, by creating an `iframe` tag.

Second, for each AST, *Revolver* determines if it is malicious or benign, based on the Oracle's input. More precisely, an AST is considered malicious if it is the parent of a malicious AST, or if it issued a web request that led to the execution of malicious code. This makes *Revolver* flexible enough to work with any Oracle.

3.2 Abstract Syntax Trees

Revolver's core analysis is based on the examination of ASTs rather than the source code of a script. The rationale for using ASTs is that they abstract away details that are

irrelevant for our analysis (and, in fact, often undesirable), while retaining enough precision to achieve good results.

For example, consider a script obtained from the code in Figure 1 via simple obfuscation techniques: renaming of variables and function names, introduction of comments, and randomization of whitespace. Clearly, we want *Revolver* to consider these scripts as similar. Making this decision can be non-trivial when inspecting the source code of the scripts. In fact, as a simple validation, we ran Moss, a system for determining the similarity of programs, which is often used as a plagiarism detection tool [37], on the original script and the one obtained via obfuscation. Moss failed to flag the two scripts as similar, as shown in the tool's output here [23]. However, the two scripts are identical when their AST representations are considered, since, in the trees, variables are represented by generic `VAR` nodes, independently of their names, and comments and whitespaces are ignored. This makes tree-based code similarity approaches more suitable for malicious JavaScript comparisons (and this is the reason why our analysis leverages ASTs as well). However, as shown in §3.4, we need to treat malicious code in a way that is different from previous techniques targeting benign codebases. Below, we describe our approach and necessary extensions in more detail.

Revolver transforms the AST produced by the JavaScript compiler into a *normalized node sequence*, which is the sequence of node types obtained by performing a pre-order visit of the tree. In total, there are 88 distinct node types, corresponding to different constructs of the JavaScript language. Examples of the node types include `IF`, `WHILE`, and `ASSIGN` nodes.

Figure 4 summarizes the data structures used by *Revolver* during its processing. We discuss *sequence summaries* in the next Section.

3.3 Similarity Detection

After extracting an AST and transforming it in its normalized node sequence, *Revolver* finds similar normalized node sequences. The result is a list of normalized node sequence

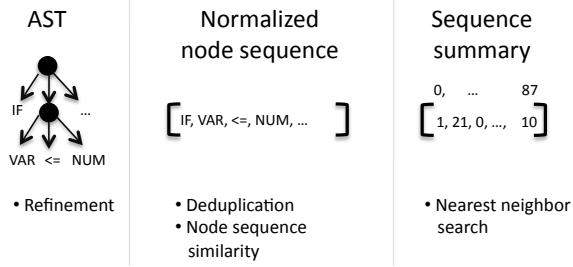


Figure 4: Data structures used by *Revolver*.

pairs. In particular, pairs of malicious sequences are compared to identify cases of evolution; pairs where one of the sequences is benign and the other malicious are analyzed to identify possible evasion attempts.

The similarity computation is based on computing the directed edit distance between two node sequences, which, intuitively, corresponds to the number of operations that are required to transform one benign sequence into the malicious one. Before discussing the actual similarity measurement, we discuss a number of minimization techniques that we use to make the computation of the similarity score feasible in datasets comprising millions of node sequences.

Deduplication. As a first step to reduce the number of similarity computations, we discard duplicates in our dataset of normalized node sequences. Since we use a canonical representation for the ASTs, we can easily and efficiently compute hashes of each sequence, which enables us to quickly identify groups of identical node sequences. In the remaining processing phases, we only need to consider one member of a group of identical node sequences (rather than all of its elements). Notice that identical normalized node sequences may correspond to different scripts, and may also have a different detection classification (we describe such cases in §3.5). Therefore, throughout this processing, we always maintain the association between node sequences and the scripts they correspond to, and whether they have been classified as malicious or benign.

Approximate nearest neighbors. Given a repository of n benign ASTs and m malicious ones, *Revolver* needs to compute $n \times m$ comparisons over (potentially long) node sequences. Even after the deduplication step, this may require a significantly large number of operations.

To address this problem, we introduce the idea of *sequence summaries*. A sequence summary is a compact summarization of a regular normalized node sequence, which stores the number of times each node type appears in the corresponding AST. Since there are 88 distinct node types, each node sequence is mapped into a point in an 88-dimensional Euclidean space. An advantage of sequence summaries is that they bound the length of the objects that will be compared (from potentially very large node sequences, corresponding to large ASTs, down to more manageable vectors

of fixed length).

Then, for each sequence summary s , we identify its *malicious neighbors*, that is, up to k malicious sequence summaries t , such that the distance between s and t is less than a chosen threshold τ_s . Intuitively, the malicious neighbors correspond to the set of ASTs that we expect to be most similar to a given AST. Determining the malicious neighbors of a sequence summary is an instance of the k-nearest neighbor search problem, for which various efficient algorithms have been proposed. In particular, we solve it by using the FLANN library [24].

In the remaining step, we compare sequence summaries only with their malicious neighbors, thus dramatically reducing the number of comparison to be performed.

Normalized node sequence similarity. Finally, we can compute the similarity between two normalized node sequences. More precisely, *Revolver* compares the normalized node sequence corresponding to a sequence summary s with each normalized node sequence that corresponds to a sequence summary of the malicious neighbors of s .

The similarity measurement is based on the pattern matching algorithm by Ratcliff et al. [33]. More precisely, given two node sequences, a and b , we first find their longest contiguous common subsequence (LCCS). Then, we recursively search for LCCS between the pieces of a and b to the left and right of the matching subsequence. The similarity of a and b is then returned as the number of nodes in common divided by the total number of nodes in the malicious node sequence. Therefore, identical ASTs will have similarity 1, and the similarity values decrease toward zero as two ASTs contain higher amounts of different nodes. This technique is robust against injections, where one benign script includes a malicious one, since all malicious nodes will be matched.

In addition to a numeric similarity score, the algorithm also provides a list of insertions for the two node sequences, that is, a list of AST nodes that would need to be added to one sequence to transform it into the other one. This information is very useful for our analysis, since it identifies the actual code that was added to an original malicious code.

After the similarity score is computed, we discard any pairs that have a similarity below a predetermined threshold τ_s .

Expansion. Once pairs of ASTs with high similarity have been identified, we need to determine the Oracle’s classification of the scripts they originate from. We, therefore, expand out any pairs that we deduplicated in the initial *Deduplication* step so that we associate the AST similarities to the scripts that they correspond to.

3.4 Optimizations

There are several techniques that we utilize to improve the results produced by the similarity detection steps. In particular, our objective is to restrict the pairs identified

as similar to “interesting” ones, i.e., those that are more likely to correspond to evasion attempts or significant new functionality. The techniques introduced here build upon tree-based code similarity approaches and are specific to malicious JavaScript.

Size matters. We observed that JavaScript code contains a lot of very small scripts. In the extreme case, it includes scripts comprising a single statement. We determined that the majority of such scripts are generated dynamically through calls to `eval()`, which, for example, dynamically invoke a second function. Such tiny scripts are problematic for our analysis: they have not enough functionality to perform malicious actions and they end up matching other short scripts, but their similarity is not particularly relevant. As a consequence, we **combine** ASTs that contain less than a set number of nodes (τ_z). We do this by taking into account how a script was generated: if another script generated code under our threshold, we inline the generated script back to its parent. If the script was not dynamically generated, then we treat it as if one script contained all static code under our threshold. This way the attacker cannot split the malicious code into multiple parts under our threshold in order to evade *Revolver*.

Repeated pattern detection. We also observed that, in certain cases, an AST may contain a set of nodes repeated a large number of times. This commonly occurs when the script uses some JavaScript data structure that yields many repeated AST nodes. For example, malicious scripts that unpack or deobfuscate their exploit payload frequently utilize a JavaScript *Array* of elements to store the payload. Their ASTs contain a node for every single element in the *Array*, which, in many cases, may have thousands of instances. An unwanted consequence, then, is that any script with a large *Array* will be considered similar to the malicious script (due to the high similarity of the array nodes), regardless of the presence of a decoding/unpacking routine (which, instead, is critical to determine the similarity of the scripts from a functional point of view). These obfuscation artifacts affect tree-based similarity algorithms, which will result in the detection of similar code pairs where the common parts are of no interest in the context of malicious JavaScript. To avoid this problem, we identify sequences of nodes that are repeated in a script more than a threshold (τ_p) and truncate them.

Similarity fragmentation. Although we have identified blocks of code that are shared across two scripts, it can be the case that these blocks are not continuous. One script can be broken down into small fragments that are matched to the other script in different positions. This is why we take into account the fragmentation of the matching blocks. To prune these cases, we recognize a similarity only if the fragmentation of the similarities is below a set threshold τ_f .

AST	Executed nodes	Classification
=	*	Data-dependency
*	=	Data-dependency
$B \subseteq M$	\neq	JavaScript injection
$M \subseteq B$	\neq	Evasion
\neq	\neq	General evolution

Table 1: Candidate pairs classification (B is a benign sequence, M is a malicious sequence, * indicates a wildcard value).

3.5 Classification

The outcome of the previous similarity detection step is a list of pairs of scripts that are similar. As we show in §5.1 we can have hundreds of thousands of similar pairs. Therefore, *Revolver* performs a classification step of similar pairs. That is, *Revolver* interprets the changes that were made between two scripts and classifies them. There are two cases, depending on the Oracle’s classification of the scripts in a pair. If the pair consists solely of malicious scripts, then we classify the similarity as a malicious evolution. The other case is a pair in which one script is malicious and one script is benign. We call such pairs *candidate pairs* (they need to be further tested before we can classify their differences). While the similarity detection has operated on a syntactic level (essentially, by comparing AST nodes), *Revolver* now attempts to determine the semantics of the differences.

In practice, *Revolver* classifies the scripts and their similarities into one of several categories, corresponding to different cases where an Oracle may flag differently scripts that are similar. Table 1 summarizes the classification algorithm used by *Revolver*.

Data-dependency category. *Revolver* checks if a pair of scripts belongs to the data-dependency category. A typical example of scripts that fall into this category is packers. Packers are tools that allow JavaScript authors to deliver their code in a packed format, significantly reducing the size of the code. In packed scripts, the original code of the script is stored as a compacted string or array, and its clear-text version is retrieved at run-time by executing an unpacking routine. Packers have legitimate uses (mostly, size compression): in fact, several open-source popular packers exist [9], and they are frequently used to distribute packed version of legitimate JavaScript libraries, such as jQuery. However, malware authors also rely on these very same packers to obfuscate their code and make it harder to be fingerprinted.

Notice that the ASTs of packed scripts (generated by the same packer) are identical, independently of their (unpacked) payload: in fact, they consist of the nodes of the unpacking routine (which is fixed) and of the nodes holding the packed data (typically, the assignment of a string literal

to a variable). However, the actual packed contents, which eventually determine whether the script is malicious or benign, are not retained at the AST level of the packer, but the packed content will eventually determine the nature of the overall script as benign or malicious.

Revolver categorizes as data-dependent pairs of scripts that are identical and have different detection classification.

As a slight variation to this scenario, *Revolver* also classifies as data-dependent pairs of scripts for which the ASTs are not identical, but the set of nodes that were actually executed are indeed the same. For example, this corresponds to cases where a function is added to the packer but is never actually executed during the unpacking.

Control-flow differences. The remaining categories are based on the analysis of AST nodes that are different in the two scripts, and, specifically, of nodes representing control-flow statement. We focus on such nodes because they give an attacker a natural way to implement a check designed to evade detection. In fact, such checks generally test a condition and modify the control flow depending on the result of the test.

More precisely, we consider the following control-flow related nodes: *TRY*, *CATCH*, *CALL*, *WHILE*, *FOR*, *IF*, *ELSE*, *HOOK*, *BREAK*, *THROW*, *SWITCH*, *CASE*, *CONTINUE*, *RETURN*, *LT(<)*, *LE(<=)*, *GT(>)*, *GE(>=)*, *EQ(==)*, *NE(!=)*, *SHEQ(====)*, *SNE(!==)*, *AND*, and *OR*. Depending on where these control-flow nodes were added, whether in the benign or in the malicious script, a candidate pair can be classified as a JavaScript injection or an evasion. Notice that we leverage here the *execution bits* to detect control flow changes that were actually executed and affected the execution of code that was found as malicious before.

JavaScript injection category. In some cases, malware authors insert malicious JavaScript code into existing benign scripts on a compromised host. This is done because, when investigating a compromise, webmasters may neglect to inspect files that are familiar to them, and thus such injections can go undetected. In particular, it is common for malware authors to add their malicious scripts to the code of popular JavaScript libraries hosted on a compromised site, such as jQuery and SWFObject.

In these cases, *Revolver* identifies similarities between a benign script (the original, legitimate jQuery code) and a malicious script (the library with the added malicious code). In addition, *Revolver* detects that the difference between the two scripts is due to the presence of control-flow nodes in the malicious script (the additional code added to the library), which are missing in the benign script. *Revolver* classifies such similarities as JavaScript injections, since the classification of the analyzed script changes from benign to malicious due to the introduction of additional code in the malicious version of the script.

Evasions category. Pairs of scripts that only differ because of the presence of additional control-flow nodes in the benign

script are categorized as evasions. In fact, these correspond to cases where a script, which was originally flagged as malicious by an Oracle, is modified to include some additional functionality that modifies its control flow (i.e., an evasive check) and, as a consequence, appears to be benign to the Oracle.

General evolution cases. Finally, if none of the previous categories applies to the current pair of scripts, it means that their differences are caused by the insertion of control-flow nodes in both the benign and malicious scripts. Unlike similarities in the evasion category, these similarities may signify generic evolution between the two scripts. *Revolver* flags these cases for manual review, at a lower priority than evasive cases.

4 Implementation

In this section, we discuss specific implementation choices for our approach.

We used the Wepawet honeyclient [6] as the Oracle of *Revolver*. In particular, the input to *Revolver* was the web pages processed by the Wepawet tool at real-time together with their detection classification. We used *Revolver* to extract ASTs from the pages analyzed by Wepawet, and to perform the similarity processing described in the previous sections.

As our processing infrastructure, we used a cluster of four worker machines to process submissions in parallel with the Oracle. Notice that all the steps in *Revolver*'s processing can be easily parallelized. In terms of performance, we managed to process up to 591,543 scripts on a single day, which was the maximum number of scripts that we got on a single day from the Oracle during our experiments.

We will now discuss the parameters that can be tuned in our algorithms (discussed in §3), explaining the concrete values we have chosen for our experiments.

Minimum tree size (τ_c). We chose 25 nodes as the minimum size of the AST trees that we will process before combining them to their parent. Smaller ASTs can result from calls to *eval* with tiny arguments, and from calls to short event handlers, such as *onLoad* and *onMouseOver*. We expect that such small ASTs correspond to short scripts that do not implement any interesting functionality alone, but complement the functionality of their parent script.

Minimum pattern size (τ_p). Another threshold that we set is the minimum pattern size. Any node sequence that is repeated more than this threshold is truncated to the threshold value. The primary application of pattern detection is to handle similar packers that decode payloads of different size. We chose 16 for this value, as current packers either work on relatively long arrays (longer than 16, and thus detected) or on single strings (one node, and thus irrelevant to this issue). This amount also excludes the possibility of compressing interesting code sequences, since we rarely see such long

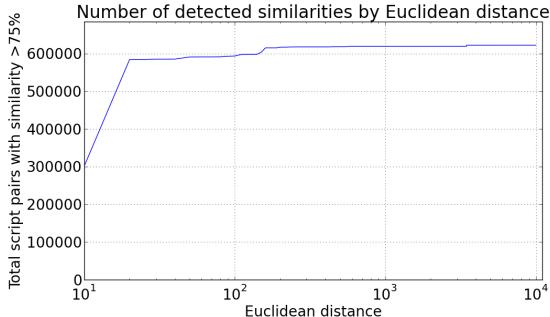


Figure 5: Number of detected similarities as a function of the distance threshold.

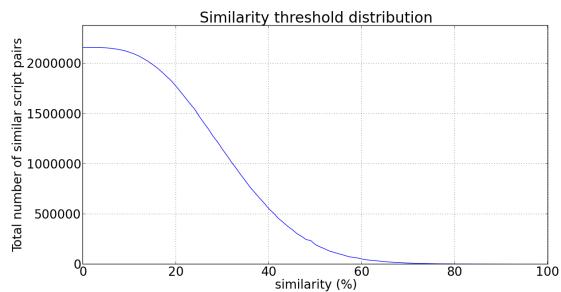


Figure 6: The resulting amount of similarities for different similarity thresholds.

patterns outside of packed payloads. Reducing this value would have the effect of making the tree similarity algorithm much more lax.

Nearest neighbor threshold (τ_n). In the nearest neighbors computation, we discard node sequences that are farther than a given distance d from the node sequence currently being inspected. We empirically determined a value for this parameter, by evaluating various values for d and inspecting the resulting similarities. From Figure 5, it is apparent that the amount of similarities that are detected levels off fairly sharply past $d = 1,000$. We determined that $10,000$ is a safe threshold that includes a majority of trees while allowing the similarity calculation to be computationally feasible.

Normalized node sequence similarity threshold (τ_s). Care has to be taken when choosing the threshold used to identify similar normalized node sequences. Intuitively, if this value is too low, we risk introducing significant noise into our analysis, which will make *Revolver* consider as similar scripts that in reality are not related to each other. On the contrary, if the value is too high, it will discard interesting similarities. Experimentally (see Figure 6), we determined that this occurs for similarity values in the 70%–80% interval. Therefore, we chose 75% as our similarity threshold (in other words, only node sequences that are 75% or more similar are further considered by *Revolver*).

Category	Similar Scripts	# Groups by malicious AST
JavaScript Injections	6,996	701
Data-dependencies	101,039	475
Evasions	4,147	155
General evolutions	2,490	273
Total	114,672	1,604

Table 2: Benign scripts from Wepawet that have similarities with malicious scripts and their classification from *Revolver*.

5 Evaluation

We evaluated the ability of *Revolver* to aid in detecting evasive changes to malicious scripts in real-world scenarios. While *Revolver* can be leveraged to solve other problems, we feel that automatically identifying evasions is the most important contribution to improving the detection of web-based malware.

5.1 Evasions in the wild

Revolver identifies possible evasion attempts by identifying similarities between malicious and benign code. Therefore, *Revolver*'s input is the output of any Oracle that classifies JavaScript code as either malicious or benign. To evaluate *Revolver*, we continuously submitted to *Revolver* all web pages that Wepawet examined. Since September 2012, we collected 6,468,623 web pages out of which 265,692 were malicious. We analyzed 20,732,766 total benign scripts and 186,032 total malicious scripts. Out of these scripts, we obtained 705,472 unique benign ASTs and 5,701 unique malicious ASTs.

Revolver applied the AST similarity analysis described in Section 3, and extracted the pairs of similar ASTs. Table 2 summarizes the results of classifying these similarities in the categories considered by *Revolver*. In particular, *Revolver* identified 6,996 scripts where malicious JavaScript was injected, 101,039 scripts with data-dependencies, 4,147 evasive scripts, and 2,490 scripts as general evolutions. We observe that many of these scripts can be easily grouped by their similarities with the same malicious script. Therefore, for ease of analysis, we group the pairs by their malicious AST component, and identify 701 JavaScript injections, 475 data-dependencies, 155 evasions, and 273 general evolutions. Our results indicate a high number of malicious scripts that share similarities with benign ones. This is due to the fact that injections and data-dependent malicious scripts naturally share similarities with benign scripts and we are observing many of these attacks in the wild.

To verify the results produced by *Revolver*, we manually analyzed all groups categorized as “evasions”. For the

rest of the categories we grouped the malicious ASTs into families based on their similarities with each other and examined a few similar pairs from each family. We found the results for the JavaScript injection and data-dependencies categories to be correct. The reason why *Revolver* classified a large number of scripts as data-dependencies is due to the extensive use of a few popular packers, such as the Edwards' packer [9]. For example, the jQuery library was previously officially distributed in a packed state to reduce its size.

Of the 155 evasions groups, we found that only five were not intended evasion attempts. We cannot describe all evasions in detail here, but we provide a brief summary for the most interesting ones in the next section.

The pairs in the “general evolutions” category consisted of cases where *Revolver* identified control flow changes in both the benign and malicious scripts. We manually looked into them and did not find any behavior that could be classified as evasive.

5.2 Evasions case studies

The evasions presented here exploit differences in the implementation of Wepawet’s JavaScript interpreter and the one used by regular browsers. Notice that these evasions can affect Oracles other than Wepawet; in particular, low-interaction honeyclients, such as the popular jsunpack [13] and PhoneyC [25].

We describe in more detail a subset of the evasions that we found from our experiment on real-world data. In the 22 evasion groups described here, we identified seven distinct evasion techniques, and one programming mistake in a malicious PDF.

We found three cases which leveraged subtle details in the handling of regular expressions and Unicode to cause a failure in a deobfuscation routine when executing in the Oracle (on the contrary, regular browsers would not be affected). In another case, the attackers replaced the JavaScript code used to launch an ActiveX exploit code with equivalent VBScript code. This is done because Internet Explorer can interpret VBScript, while most emulators do not support it. In a different case, the evasive code creates a `div` tag and checks for specific CSS properties, which are set correctly in Internet Explorer but not when executing in our Oracle. We will examine in more detail the next four evasion techniques.

Variable scope inside eval. We found that a successful evasion attack can be launched with minor changes to a malicious script. In one such case, shown in Figure 7, the authors of the malicious script changed a `replace` call with a call to `eval`, which, in turn, executed the same `replace`. While this change did not affect the functionality of the script in Internet Explorer, it did change it for our Oracle. In fact, in Wepawet’s JavaScript engine, the code inside the `eval` runs in a different scope, and thus, the locally-defined variable on which `replace` is called is not accessible. While

```

1 // Malicious
2 function foo() {
3 ...
4 W6Kh6V5E4 = W6Kh6V5E4.replace(/\W/g,Bm2v5BSJE);
5 ...
6 }
7 // Evasion
8 function foo() {
9 ...
10 var enryA = mxNEN+F7B07;
11 F7B07 = eval;
12 {}
13 enryA = F7B07('enryA.rep' + 'lace(/\W/g,CxFHg)←
14 ...
15 }
```

Figure 7: Evasion based on differences in the scope handling inside `eval` in different JavaScript engines.

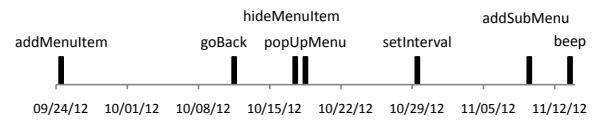


Figure 8: Timeline of PDF evasions automatically detected by *Revolver*.

the code successfully exploits Internet Explorer, it fails in our Oracle and is marked as benign.

Adobe-specific JavaScript execution. Figure 9 shows an evasion that leverages a specific characteristic of the JavaScript engine implementation included in Adobe Reader. In particular, in Adobe’s engine (unlike other engines), the `eval` function is accessible as a property of native objects, e.g., a string (line 8) [2]. Wepawet’s analyzer is not aware of this difference and fails on scripts that make use of this feature (marking them as benign). The functionally-identical script that does not use this trick, but directly invokes the `eval()` function, is correctly marked as malicious. We also found several instances of evasions related to PDF specific objects, like `app` and `target`, where missing functionality was used to render the malicious script harmless. We show a snippet of this evasion type found in the wild in Figure 10.

In Figure 8 we see the evasions related to the `app` object that were automatically detected by *Revolver* as found in the wild. Every time *Revolver* detected an evasion there is a spike in the figure, and we made the appropriate patches to Wepawet as soon as possible. What is of particular interest is the fact that attackers react to Wepawet’s patching by introducing a new evasion within a few days, making a tool like *Revolver* necessary to automatically keep track of this behavior and keep false negative detections as low as possible.

Evasion through exceptions. Another interesting evasion that *Revolver* found also leverages subtle differences in the JavaScript implementation used in Wepawet and in real browsers. In this case, the malicious script consists of a decryption function and a call to that function. The function

```

1 // Malicious
2 OlhG='evil_code'
3 wTGB4=eval
4 wTGB4(OlhG)
5
6 // Evasion
7 OlhG='evil_code'
8 wTGB4="this"]["eval"]
9 wTGB4(OlhG)

```

Figure 9: Evasion based on the ability to access the `eval` function as a property of native objects in Adobe’s JavaScript engine.

```

1 if((app.setInterval+/**/"")["indexOf"](aa)!=-1){
2     a=/**/target.creationDate.split('|')[0];

```

Figure 10: Evasion based on PDF specific objects *app* and *target*.

first initializes a variable with a XOR key, which will be used to decrypt a string value (encoding a malicious payload). The decoded payload is then evaluated via `eval`.

The evasion that we found follows the same pattern (see Figure 11), but with a few critical changes. In the modified code, the variable containing the XOR key is only initialized the first time that the decryption function runs; in sequential runs, the value of the key is modified in a way that depends on its prior value (Lines 16–17). After the key computation, a global variable is accessed. This variable is not defined the first time the decryption function is called, so that the function exits with an exception (Line 19). On Internet Explorer, this exception is caught, the variable is defined, and the decryption function is called again. The function then runs through the key calculation and then decrypts and executes the encrypted code by calling `eval`.

On our Oracle, a subtle bug (again, in the handling of `eval` calls) in the JavaScript engine caused the function to throw an exception the first *two* times that it was called. When the function is called the third time, it finally succeeds, modifies the XOR key, and attempts to decrypt the string. However, since the key calculation is run *three* times instead of two, the key is incorrect, and the decrypted string results in garbage data. We found three variations of this technique in our experiments.

A very interesting exception-based evasion that we found with *Revolver* was based on the immutability of `window.document.body`. The attacker checks if she can replace the `body` object with a *string*, something that should not be possible and should result in an exception, but it does not raise an exception in our Oracle because the `body` object is mutable. The interesting part is that we found three completely different malicious scripts evolving to incorporate this evasion, one of them being part of the popular exploit kit *Blackhole 2.0*. This is the first indication that evasion techniques are propagating to different attacking compo-

```

1 // Malicious
2 function deobfuscate(){
3     ... // Define var xorkey and compute its ←
4         value
5     for(...) { ... // XOR decryption with xorkey←
6         }
7     eval(deobfuscated_string);
8 }
9 try {
10    eval('deobfuscate();')
11 }
12 }
13
14 // Evasion
15 function deobfuscate(){
16     try { ... // is variable xorkey defined? }
17     catch(e){ xorkey=0; }
18     ... // Compute value of xorkey
19     VhplKO8 += 0; // throws exception the first ←
20         time
21     for(...) { ... // XOR decryption with xorkey←
22         }
23     eval(deobfuscated_string);
24 }
25 try { eval('deobfuscate();') } // 1st call
26 catch (e){
27     // Variable VhplKO8 is not defined
28     try {
29         VhplKO8 = 0; // define variable
30         eval('deobfuscate();'); // 2nd call
31     }
32     catch (e){
33         alert('ere');
34     }
35 }

```

Figure 11: An evasion taking advantage of a subtle bug in Wepawet’s JavaScript engine in order to protect the XOR key.

nents and indicates that attackers care to keep their attacks as stealthy as possible.

Unicode deobfuscation evasion. This evasion leveraged the fact that Unicode strings in string initializations and regular expressions are treated differently by different JavaScript engines. For example, *Revolver* found two scripts with a similarity of 82.6%. The script flagged as benign contained an additional piece of code that modified the way a function reference to `eval` was computed. More precisely, the benign script computed the reference by performing a regular expression replacement. While this operation executes correctly in Internet Explorer, it causes an error in the JavaScript engine used by Wepawet due to a bug in the implementation of regular expressions.

Incorrect PDF version check. Another similarity that *Revolver* identified involved two scripts contained inside two PDF files, one flagged as benign by Wepawet and the other as malicious. These scripts had a similarity of 99.7%. We determined that the PDF contained an exploit targeting Adobe Reader with versions between 7.1 and 9. The difference found by *Revolver* was caused by an incorrect version check in the exploit code. The benign code mistakenly checked for version greater or equal to 9 instead of less or equal to 9,

which combined with the previous checks for the version results in an impossible browser configuration and as a consequence the exploit was never fired. This case, instead of being an actual evasion, is the result of a mistake performed by the attacker. However, the authors quickly fixed their code and re-submitted it to Wepawet just 13 minutes after the initial, flawed submission.

False positives. The evasion groups contained five false positives. In this context, a false positive means that the similarity identified by *Revolver* is not responsible for the Oracle’s misdetection. More precisely, of these false positives, four corresponded to cases where the script execution terminated due to runtime JavaScript errors before the actual exploit was launched. While such behavior could be evasive in nature, we determined that the errors were not caused by any changes in the code, but by other dependencies. These can be due to missing external resources required by the exploit or because of a runtime error. In the remaining case, the control-flow change identified by *Revolver* was not responsible for the misdetection of the script.

Revolver’s impact on honeyclients. By continuously running *Revolver* in parallel with a honeyclient, we can improve the honeyclient’s accuracy by observing the evolution of malicious JavaScript. The results from such an integration with Wepawet indicate a shift in the attackers’ efforts from hardening their obfuscation techniques to finding discrepancies between analysis systems and targeted browsers. Popular exploit kits like *Blackhole* are adopting evasions to avoid detection, which shows that such evasions have emerged as a new problem in the detection of malicious web pages. *Revolver*’s ability to pinpoint, with high accuracy, these new techniques out of millions of analyzed scripts not only gives a unique view into the attackers’ latest steps, but indicates the necessity of such system as part of any honeyclient that analyzes web malware.

6 Discussion

As with any detection method, malware authors could find ways to attempt to evade *Revolver*. One possibility consists in splitting the malicious code into small segments, each of which would be interpreted separately through `eval`. *Revolver* is resilient against code fragmentation like this because it combines such scripts back to the parent script that generated them, reconstructing this way the original non-fragmented script.

It is also possible for malware authors to purposefully increase the Euclidean distance between their scripts so that otherwise similar scripts are no longer considered neighbors by the nearest neighbor algorithm. For example, malware authors could swap statements in their code, or inject junk code that has no effect other than decreasing the similarity score. Attackers could also create fully metamorphic scripts, similar to what some binary malware does [19]. We

can counteract these attacks by improving the algorithms we use to compute the similarity of scripts. For example, we could use a preprocessing step to normalize a script’s code (e.g., removing dead code). A completely different approach would be to leverage *Revolver* to correlate differences in the code of the same web pages when visited by multiple oracles: if *Revolver* detects significant differences in the code returned during these visits, then we can identify metamorphic web pages. In addition, metamorphic code raises the bar, since an attack needs to be programmatically different every time, and the code must be automatically generated without clearly-detectable patterns. Therefore, this would force attackers to give up their current obfuscation techniques and ability to reuse code.

An attacker could include an evasion and dynamically generate the attack code only if the evasion is successful. The attacker has two options: He can include the evasion code as the first step of the attack, or after initial obfuscation and environment setup. Evasions are hard to find and require significant manual effort by the attackers. Therefore, attackers will not reveal their evasion techniques since they are almost as valuable as the exploits they deliver. Moreover, introducing unobfuscated code compromises the stealthiness of the attack and can yield into detection through signature matching. The second option works in *Revolver*’s favor, since it allows our system to detect similarities in obfuscation and in environmental setup code.

Finally, an operational drawback of *Revolver* is the fact that manual inspection of the similarities that it identifies is currently needed to confirm the results it produces. The number of similarities that were found during our experiments made it possible to perform such manual analysis. In the future, we plan to build tools to support the inspection of similarities and to automatically confirm similarities based on previous analyses.

7 Related Work

Detection of evasive code. The detection of code that behaves differently when run in an analysis environment than when executed on a regular machine is a well-known problem in the binary malware community. A number of techniques have been developed to check if a binary is running inside an emulator or a virtual machine [10, 30, 36]. In this context, evasive code consists of instructions that produce different results or side-effects on an emulator and on a real host [21, 26]. The original malware code is modified to run these checks: if the check identifies an analysis system, the code behaves in a benign way, thus evading the detection.

Researchers have dealt with such evasive checks in two ways. First, they have designed systems that remain transparent to a wide range of malware checks [8, 39]. Second, they have developed techniques to detect the presence of such checks, for example by comparing the behavior of a sample

on a reference machine with that obtained by running it on an analysis host [3, 15].

Similar to the case of evasions against binary analysis environments, the results produced by honeyclients (i.e., the classification of a web page as either malicious or benign) can be confused by sufficiently-sophisticated evasion techniques. Honeyclients are not perfect and attackers have found ways to evade them [16, 31, 40]. For example, malicious web pages may be designed to launch an exploit only after they have verified that the current visitor is a regular user, rather than an automated detection tool. A web page may check that the visitor performs some activity, such as moving the mouse or clicking on links, or that the browser possesses the idiosyncratic properties of commonly-used modern browsers, rather than being a simple emulator. If any of these checks are not satisfied, the malicious web page will refrain from launching the attack, and, as a consequence, will be incorrectly classified as benign, thus evading detection.

The problem of evasive code in web attacks has only recently been investigated. Kolbitsch et al. [17] have studied the “fragility” of malicious code, i.e., its dependence for correct execution on the presence of a particular execution environment (e.g., specific browser and plugins versions). They report several techniques used by malicious code for environment matching: some of these techniques may well be used to distinguish analysis tools from regular browsers and evade detection. They propose ROZZLE, a system that explores multiple execution paths in a program, thus bypassing environment checks. Rozzle only detects fingerprinting that leverages control flow branches and depends upon the environment. It can be evaded by techniques that do not need control-flow branches, e.g., those based on browser or JavaScript quirks. For example, the property `window.innerWidth` contains the width of the browser window viewport in Firefox and Chrome, and is `undefined` in Internet Explorer. Therefore, a malicious code that initialized a decoding key as `xorkey=window.innerWidth*0+3` would compute a different result for xorkey in Firefox/Chrome (3) and IE (Not a Number error), and could be used to decode malicious code in specific browsers. Rozzle will not trigger its multi-path techniques in such cases and can be evaded.

Revolver takes a different approach to identifying evasive code in JavaScript programs. Instead of forcing an evasive program to display its full behavior (by executing it in parallel on a reference host and in an analysis environment [3], or by forcing the execution through multiple, interesting paths [17]), it leverages the existence of two distinct but similar pieces of code and the fact that, despite their similarity, they are classified differently by detection tools. In addition, *Revolver* can precisely and automatically identify the code responsible for an evasion.

JavaScript code analysis. In the last few years, there have been a number of approaches to analyzing JavaScript

code. For example, Prophiler [5] and ZOZZLE [7] have used characteristics of JavaScript code to predict if a script is malicious or benign. ZOZZLE, in particular, leverages features associated with AST context information (such as, the presence of a variable named `scode` in the context of a loop), for its classification.

Cujo [34] uses static and dynamic code features to identify malicious JavaScript programs. More precisely, it processes the static program and traces of its execution into q-grams that are classified using machine learning techniques.

Revolver performs the core of its analysis statically, by computing the similarity between pairs of ASTs. However, *Revolver* also relies on dynamic analysis, in particular to obtain access to the code generated dynamically by a script (e.g., via the `eval()` function), which is a common technique used by obfuscated and malicious code.

Code similarity. The task of automatically detecting “clones,” i.e., segments of code that are similar (according to some notion of similarity), is an established line of work in the software engineering community [27, 35]. Unfortunately, many of the techniques developed here assume that the code under analysis is well-behaved or at least not adversarial, that is, not actively trying to elude the classification. Of course, this assumption does not hold when examining malicious code.

Similarity between malicious binaries has been used to quickly identify different variants of the same malware family. The main challenge in this context is dealing with extremely large numbers of samples without source code and large feature spaces from runtime data. Different techniques have been proposed to overcome these issues: for example, Bayer et al. [4] rely on locality sensitive hashing to reduce the number of items to compare, while Jong et al. [14] use feature hashing to reduce the number of features.

As a comparison, *Revolver* aims not only to identify pieces of JavaScript code that are similar, but also to understand why they differ and especially if these differences are responsible for changing the classification of the sample.

8 Conclusions

In this paper, we have introduced and demonstrated *Revolver*, a novel approach and tool for detecting malicious JavaScript code similarities on a large scale. *Revolver*’s approach is based on identifying scripts that are similar and taking into account an Oracle’s classification of every script. By doing this, *Revolver* can pinpoint scripts that have high similarity but are classified differently (detecting likely evasion attempts) and improve the accuracy of the Oracle.

We performed a large-scale evaluation of *Revolver* by running it in parallel with the popular Wepawet drive-by-detection tool. We identified several cases of evasions that are used in the wild to evade this tool (and, likely, other tools

based on similar techniques) and fixed them, improving this way the accuracy of the honeyclient.

Acknowledgements: This work was supported by the Office of Naval Research (ONR) under grant N00014-12-1-0165 and under grant N00014-09-1-1042, and the National Science Foundation (NSF) under grants CNS-0845559 and CNS-0905537, and by Secure Business Austria.

References

- [1] HtmlUnit. <http://htmlunit.sourceforge.net/>.
- [2] JavaScript for Acrobat API Reference. http://wwwimages.adobe.com/www.adobe.com/content/dam/Adobe/en/devnet/acrobat/pdfs/js_api_reference.pdf.
- [3] D. Balzarotti, M. Cova, C. Karlberger, C. Kruegel, E. Kirda, and G. Vigna. Efficient Detection of Split Personalities in Malware. In *Proc. of the Symposium on Network and Distributed System Security (NDSS)*, 2010.
- [4] U. Bayer, P. M. Comparetti, C. Hlauschek, C. Kruegel, and E. Kirda. Scalable, Behavior-Based Malware Clustering. In *Proc. of the Symposium on Network and Distributed System Security (NDSS)*, 2009.
- [5] D. Canali, M. Cova, G. Vigna, and C. Kruegel. Prophiler: A Fast Filter for the Large-scale Detection of Malicious Web Pages. In *Proc. of the International World Wide Web Conference (WWW)*, 2011.
- [6] M. Cova, C. Kruegel, and G. Vigna. Detection and Analysis of Drive-by-Download Attacks and Malicious JavaScript Code. In *Proc. of the International World Wide Web Conference (WWW)*, 2010.
- [7] C. Curtsinger, B. Livshits, B. Zorn, and C. Seifert. Zozzle: Low-overhead Mostly Static JavaScript Malware Detection. In *Proc. of the USENIX Security Symposium*, 2011.
- [8] A. Dinaburg, P. Royal, M. Sharif, and W. Lee. Ether: Malware Analysis via Hardware Virtualization Extensions. In *Proc. of the ACM Conference on Computer and Communications Security (CCS)*, 2008.
- [9] D. Edwards. Dean Edwards Packer. <http://bit.ly/TWQ46b>.
- [10] P. Ferrie. Attacks on Virtual Machines. In *Proc. of the Association of Anti-Virus Asia Researchers Conference*, 2003.
- [11] Google. Safe Browsing API. <http://code.google.com/apis/safebrowsing/>.
- [12] C. Grier, L. Ballard, J. Caballero, N. Chachra, C. J. Dietrich, K. Levchenko, P. Mavrommatis, D. McCoy, A. Nappa, A. Pitsillidis, N. Provos, M. Z. Rafique, M. A. Rajab, C. Rossow, K. Thomas, V. Paxson, S. Savage, and G. M. Voelker. Manufacturing Compromise: The Emergence of Exploit-as-a-Service. In *Proc. of the ACM Conference on Computer and Communications Security (CCS)*, 2012.
- [13] B. Hartstein. jsunpack – a generic JavaScript unpacker. <http://jsunpack.jeek.org/dec/go>.
- [14] J. Jang, D. Brumley, and S. Venkataraman. BitShred: Feature Hashing Malware for Scalable Triage and Semantic Analysis. In *Proc. of the ACM Conference on Computer and Communications Security (CCS)*, 2011.
- [15] M. G. Kang, H. Yin, S. Hanna, S. McCamant, and D. Song. Emulating Emulation-Resistant Malware. In *Proc. of the Workshop on Virtual Machine Security (VMSec)*, 2009.
- [16] A. Kapravelos, M. Cova, C. Kruegel, and G. Vigna. Escape from Monkey Island: Evading High-Interaction Honeyclients. In *Proc. of the Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*, 2011.
- [17] C. Kolbitsch, B. Livshits, B. Zorn, and C. Seifert. Razzle: De-Cloaking Internet Malware. In *Proc. of the IEEE Symposium on Security and Privacy*, 2012.
- [18] B. Krebs. Virus Scanners for Virus Authors, Part II. <http://krebsonsecurity.com/2010/04/virus-scanners-for-virus-authors-part-ii/>, 2010.
- [19] F. Leder, B. Steinbock, and P. Martini. Classification and detection of metamorphic malware using value set analysis. In *Proc. of the Conference on Malicious and Unwanted Software (MALWARE)*, 2009.
- [20] L. Lu, V. Yegneswaran, P. Porras, and W. Lee. BLADE: An Attack-Agnostic Approach for Preventing Drive-By Malware Infections. In *Proc. of the ACM Conference on Computer and Communications Security (CCS)*, 2010.
- [21] L. Martignoni, R. Paleari, G. F. Roglia, and D. Bruschi. Testing CPU Emulators. In *Proc. of the International Symposium on Software Testing and Analysis (ISSTA)*, 2009.
- [22] Microsoft. Microsoft Security Intelligence Report, Volume 13. Technical report, Microsoft Corporation, 2012.

- [23] Moss. Moss with obfuscated scripts. <http://goo.gl/XzJ7M>.
- [24] M. Muja and D. G. Lowe. Fast Approximate Nearest Neighbors with Automatic Algorithm Configuration. In *Proc. of the Conference on Computer Vision Theory and Applications (VISAPP)*, 2009.
- [25] J. Nazario. PhoneyC: A Virtual Client Honeypot. In *Proc. of the USENIX Workshop on Large-Scale Exploits and Emergent Threats (LEET)*, 2009.
- [26] R. Paleari, L. Martignoni, G. F. Roglia, and D. Bruschi. A Fistful of Red-Pills: How to Automatically Generate Procedures to Detect CPU Emulators. In *Proc. of the USENIX Workshop on Offensive Technologies (WOOT)*, 2009.
- [27] J. Pate, R. Tairas, and N. Kraft. Clone Evolution: a Systematic Review. *Journal of Software Maintenance and Evolution: Research and Practice*, 2011.
- [28] N. Provos, P. Mavrommatis, M. Rajab, and F. Monroe. All Your iFRAMEs Point to Us. In *Proc. of the USENIX Security Symposium*, 2008.
- [29] N. Provos, D. McNamee, P. Mavrommatis, K. Wang, and N. Modadugu. The Ghost in the Browser: Analysis of Web-based Malware. In *Proc. of the USENIX Workshop on Hot Topics in Understanding Botnet*, 2007.
- [30] T. Raffetseder, C. Kruegel, and E. Kirda. Detecting System Emulators. In *Proc. of the Information Security Conference*, 2007.
- [31] M. A. Rajab, L. Ballard, N. Jagpal, P. Mavrommatis, D. Nojiri, N. Provos, and L. Schmidt. Trends in Circumventing Web-Malware Detection. Technical report, Google, 2011.
- [32] P. Ratanaworabhan, B. Livshits, and B. Zorn. Nozzle: A Defense Against Heap-spraying Code Injection Attacks. In *Proc. of the USENIX Security Symposium*, 2009.
- [33] J. W. Ratclif. Pattern Matching: the Gestalt Approach. *Dr. Dobb's*, 1988.
- [34] K. Rieck, T. Krueger, and A. Dewald. Cujo: Efficient Detection and Prevention of Drive-by-Download Attacks. In *Proc. of the Annual Computer Security Applications Conference (ACSAC)*, 2010.
- [35] C. K. Roy and J. R. Cordy. A Survey on Software Clone Detection Research. Technical report, School of Computing, Queen's University, 2007.
- [36] J. Rutkowska. Red Pill... or how to detect VMM using (almost) one CPU instruction. <http://www.invisiblethings.org/papers/redpill.html>, 2004.
- [37] S. Schleimer, D. Wilkerson, and A. Aiken. Winnowing: local algorithms for document fingerprinting. In *Proc. of the 2003 ACM SIGMOD international conference on Management of data*, 2003.
- [38] The Honeynet Project. Capture-HPC. <https://projects.honeynet.org/capture-hpc>.
- [39] A. Vasudevan and R. Yerraballi. Cobra: Fine-grained Malware Analysis using Stealth Localized Executions. In *Proc. of the IEEE Symposium on Security and Privacy*, 2006.
- [40] Y.-M. Wang, D. Beck, X. Jiang, R. Roussev, C. Verbowksi, S. Chen, and S. King. Automated Web Patrol with Strider HoneyMonkeys: Finding Web Sites That Exploit Browser Vulnerabilities. In *Proc. of the Symposium on Network and Distributed System Security (NDSS)*, 2006.

Language-based Defenses against Untrusted Browser Origins

Karthikeyan Bhargavan
INRIA Paris-Rocquencourt

Antoine Delignat-Lavaud
INRIA Paris-Rocquencourt

Sergio Maffeis
Imperial College London

Abstract

We present new attacks and robust countermeasures for security-sensitive components, such as single sign-on APIs and client-side cryptographic libraries, that need to be safely deployed on untrusted web pages. We show how failing to isolate such components leaves them vulnerable to attacks both from the hosting website and other components running on the same page. These attacks are not prevented by browser security mechanisms alone, because they are caused by code interacting within the same origin. To mitigate these attacks, we propose to combine fine-grained component isolation at the JavaScript level with cryptographic mechanisms. We present Defensive JavaScript (DJS), a subset of the language that guarantees the behavior integrity of scripts even when loaded in a hostile environment. We give a sound type system, type inference tool, and build defensive libraries for cryptography and data encodings. We show the effectiveness of our solution by implementing several applications using defensive patterns that fix some of our original attacks. We present a model extraction tool to analyze the security properties of our applications using a cryptographic protocol verifier.

1 Defensive Web Components

Web users increasingly store sensitive data on servers spread across the web. The main advantage of this dispersal is that users can access their data from browsers on multiple devices, and easily share this data with friends and colleagues. The main drawback is that concentrating sensitive data on servers makes them tempting targets for cyber-criminals, who use increasingly sophisticated browser-based attacks to steal user data.

In response to these concerns, web applications now offer users more control over who gets access to their data, using authorization protocols such as OAuth [23] and application-level cryptography. These security mechanisms are often implemented as JavaScript components that may be included by any website, where they mediate a three-party interaction between the host website, the user (represented by her browser), and a server that holds the sensitive data on behalf of the user.

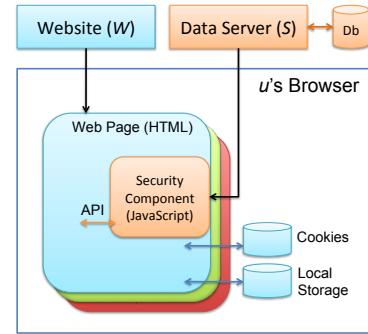


Figure 1: JavaScript Security Component

The typical deployment scenario that concerns us is depicted in Figure 1. A website W wishes to access sensitive user data stored at S . So, it embeds a JavaScript component provided by S . When a user visits the website, the component authenticates the user and exposes an API through which W may access the user's data, if the user has previously authorized W at S . For authenticated users on authorized websites, the component typically holds some client-side secret, such as an access token or encryption key, which it can use to validate data requests and responses. When the user closes or navigates away from the website, the component disappears and the website can no longer access the API.

A popular example of this scenario is single sign-on mechanism, such as Login with Facebook (detailed in Section 2). Facebook (S) provides a JavaScript component that websites like Pinterest (W) may use to request the identity and social profile of a visiting user, via an API that obtains a secret OAuth token for the current user and attaches it with each request to Facebook.

Other examples include payment processing APIs like Google Checkout, password manager bookmarklets like Lastpass, anti-CSRF protections like OWASP CSRF-Guard, and client-side encryption libraries for cloud storage services like Mega. More generally, a website may host a number of components from different providers, each keeping its own secrets and protecting its own API.

What we find particularly interesting is that the data and functionality of these JavaScript components is often of higher value than the website that hosts it. This is contrary to the usual web security threat model where

a website tries to defend itself from third-party components. Instead, we consider components that are designed to increase security of a website by delegating sensitive operations (e.g. password storage, credit card approval) to trusted third-party servers. For the data handled by such components, we seek to offer a limited security guarantee to the user. If a user temporarily visits (and authorizes) a compromised website W , any data read by the website during the visit may be leaked to the adversary, but the user can still expect the component to protect long-term access to her data on S . Our aim is not to prevent compromises in W or to prevent all data leaks. Instead, we enable a robust defense-in-depth strategy, where the security mechanisms of a website do not completely break if it loads a single malicious script.

Goals, Threats, and Attacks. Our goal is to design hardened JavaScript components that can protect sensitive user data and other long-term secrets such as access tokens and encryption keys from unauthorized parties. So far, such goals have proven surprisingly hard to guarantee for components written in JavaScript that run in the browser environment and interact with standard websites (e.g. see [1, 5, 6, 10, 41, 42]). What makes such components so hard to secure?

In Section 2, we survey the state of the art in three categories of security components: single sign-on mechanisms, password managers, and client-side encryption libraries used for cloud storage. We find that these components must defend against three kinds of threats. First, they may be loaded into a malicious website that pretends to be a trusted website. Second, even on a trusted website they may be loaded alongside other scripts that may innocently (or maliciously) modify the JavaScript builtin objects in a way that changes the runtime behavior of the component. Third, some webpage on the same domain (or subdomain) as W may either host malicious user-provided content or might contain a cross-site scripting (XSS) attack or any number of web vulnerabilities.

We found that the defenses against these threats prove inadequate for many of the components in our survey. We report previously-unknown attacks on widely-used components that completely compromise their stated security goals, despite their use of sophisticated protocols and cryptographic mechanisms. Our attacks exploit a wide range of problems, such as bugs in JavaScript components, bugs in browsers, and standard web vulnerabilities (XSS, CSRF, open redirectors), and build upon them to fool components into revealing their secrets. Eliminating specific bugs and vulnerabilities can only be a stop-gap measure. We aim instead to design JavaScript components that are provably robust against untrusted hosts.

Same Origin Policy (SOP). Most browser security mechanisms (including new HTML5 APIs, such as

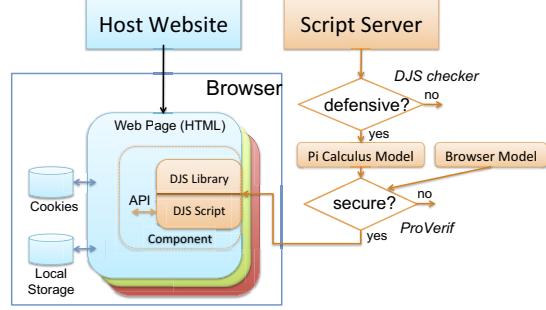


Figure 2: DJS Architecture

postMessage, localStorage, and WebCrypto) are based on the *origin* from which a webpage was loaded, defined as the domain of the website and the protocol and port used to retrieve it (e.g. <https://facebook.com:443>). The SOP isolates the JavaScript execution environments of frames and windows loaded from different origins from each other. In contrast, frames from the same origin can directly access each other’s variables and functions, across a page and even across windows.

The SOP does not directly apply to our scenario since our components run in the same origin as the host website. To use the SOP, components must open new frames or windows on a separate origin and implement a messaging protocol between them and the host website. As we show in Section 2, such components are difficult to get right and the JavaScript programs that implement them require close analysis.

Our Proposal. We advocate a language-based approach that is complementary to the SOP and protects scripts running in the same origin from each other. This enables a defense-in-depth strategy where the functionality and secrets of a component can be protected even if some page on the host origin is compromised.

We propose a *defensive* architecture (Figure 2) that enables developers to write verified JavaScript components that combine cryptography and browser security mechanisms to provide strong formal guarantees against entire classes of attacks. Its main elements are:

DJS: A defensive subset of JavaScript, with a static type checker, for writing security-critical components.

DJS Library: A library written (and typechecked) in DJS, with cryptographic and encoding functions.

DJS2PV: A tool that automatically analyzes the compositional security of a DJS component by translating it to the applied pi calculus for verification when combined with models of the browser and DJS library, using the ProVerif protocol analyzer.

Script Server: A verified server for distributing defensive scripts embedded with session-specific encryption keys.

Our architecture relies on the willingness of developers to program security-critical code in DJS, a well-defined restricted subset of JavaScript. In return, they obtain automated analysis and strong security guarantees for their code. Moreover, no restriction is enforced on untrusted code. In order to verify authentication and secrecy properties of the defensive components once embedded in the browser, we rely on ProVerif [13], a standard protocol verification tool that has been used extensively to analyze cryptographic mechanisms, with the WebSpi library [6], a recent model for web security mechanisms. Unlike previous works that use WebSpi, we automatically extract models from DJS code.

As we show in Section 6, DJS can significantly improve the security of current web applications with minimal changes to their functionality. Emerging web security solutions, such as Content Security Policy, ECMAScript 5 Strict, and WebCryptoAPI, offer complementary protections, and when they become widespread, they may enable us to relax some DJS restrictions, while retaining its strong security guarantees.

Towards Defensive JavaScript. A cornerstone of our defensive architecture is the ability of trusted scripts to resist same-origin attacks, because requiring that all scripts on an origin be trusted is too demanding. We investigate language-based isolation for such trusted scripts, and identify the *defensive JavaScript problem*:

Define a defensive subset of JavaScript to write stateful functions whose behavior cannot be influenced (besides by their arguments) by untrusted code running in the same environment, before or after such functions are defined. Untrusted code should not be able to learn secrets by accessing the source code of defensive functions or directly accessing their internal state.

This problem is harder than the one tackled by JavaScript subsets such as ADsafe [16] or Caja [40], which aim to protect trusted scripts by sandboxing untrusted components. In particular, those subsets assume the initial JavaScript environment is trusted, and that all untrusted code can be restricted. In our case, defensive code must run securely in a JavaScript engine that is running arbitrary untrusted code.

Contributions. Our main contributions are:

1. We identify common concerns for applications that embed secure components in arbitrary third party websites, and new attacks on these applications;
2. We present DJS, a defensive subset of JavaScript for programming security components. DJS is the first language-based isolation mechanism that does not restrict untrusted JavaScript and does not rely on a first-running bootstrapper;
3. We develop tools to verify that JavaScript code is valid DJS, and to extract ProVerif models from DJS;

4. We define DJCL, a defensive crypto library with encoding and decoding utilities that can be safely used in untrusted JavaScript environments. DJCL can be included *as is* on any website;
5. We identify general patterns that leverage DJS and cryptography to enforce component isolation in the browser, and in particular, we propose fixes to several broken web applications.

Supporting materials for this paper, including code, demos, and a technical report with proofs are available online [11].

2 Attacks on Web Security Components

We survey a series of web security components and investigate their security; Table 1 presents our results. Our survey focuses on three categories of security components that implement the pattern depicted in Figure 1.

Single Sign-On Buttons: (e.g. Facebook login on Hulu)

W loads a script from S that allows it to access the verified identity of u at S , and possibly other social data (photo, friend list, etc.).

Password Managers: (e.g. LastPass, 1Password)

u installs a browser plugin or bookmarklet from S ; when the browser visits W , the plugin retrieves an (encrypted) password or credit card number for u from S and uses it to fill in a form on W .

Host-Proof Cloud Storage: (e.g. ConfiChair, Mega)

A privacy-sensitive website W loads a client-side encryption library from S that retrieves an encrypted file from the cloud, decrypts it with a user-specified key (or passphrase) and releases the file to W .

We conjecture that other security components that fit our threat model, such as payment processing APIs and social sharing widgets, would have similar security goals and solutions, and suffer from similar weaknesses.

Methodology. Our method for studying each component is as follows. We first study the source code of each component and run it in various environments to discover the core protection mechanisms that it depends on. For example, in order to protect the integrity of their JavaScript code from the hosting webpage, some components require users to install them as bookmarklets (e.g. LastPass) or browser extensions (e.g. 1Password), whereas others rely on their code being downloaded within frames (e.g. Facebook), within signed Java applets (e.g. Wuala) or as signed JavaScript (e.g. Mega). In order to protect the confidentiality of data, many components rely on cryptography, implemented either in Java or in JavaScript. We anticipate that many of these will eventually use the native HTML Web Cryptography API when it becomes widely available.

Product	Category	Protection Mechanism	Attack Vectors Found	Secrets Stolen
Facebook	Single Sign-On Provider	Frames	Origin Spoofing, URL Parsing Confusion	Login Credential, API Access Token
Helios, Yahoo, Bitly, WordPress, Dropbox, Firefox	Single Sign-On Clients	OAuth Login	HTTP Redirector, Hosted Pages	Login Credential, API Access Token
	Web Browser	Same-Origin Policy	Malicious JavaScript, CSP Reports	Login Credential, API Access Token
1Password, RoboForm	Password Manager	Browser Extension	URL Parsing Confusion, Metadata Tampering	Password
LastPass, PassPack, Verisign, SuperGenPass	Password Manager	Bookmarklet, Frames, JavaScript Crypto	Malicious JavaScript, URL Parsing Confusion	Bookmarklet Secret, Encryption Key
SpiderOak	Encrypted Cloud Storage	Server-side Crypto	CSRF	Files, Encryption Key
Wuala	Encrypted Cloud Storage	Java Applet, Crypto	Client-side Exposure	Files, Encryption Key
Mega, ConfiChair, Helios	Encrypted Cloud Storage, Crypto Web Applications	JavaScript Crypto	XSS	Encryption Key
		Java Applet, Crypto	XSS	Password, Encryption Key

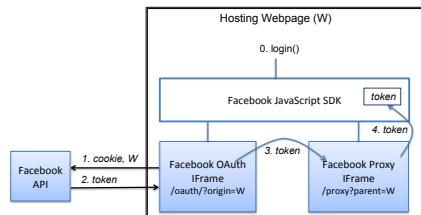
Table 1: Survey: Representative Attacks on Security Components

Next, we investigate whether any of these protection mechanisms make assumptions about the browser, or the security of the host website, or component server, that could be easily broken. We found a variety of bugs in specific JavaScript components and in the Firefox browser, and we found standard web vulnerabilities in various websites (CSRF, XSS, Open Redirectors).

Finally, the bulk of the analysis consists in converting these bugs and vulnerabilities to concrete exploits on our target JavaScript components. Table 1 only reports the exploits that resulted in a complete circumvention of the component’s security, that is, attacks where long-term secrets like encryption keys and user files are leaked. We also found other, arguably less serious, attacks not noted here, such as CSRF and login CSRF attacks on the data server and attacks that enable user tracking and fingerprinting.

In this section, we detail two illustrative examples of our analysis. For details on our other attacks, see [11].

2.1 Login with Facebook



When a website W wants to incorporate single-sign on with Facebook (S) on one of its pages, it can simply include the Facebook JavaScript SDK and call `FB.login()`. Behind the scene, this kicks off a three-party authoriza-

tion protocol called OAuth 2.0 [23], where an authorization server on Facebook issues an *access token* to W if the currently logged-in user has authorized W for single sign-on; otherwise, the user is asked to log in and authorize W . W may then call `FB.getAccessToken` to obtain the raw token, but more commonly, it calls `FB.api` to make specific calls to Facebook’s REST API (with the token attached). Hence, W can read the current user’s verified identity at Facebook or other social data. Google, Live, and Twitter provide a similar experience with their JavaScript SDKs.

When W calls `FB.login`, two iframes are created.

The first *OAuth* iframe is sourced from Facebook’s authorization server with W ’s client id (I_W) as parameter:

https://www.facebook.com/dialog/oauth?client_id=Iw

This page authenticates the user (with a cookie), verifies that she has authorized W , issues a fresh access token (T) and redirects the iframe to a Facebook URL with the token as fragment identifier:

https://static.ak.facebook.com/connect/xd_arbiter.php#token=T

Meanwhile, the second *Proxy* iframe is loaded from:

https://static.ak.facebook.com/connect/xd_arbiter.php#origin=W

where the fragment identifier indicates the origin W of the host page. Since both frames are now on the same origin, they can directly read each other’s variables and call each other’s functions. The OAuth iframe calls a function on the Proxy iframe with the access token T , and this function forwards T in a `postMessage` event to the parent frame (with target origin set to W). The token is then received by a waiting `FB.login` callback function, and token retrieval is complete. W can call `FB.api` to verify the user’s identity and access token.

Protection Mechanisms. The main threat to the above exchange is from a malicious website M pretending to be W . The Facebook JavaScript SDK relies on the following browser security mechanisms:

- Both iframes are sourced from origins distinct from M , so scripts on M cannot interfere with these frames, except to set their source URIs;
- The redirection of the OAuth frame is transparent to the page; M cannot read the redirection URI;
- Scripts on M cannot directly access Facebook because the browser and the web server will prevent such cross-origin accesses;
- Scripts on M will not be able to read the `postMessage` event, since it is set to target origin W .

All four mechanisms are variations of the SOP (applied to iframes, redirection URIs, `XmlHttpRequest`, and `postMessage`). The intuition is that if M and W are different origins, their actions (even on the same page) are opaque to each other. However, many aspects of the SOP are not standard but browser-specific and open to interpretation [43]. For example, we show bugs in recent versions of Firefox that break redirection transparency.

Writing JavaScript code to compose browser mechanisms securely is not easy. We demonstrate several bugs in the Facebook SDK that enable M to bypass origin authentication. Moreover, the SOP does not distinguish between same-origin pages or scripts. Hence, a hidden assumption in the above exchange is that all scripts loaded on all pages of W have access to the token and must be trusted. We show how sub-origin attacks on Facebook’s client can steal tokens.

Breaking Redirection Transparency on Firefox. We found two bugs in how Firefox enforced the same origin policy for redirection URIs.

First, we found that recent versions of the Firefox browser failed to isolate frame locations. If a script opens an `iframe` and stores a pointer to its `document.location` object, then it continues to have access to this object even if the URL of the frame changes, because of a user action or a server redirection.

A second bug was in Firefox’s implementation of Content Security Policy (CSP) [38], a new mechanism to restrict loading of external contents to a authorized URIs. In its CSP, a website can ask for a report on all policy violations. If M sets its CSP to block all access to W , a frame on M gets redirected to W , M would be notified of this violation by the browser. A bug in Firefox caused the violation report to include the full URL (including fragment identifier) of the redirection, despite W and M being different origins.

By themselves, these bugs do not seem very serious; they only allow adversaries to read URIs, not even page contents, on frames that the adversary himself has

created. However, when combined with protocols like OAuth that use HTTP redirection to transmit secret tokens in URIs, these bugs become quite serious. For example, a malicious website M can steal a user’s Facebook token by creating an OAuth iframe with W ’s client id and reading the token in the redirected Facebook URI.

We reported these bugs and they are now fixed, but they highlight the difficulty of implementing a consistent policy across an increasing number of browser features.

Breaking Origin Authentication in `FB.login`. Although the OAuth iframe only obtains access tokens for an authorized origin W and the Proxy iframe only releases access tokens to the origin in its fragment identifier, there is no check guaranteeing that these origins are the same. Suppose a malicious website M opened the OAuth iframe with W ’s client id, but a Proxy iframe with M ’s origin. The OAuth iframe duly gets the token for W and passes it to the Proxy iframe that forwards the token to M . Hence, M has stolen the user’s access token for an arbitrary W .

We reported this bug and Facebook quickly addressed the attack by adding code for origin agreement between the two frames. However, we found two other ways to bypass this origin comparison by exploiting bugs in the component’s URL parsing functions.

Sub-origin Attacks on Facebook Clients. The design of the Facebook login component protects against cross-origin attackers (e.g. an unauthorized host website) but not provide any protections against untrusted content and ordinary web vulnerabilities on authorized host websites.

We found that Wordpress and Dropbox both allow users to host HTML pages on subdomains; we were able to exploit this feature to write user content that obtained access tokens meant for the main website. We also found an open redirector on the electronic voting site Helios that allowed any malicious website to steal a user’s access token for Helios; the website could then vote in the user’s name. This was a bug, but similar redirectors appear by design on Yahoo search and Bitly, leading to token theft, as shown in previous work [6].

These attacks were reported and are now prevented by either moving user content to a different domain or by ensuring that Facebook only releases tokens to a distinct subdomain (e.g. `open.login.yahoo.com`). However, pages on the main website still need to be given the token so that they can access the Facebook profile of the user. We found that websites like Wordpress and Hulu leave their Facebook access tokens embedded in their web-pages, where they may be read by any number of other scripts, including competing social plugins from Twitter, framework libraries like jQuery, and advertising and analytics libraries from Google and others. At their most benign, these scripts could read the access token to track

Facebook users; if they were malicious, they could impersonate the user and read her Yahoo mail or exfiltrate her full social profile for advertising use.

2.2 Client-side Decryption for Cloud Data

Web applications often use cryptography to protect sensitive user data that may be stored on untrusted servers or may pass through untrusted browsers. A typical example is a cloud-based file storage service, where both users and server owners would prefer the cloud server not to be able to read or modify any user file. To be host-proof in this way, all user files are stored encrypted in the cloud, using keys that are known only to the user or her browser, but not to the storage service. All plaintext data accesses are performed in the browser, after downloading and decrypting ciphertext from the cloud. This architecture has also been adopted by password managers and other privacy conscious applications such as electronic voting, encrypted chats, and conference management.

There are many challenges in getting browser-based cryptographic solutions right, but the two main design questions are how to trust the cryptographic library and protect its execution, and how to store encryption keys securely. Our survey found a variety of choices:

Browser Extensions. Password managers are often implemented as browser extensions so that they can read and write into login forms on webpages while being isolated from the page. Communication between the website and the page uses a browser-specific messaging API. We found attacks on the 1Password and RoboForm extensions where a malicious website could use this API to steal user passwords for trusted websites by exploiting buggy URL parsing and the lack of metadata integrity in the encrypted password database format.

Bookmarklets. Some password managers offer login bookmarklets that contain JavaScript code with an embedded encryption key that users can download and store in their browsers. When the bookmarklet is clicked on the login page of a website, its code is injected into the page; it retrieves encrypted login data from the password manager website, decrypts it, and fills in the login form. Even if the bookmarklet is accidentally clicked on a malicious page that tampers with the JavaScript builtin objects and pretends to be a different website, the bookmarklet is meant to at most reveal the user’s password for the current site. Indeed, several bookmarklets modified their designs to guarantee this security goal in response to previously found attacks [1]. However, we found several new attacks on a number of these fixed bookmarklets that still enabled malicious websites to steal passwords, the bookmarklet encryption key, and even the user’s master encryption key.

Website JavaScript. Cloud storage services and cryptographic web applications use JavaScript in the webpage to decrypt and display files downloaded from the cloud. Some of them (e.g. ConfiChair) use Java applets to implement cryptography whereas others (e.g. Mega) rely on reputed JavaScript libraries such as SJCL [37]. However, storing encryption keys securely during an ongoing session remains an open challenge. ConfiChair stores keys in HTML5 `localStorage`; SpiderOak stores keys for shared folders on the server, and Wuala stores encryption keys in a hidden user file on the client. We found a CSRF attack on SpiderOak, a client-side bug on Wuala, and an XSS attack on ConfiChair, all three of which allowed malicious websites to steal a user’s encryption keys if the user visited the website when logged into the corresponding web application.

2.3 Summary

All the attacks described in this survey were responsibly disclosed; most were found first by us and fixed on our suggestion; a few were reported by us in previous work [5, 6, 10]; some were reported and fixed independently. Our survey is not exhaustive, and many of the attack vectors we employed are quite well-known. While finding exploits on individual components took time and expertise, the ease with which we were able to find web vulnerabilities on which we built these exploits was surprising. In many cases, these vulnerabilities were not considered serious until we showed that they enabled unintended interactions with specific security components.

On the evidence of our survey, eliminating all untrusted contents and other web vulnerabilities from hosting websites seems infeasible. Instead, security components should seek to defend themselves against both malicious websites and same-origin attackers on trusted websites. Moreover, security checks in JavaScript components are hard to get right, and a number of our attacks relied on bugs in that part of the application logic. This motivates a more formal and systematic approach to the analysis of security-sensitive components.

3 DJS: Defensive JavaScript

In this section we define DJS, a subset of JavaScript that enforces a strict defensive programming style using language restrictions and static typing. DJS makes it possible to write JavaScript security components that preserve their behavior and protect their secrets even when loaded into an untrusted page after other scripts have tampered with the execution environment.

We advocate using DJS only for security-critical code; other code in the component or on the page may remain in full JavaScript. Hence, our approach is more suited to

our target applications than previous proposals that seek to restrict untrusted code (e.g. [16, 26, 39, 40] or require trusted code to run first (e.g. [2]).

The rest of the section informally describes the DJS subset and its security properties; full formal definitions can be found in the technical report [11].

3.1 Defensiveness

The goal of defensiveness is to protect the behavioral integrity of sensitive JavaScript functions that will be invoked in an environment where arbitrary adversarial code has already run. How do we model the capabilities of an adversary who may be able to exploit browser and server features that fall outside JavaScript, such as frames, browser extensions, REST APIs, etc?

We propose a powerful attacker model inspired by the successful Dolev-Yao attacker [18] for cryptographic protocols, where *the network is the attacker*. In JavaScript, we claim that *the memory is the attacker*. We allow the attacker to arbitrarily change one (well-formed) JavaScript memory into another, thus capturing even non-standard or undocumented features of JavaScript.

Without further assumptions, this attacker is too powerful to state any property of trusted programs. Hence, like in the Dolev-Yao case where the attacker is assumed unable to break encryption, we make the reasonable assumptions that the attacker cannot forge pointers to memory locations it doesn't have access to, and that it cannot break into the scope frames of functions. This assumption holds *in principle* for all known JavaScript implementations, but in practice it may fail to hold because of use-after-free bugs or prototype hijacking attacks [22].

Let a heap be a map from memory locations to language values, including locations themselves (like pointers). We often reason about equivalent heaps up to renaming of locations and garbage collection (removal of locations unreachable from the native objects). Let an *attacker memory* be any well-formed region of the JavaScript heap containing at least all native objects required by the semantics, and without any dangling pointer. Let a *user memory* be any region of the JavaScript heap that only contains user-defined JavaScript objects. A user memory may contain pointers to the attacker memory. Let *attacker code* and *user code* be function objects stored respectively in the attacker and user memories.

Assumption 1 (Memory safety). *In any reasonable JavaScript semantics, starting from a memory that can be partitioned in two regions, where one is an attacker memory and the other a user memory, the execution of attacker code does not alter the user memory.*

User code cannot run in user memory alone because it

lacks native objects and default prototypes necessary for JavaScript executions. For that reason, we consider user code that exposes an API in the form of a function that may be called by the attacker. Let a *function wrapper* be an arbitrary JavaScript expression E parametric in a function definition F , which returns a wrapped function G_F . G_F is meant to safely wrap F , acting as a proxy to call F . For example:

```
1 E = (function() {
2   var F = function(x) {
3     var secret = 42, key = 0xCOCOACAFE;
4     return x === key ? secret : 0
5   }
6   return function G_F(x) { return F(x>>>0) }
})()
```

We now informally define the two properties that capture defensiveness of function wrappers:

Definition 1 (Encapsulation). A function wrapper E encapsulates F over domain \mathcal{D} if no JavaScript program that runs E can distinguish between running E with F and running E with an arbitrary function F' without calling the wrapped function G_F . Moreover, for any tuple of values $\tilde{v} \in \mathcal{D}$, the heap resulting from calling $G_F(\tilde{v})$ is equivalent to the heap resulting from calling $F(\tilde{v})$.

In other words, encapsulation states that an attacker with access to G_F should not learn anything more about F than is revealed by calling F on values from \mathcal{D} . For example, if the above E encapsulates the oracle F (lines 2-4) on numbers, an attacker may not learn `secret` unless it is returned by F , even by trying to tamper with properties of G_F such as `arguments`, `callee`...

The next property describes the integrity of the the input-output behavior of defensive functions:

Definition 2 (Independence). A function wrapper E preserves the *independence* of F if any two sequences of calls to G_F , interleaved with arbitrary JavaScript code, return the same sequence of values whenever corresponding calls to G_F received the same parameters and no call to G_F triggered an exception.

This property is different from *functional purity* [19]: since F may be stateful, it is not enough to enforce single calls to G_F to return the same value as arbitrary call sequences must yield matching results. Note that G_F is not prevented by this definition from causing side-effects on its execution environment. For example, E given above can still satisfy independence even though it will cause a side effect when G_F is passed as argument the object `{valueOf:function(){window.leak=42;return 123}}`.

The above F (lines 2-4) returns its secret only when passed the right key, and does not cause observable side-effects. If E encapsulates F over numbers and preserves its independence, then an attacker may not learn this secret without knowing the key.

```

⟨djs-program⟩ ::= ‘(function() {
    ‘var _ = ⟨function⟩ ;
    ‘return function(x) {
        ‘if(typeof x == “string”) return _(x) ;
    }})() ;’

⟨function⟩ ::=
| ‘function’ (@identifier ‘,’)* ‘{’
| ‘var’ (@identifier (= ‘⟨expression⟩)? ‘,’)+?
| ⟨statement⟩ ‘;’)*
| ⟨return⟩ ⟨expression⟩? ‘}’

⟨statement⟩ ::= ε
| ‘with’ ⟨lhs_expression⟩ ‘;’ ⟨statement⟩
| ‘if’ ⟨expression⟩ ‘;’ ⟨statement⟩
| ‘else’ ⟨statement⟩)?
| ‘while’ ⟨expression⟩ ‘;’ ⟨statement⟩
| ‘{’ ⟨statement⟩ ‘;’}* ‘}’
| ⟨expression⟩

⟨expression⟩ ::= ⟨literal⟩
| ⟨lhs_expression⟩ ‘(’ ⟨expression⟩ ‘,’)* ‘)’
| ⟨expression⟩ ⟨binop⟩ ⟨expression⟩
| ⟨unop⟩ ⟨expression⟩
| ⟨lhs_expression⟩ ‘=’ ⟨expression⟩
| ⟨dyn_accessor⟩
| ⟨lhs_expression⟩

⟨lhs_expression⟩ ::=
| @identifier | ‘this.’ @identifier
| ⟨lhs_expression⟩ ‘[’ @number ‘]’
| ⟨lhs_expression⟩ ‘.’ @identifier

⟨dyn_accessor⟩ ::=
| ⟨(x) = @identifier⟩ ‘[⟨expression⟩
‘>> 0] % ⟨x⟩ .length ]’
| ‘⟨(y) = @identifier⟩ >>>0) <’ ⟨(x) = @identifier⟩
‘.length ? x[y] : ’ @string
| @identifier ‘[’ ⟨expression⟩ ‘&’ (n=@number ‘]’
n ∈ [1,230 - 1]

⟨literal⟩ ::= ⟨function⟩
| ‘{’ (@identifier ‘:’ ⟨expression⟩ ‘,’)* ‘}’
| ‘[’ ⟨expression⟩ ‘,’)* ‘]’
| @number | @string | @boolean

⟨binop⟩ ::= ‘+’ | ‘-’ | ‘*’ | ‘/’ | ‘%’
| ‘&’ | ‘|’ | ‘^’ | ‘>>’ | ‘<<’ | ‘>>>’
| ‘&&’ | ‘||’ | ‘==’ | ‘!=’ | ‘>’ | ‘<’ | ‘>=’ | ‘<=’

⟨unop⟩ ::= ‘+’ | ‘-’ | ‘!’ | ‘~’

```

Figure 3: DJS Syntax.

Since in practice an attacker can set up the heap in such a way that calling G_F will raise an exception (e.g. stack overflow) regardless of the parameters passed to G_F , independence only considers sequences of calls to G_F that do not trigger exceptions in G_F . When an exception occurs in G_F , the attacker may gain access to a stack trace. Even though stack traces only reveal function names and line numbers in current browsers, we prevent this information leak by always executing E within a `try` block.

3.2 DJS Language

In practice, JavaScript code is considered valid DJS if it is accepted by the automatic conformance checker described in Section 4.1, which in turn is based on the type system of Section 3.3. The type system effectively imposes a restricted grammar on DJS that is given in Figure 3. In this section, we describe the language more informally.

Besides defensiveness, the main design goals for DJS are: automated conformance checking (by typing), compatibility with currently deployed browsers (supporting ECMAScript 3 and 5), and minimal performance overhead. A side effect of our type system is to impose hygienic coding practices similar to those of the popular JSLint tool, encouraging high quality code that is easy to reason about and extract verifiable models from.

Programs. A DJS *program* is a function wrapper (in the sense of Definitions 1 and 2); its public API consists of a single stub function from string to string that is a proxy to a function (stored in a variable “`_`”) in its closure. We denote this wrapper by E_{DJS} :

```

1 (function(){
2     var _ = <function>;
3     return function(x){
4         if(typeof x == “string”) return _(x)
5     })();

```

For simplicity, functions must begin with all their local variables declarations, and end with a return statement:

```

1 function (<id>, ..., <id>){
2     var <id> = <expr>, ..., <id> = <expr>;
3     <statements>
4     return <expr>}

```

Our type system further restricts DJS statements and expressions as described below.

Preventing External References. DJS programs may not access variables or call functions that they do not define themselves. For example, they may not access DOM variables like `document.location`, call global functions like `encodeURIComponent`, or access prototype functions of native objects like `String.index0f`.

This restriction follows directly from our threat scenario, where every object not in the defensive program is in attacker memory and may have been tampered with. So, at the very least, values returned by external references must be considered tainted and not used in defensive computations to preserve independence. More worryingly, in JavaScript, an untrusted function that is called by defensive code can use the `caller` chain starting from its own `arguments` object to traverse the call stack and obtain direct pointers to defensive objects (inner functions, their `arguments` objects, etc.), hence breaking encapsulation. Some countermeasures have been proposed to pro-

tect against this kind of stack-walking, but they rely on non-standard browser features and are not very reliable (e.g. we discovered a flaw against the countermeasure in [21]: trying to set the caller property of a function to null fails, an issue immediately fixed by the authors in their online version). Future versions of JavaScript may prohibit stack-walking, but in current browsers our restriction is the prudent choice.

To enforce this restriction, the type system requires all variables used in a DJS program to be lexically scoped, within a function or scope object. For example, `var s = {x:42}; with (s){x = 4;}` is valid DJS code, but `x = 4` is not.

Preventing Implicit Function Calls. In JavaScript, non-local access can arise for example from its non-standard scoping rules, from the prototype-based inheritance mechanism, from automated type conversion and from triggering getters and setters on object properties.

Hence, to prevent defensive code from accidentally calling malicious external functions, DJS requires all expressions to be statically typed. This means that variables can only be assigned values of a single type; arrays have a fixed non-extensible number of (same-typed) values; objects have a non-extensible set of (typed) properties. Typing ensures that values are only accessed at the right type and that objects and arrays are never accessed beyond their boundaries (preventing accidental accesses to prototypes and getters/setters). To prevent automatic type conversion, overloaded operators (e.g. `+`) must only be used with arguments of the same type.

Due to these restrictions, there is no general computed property access `e[e]` in the syntax. Instead, we include a variety of *dynamic accessors* to enable numeric, within-bound property access to arrays and strings using built-in dynamic checks, such as `x[(e>>0)%x.length]`.

DJS also forbids property enumeration `for(i in o)`, constructors and prototype inheritance.

Preventing Source Code Leakage. The source code of a DJS program is considered secret, and should not be available to untrusted code. We identify four attack vectors that a trusted script can use to read (at least part of) the source code of another script in the same origin: using the `toSource` property of a function, using the `stack` property of an exception, reading the code of an inline script from the DOM, or re-loading a remote script as data using AJAX or Flash.

To avoid the first attack, DJS programs only export stub functions that internally call the functions whose source code is sensitive. Calling `toSource` on the former only shows the stub code and does not reveal the source code of the latter. As discussed at the end of Section 3.1, we can avoid the second attack by running wrapped DJS code within a `try` block. To avoid the third and fourth

Types and Environments.

$\langle\tau\rangle ::= \text{number} \mid \text{boolean} \mid \text{string} \mid \text{undefined}$	Base types
$\tilde{\tau} \rightarrow \tau$	Function
$\tilde{\tau}[\rho] \rightarrow \tau$	Method operating on properties ρ
δ	Objects and arrays
$\langle\delta\rangle ::= \sigma \mid \sigma^*$	Extensible or Fixed types
$\langle\sigma\rangle ::= \rho \mid [\tau]_n, n \in \mathbb{N}$	Array of length n
$\langle\rho\rangle ::= \{x_1 : \tau_1, \dots, x_n : \tau_n\}$	Object with fields $x_1 \dots x_n$
$\langle\kappa\rangle ::= \text{s} \mid \text{o}$	Scope kind
$\langle\Phi\rangle ::= \varepsilon \mid \Phi, x : \tau$	Scope frame
$\langle\Gamma\rangle ::= \varepsilon \mid \Gamma, [\Phi]_K$	Typing environment
$[\sigma^* \text{ and } \sigma \text{ are same thing sometimes}]$	

Subtyping.

$$\frac{\tau <: \tau}{\sigma^* <: \tau} \quad \frac{\sigma <: \tau}{\sigma^* <: \tau} \quad \frac{m \leq n}{[\tau]_n <: [\tau]_m} \quad \frac{J \subseteq I}{\{x_i : \tau_i\}_{i \in I} <: \{x_j : \tau_j\}_{j \in J}}$$

$$\frac{v_1 <: v_2 \quad \tilde{\mu}_2 <: \tilde{\mu}_1}{\tilde{\mu}_1 \rightarrow v_1 <: \tilde{\mu}_2 \rightarrow v_2} \quad \frac{\rho_2 <: \rho_1 \quad \tilde{\mu}_1 \rightarrow v_1 <: \tilde{\mu}_2 \rightarrow v_2}{\tilde{\mu}_1[\rho_1] \rightarrow v_1 <: \tilde{\mu}_2[\rho_2] \rightarrow v_2}$$

Figure 4: DJS types, subtyping and environments.

attacks, we advise that a defensive script should never be directly inlined in a page; it may either be injected and executed by a bookmarklet or browser extension, or else it should be sourced from a dedicated secure origin that does not allow cross-domain resource sharing.

From Coding Discipline to Static Analysis. DJS imposes a number of seemingly harsh restrictions on security component developers, but most of these are motivated by the hostile environments in which these components must execute, and the strict coding discipline pays dividends in static analysis. In Sections 5 and 6, we show that despite these restrictions, it is still possible to code large security components in DJS that enjoy strong defensiveness guarantees and can be automatically analyzed for security.

3.3 Type System

DJS types and their subtyping relation are defined in Figure 4. In addition to the JavaScript base types, it includes functions, methods, arrays and objects. Method types require a type ρ for the `this` parameter. Arrays are indexed by a lower bound n on their size.

The type system of DJS is static, that is, new variables must be initialized with a value of some type, and once a type is assigned to a variable it cannot subsequently change. A standard width-subtyping relation $<:$ captures polymorphism in the length of arrays and the set of properties of objects. However, fixed types σ^* do not have subtypes to guarantee soundness [14, 15, 33]. For example, our type systems does not admit a type for the term `(function(x,y){x[0]=y; return true;})([[1]],[])`.

Typing environments Γ reflect the nesting of the lexical scoping up to the expression that is being typed. Each

$\text{Obj} \frac{\Gamma \vdash e_i : \tau_i \quad i \in [1..n]}{\Gamma \vdash \{x_1 : e_1, \dots, x_n : e_n\} : \{x_i : \tau_i\}_{i \in [1..n]}^*}$	$\text{PropA} \frac{\Gamma \vdash e : \delta \quad \delta <: \{x : \tau\}}{\Gamma \vdash e.x : \tau}$	$\text{ArrA} \frac{\Gamma \vdash e : \delta \quad \delta <: [\tau]_{n+1}}{\Gamma \vdash e[n] : \tau}$
$\text{Arr} \frac{\Gamma \vdash e_i : \tau \quad i \in [1..n]}{\Gamma \vdash [e_1, \dots, e_n] : [\tau]^*_n}$	$\text{StrD} \frac{\Gamma \vdash x : \text{string} \quad \Gamma \vdash y : \text{number}}{\Gamma \vdash ((y \ggg 0) < x.\text{length}? x[y] : @\text{string}) : \text{string}}$	$\text{ArrD} \frac{\Gamma \vdash x : [\tau]_n \quad \Gamma \vdash e : \text{number} \quad n > 0}{\Gamma \vdash x[(e \ggg 0)\%x.\text{length}] : \tau}$
$\text{Scope} \frac{\Phi(x) = \tau}{\Gamma, [\Phi]_k \vdash x : \tau}$	$\text{RecScope} \frac{x \notin \text{dom}(\Phi) \quad \Gamma \vdash x : \tau}{\Gamma, [\Phi]_s \vdash x : \tau}$	$\text{FunDef} \frac{\Gamma, [\tilde{x} : \tilde{\alpha}, (y_i : \mu_i)_{i < j}]_s \vdash e_j : \mu_j \quad j \in [1..m] \quad \Gamma, [\tilde{x} : \tilde{\alpha}, \tilde{y} : \tilde{\mu}]_s \vdash s : \text{undefined} \quad \Gamma, [\tilde{x} : \tilde{\alpha}, \tilde{y} : \tilde{\mu}]_s \vdash r : \tau}{\Gamma \vdash \text{function } (\tilde{x})\{\text{var } y_1 = e_1, \dots, y_m = e_m; s; \text{return } r\} : \tilde{\alpha}[\rho] \rightarrow \tau}$
$\text{Assign} \frac{\Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1 = e_2 : \tau}$	$\text{With} \frac{\Gamma \vdash e : \{\tilde{x} : \tilde{\tau}\} \quad \Gamma, [\tilde{x} : \tilde{\tau}]_o \vdash s : \text{undefined}}{\Gamma \vdash \text{with}(e)s : \text{undefined}}$	$\text{MetDef} \frac{\Gamma \vdash \text{function } (\text{this}, \tilde{x})\{s\} : (\rho, \tilde{\alpha}) \rightarrow \tau}{\Gamma \vdash \text{function } (\tilde{x})\{s\} : \tilde{\alpha}[\rho] \rightarrow \tau}$
$\text{FunCall} \frac{\Gamma \vdash e : \mu \quad \Gamma \vdash \tilde{e} : \tilde{\alpha} \quad \mu <: \tilde{\alpha} \rightarrow \tau}{\Gamma \vdash e(\tilde{e}) : \tau}$		$\text{MetCall} \frac{\Gamma \vdash e : \mu \quad \Gamma \vdash \tilde{e} : \tilde{\alpha} \quad \mu <: \{x : \tilde{\alpha}[\rho] \rightarrow \tau\}}{\Gamma \vdash e.x(\tilde{e}) : \tau}$

Figure 5: Selected typing rules.

scope frame Φ contains bindings of identifiers to types, and is annotated with s or o depending on whether the corresponding scope object is an activation record created by calling a function, or a user object loaded onto the scope using `with`. This distinction is important to statically prevent access to prototype chains: unlike activation records, user objects cause a missing identifier to be searched in the (untrusted) object prototype rather than in the next scope frame; thus, scope resolution must stop at the first frame of kind o .

Typing Rules. Most of our typing rules are standard; here we only discuss a few representative examples, reported in Figure 5; the other typing rules are detailed in the full version [11]. For soundness, Rule Assign does not allow subtyping. Rule Obj keeps the object structure intact and only abstracts each e_i into its corresponding type τ_i . The rule for accessors and dynamic accessors ensure that the property being accessed is directly present in the corresponding string, array or object. For example, to typecheck $\Gamma \vdash s[3] : \text{number}$ using rule ArrA, s must be typeable as an array of at least 4 numbers. The rules for dynamic accessors benefit from knowing that the index is a number modulo the size of admissible index values. Rule RecScope looks up variables recursively only through activation records, as explained above. Rule With illustrates the case when an object frame is added to the typing environment. The FunDef typing rule is helped by the structure we impose on the function body. It adds an activation record frame to the typing environment and adds all the local variable declarations inductively. Finally, it typechecks the body statement s and the type of the return expression r . Rule MetDef invokes rule FunDev after adding a formal `this` parameter to the function and extending the input type with the `this` type ρ . Rule FunCall is standard, whereas rule MetCall forces an explicit syntax for method invocation in order to determine the type ρ and binding of `this`.

In particular, ρ must be such that method l has a function type compatible with the potentially more general type of its parent object l .

Formal Guarantees. The DJS type system enjoys both *type soundness* (types are preserved by computation) and *progress* (typed programs terminate with a final value and do not raise exceptions). A consequence of type soundness is that well-typed programs are defensive. All formal definitions and proofs leading to Theorem 1 can be found in the technical report [11].

Theorem 1 (Defensiveness). *If $\emptyset \vdash F : \text{string} \rightarrow \text{string}$ then the DJS wrapper E_{DJS} encapsulates F over strings and preserves its independence.*

Another consequence of type soundness is that the execution of well-typed programs does not affect attacker memory [11]. As a consequence, execution of DJS programs is invisible to the attacker.

Extensions. We do not claim that DJS is the maximal defensive subset of JavaScript: with a more expressive type system, it would for instance be possible to support one level of prototype inheritance (i.e. constructors having a literal object as prototype), or avoid certain dynamic accessors. Because we expect that DJS components will mostly consist of basic control flow and calls to our libraries, we do not think more expressive defensive subsets of JavaScript are necessary for our goals.

4 DJS Analysis Tools

We developed two analysis tools for DJS programs. The first verifies that a JavaScript program conforms to DJS. The second extracts applied pi calculus models from DJS programs, so that they may be verified for security properties. For lack of space, we do not detail the implementation of these tools; both are available from our website.

```

# ./djs --check
x = function(s){return s.split(",")}; x("a,b");
Cannot type the following expression at file <stdio>,
line 1:38 to 1:46: x("a,b")
type <{"split":(string) -> 'a}> was expected but got <string>.

# ./djs --pv >model.pv && proverif -lib djcl model.pv
(function(){ var mackey = _lib.secret("xxx")+"";
var _ = function(s){return _lib.hmac(s,mackey)};
return function(s){if(typeof s=="string") return _(s)}}

Typing successful, CPU time: 4ms.
--- Free variables ---
_lib:{`hmac`:(string,string)->string,"secret":string->string}
Process:
{1}new fun_9: channel;
(
{2}!
{3}in(fun_9, ret_10: channel);
{4}new var_mackey: Memloc;
{5}let s_11: String = str_1 in

```

Figure 6: Screenshot of the DJS tool: first a type-checking error, then a (cut off) ProVerif translation.

4.1 Conformance Checker

We implement fully automatic type inference for the DJS type system. Our tool can check if an input script is valid DJS and provides informative error messages if it fails to typecheck. Figure 6 shows a screenshot with a type error and then the correct inferred type.

In our type system, an object such as `{a:0, b:1}` can be assigned multiple types: `{a:number,b:number}`, `{a:number}`, `{b:number}` or `{}`. Subtyping induces a partial order relation on the admissible types of an expression; the goal of type inference is to compute the maximal admissible type of a given expression.

To compute this type, we implement a restricted variant of Hindley–Milner inference that incorporates width subtyping and infers type schemes. For example, the generalized type for the function `function f(x){return x[0]}` is $\exists \tau. [\tau]_1 \rightarrow \tau$. Note the existential quantifier in front of τ : function types are not generalized, which would be unsound because of mutable variables. Thus, if the type inference processes the term `f([1])`, unification will force $\tau = \text{number}$, and any later attempt to use `f(["a"])` will fail, while `f([1,2])` will be accepted.

The unification of object type schemes yields the union of the two sets of properties: starting from $x : \tau$, after processing $x.a + x.b$, unification yields $\tau = \{a : \tau_1, b : \tau_2\}$ and $\tau_1 = \tau_2$. Literal constructors are assigned their maximal, fixed object type $\{x_i : T_i\}_{i \in [1..n]}^*$. Unification of an object type $\{X\}$ with the fixed $\{x_i : T_i\}_{i \in [1..n]}^*$ ensures $X \subseteq \{x_i : T_i\}_{i \in [1..n]}$.

Our tool uses type inference as a heuristic, and relies on the soundness of the type checking rules of Section 3.3 for its correctness. Our inference and unification algorithms are standard. We refer interested readers to our implementation for additional details.

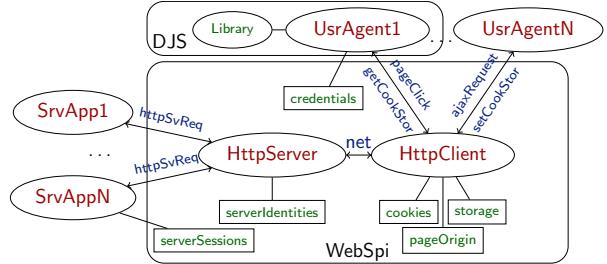


Figure 7: WebSpi model and DJS components

4.2 Model Extraction

DJS is a useful starting point for a security component developer, but defensiveness does not in itself guarantee security: for example it does not say that a program will not leak its secrets to the hosting webpage, say by exposing them in its exported API. Moreover, security components like those in Section 2 consist of several scripts exchanging encrypted messages with each other and with other frames and websites. Such designs are complex and prone to errors, analyzing their security thus requires a detailed model of cryptography, the browser environment and the web attacker.

In prior work, the WebSpi library of the ProVerif tool has been used to analyze the security of web applications [5, 6]. The main processes, channels and data tables of WebSpi are represented on Figure 7. `UsrAgent` processes model the behavior of JavaScript running on a page, while the other processes handle communications and processing of server requests.

The advantage of this methodology is that an application can be automatically verified against entire classes of web attackers. ProVerif can handle an unbounded number of sessions, but may fail to terminate. If it verifies a model, it can serve to increase confidence in the security application. The disadvantage is that to model a JavaScript component in WebSpi, a programmer normally has to write an applied pi calculus process for each script by hand.

We developed a model extraction tool that automatically generates user agent process models of components written in the subset of DJS without loops, using a process and data constructor library for cryptographic operations and serialization (matching our implemented DJS libraries introduced in the next section).

Our generated processes may then be composed with existing WebSpi models of the browser and (if necessary) hand-written models of trusted servers and automatically verified. To support our translation, we extended the WebSpi model with a more realistic treatment of JavaScript that allowed multiple processes to share the same heap.

We do not fully detail our translation from DJS to the

applied pi calculus here for lack of space; it follows Milner’s famous “functions as processes” encoding of the lambda calculus into the pi calculus [30]. Similar translations to ours have previously been defined (and proved sound) for F# [12] and Java [4]. Our translation only works for well-typed DJS programs that use our DJS libraries; it does not apply to arbitrary JavaScript.

DJS programs may prefix a function name by `_lib` to indicate that the code of certain functions should not be translated to applied pi and they must instead be treated as trusted primitives. A typical example is cryptographic functions, which get translated to symbolic functions.

Our translation recognizes two kinds of security annotations in source DJS programs. First, functions may be annotated with security events; for example, the expression `_lib.event(Send(a,b,x))` may be triggered before `a` uses a secret key shared with `b` to compute a MAC of `x`. Second, functions may label certain values as secrets `_lib.secret(x)`. Such annotations are reflected in the generated models and can be analyzed by ProVerif to prove authentication and secrecy queries; we describe complex components we verified in Section 6.

5 Defensive Libraries

In this section, we present defensive libraries for cryptography (DJCL), data encoding (DJSON), and JSON signature and encryption (JOSE). These libraries amount to about two thousand lines of DJS code, verified for defensiveness using our conformance checker. Hence, they can be relied upon even in hostile environments.

5.1 Defensive JavaScript Crypto Library

Our starting points for DJCL are two widely used JavaScript libraries for cryptography: SJCL [37] (covering hashing, block ciphers, encoding and number generation) and JSBN (covering big integers, RSA, ECC, key generation and used in the Chrome benchmark suite). We rewrote and verified these libraries in DJS.

Our implementation covers the following primitives: AES on 256 bit keys in CBC and CCM/GCM modes, SHA-1 and SHA-256, HMAC, RSA encryption and signature on keys up to 2048 bits with OAEP/PSS padding. All our functions operate on byte arrays encoded as strings; DJCL also includes related encoding and decoding functions (UTF-8, ASCII, hexadecimal, and base64).

We evaluated the performance of DJCL using the jsperf benchmark engine on Chrome 24, Firefox 18, Safari 6.0 and IE 9. We found that our AES block function, SHA compression functions and RSA exponentiation performed at least as fast as their SJCL and JSBN counterparts, and sometimes even faster. Defensive coding is well suited for bit-level, self-contained crypto com-

putations, and JavaScript engines can easily optimize our non-extensible arrays and objects.

On the other hand, when implementing high-level constructions such as HMAC or CCM encryption that operate on variable-length inputs, we pay a cost for not being able to access native objects in DJS. DJCL encodes variable-length inputs in strings, since it cannot use more efficient but non-defensive objects like `Int32Array`. Encoding and decoding UTF-8 strings without relying on a pristine `String.fromCharCode` and `String.charCodeAt` means that we need to use table lookups that are substantially more expensive than the native functions. The resulting performance penalty is highly dependent on the amount of encoding, the browser and hardware being used, but even on mobile devices, DJCL achieves encryption and hashing rates upwards of 150KB/s, which is sufficient for most applications. Of course, performance can be greatly improved in environments where prototypes of the primordial `String` object can be trusted (for instance, by using `Object.freeze` before any script is run).

5.2 Defensive JSON and JOSE

In most of our applications, the input string of a DJS program represents a JSON object; our DJSON library serializes and parses such objects defensively for the internal processing of such data within a defensive program.

`DJSON.stringify` takes a JSON object and a schema describing its structure (i.e. an object describing its DJS type) and generates a serialized string. Deserializing JSON strings generally requires the ability to create extensible objects. Instead, we rewrite `DJSON.parse` defensively by requiring two additional parameters: the first is a schema representing the shape of the expected JSON object; the second is a preallocated object of expected shape that will be filled by `DJSON.parse`. Our typechecker processes these schemas as type annotations and uses them to infer types for code that uses these functions.

This approach imposes two restrictions. Since DJS typing fixes the length of objects, our library only works with objects whose sizes are known in advance. This restriction may be relaxed by using extensions of DJS (described in our technical report [11]) that use algebraic constructors for extensible objects and arrays. Also, at present, we require users of the DJSON library to provide the extra parameters (schemas, preallocated objects), but we plan to extend our conformance checker to automatically inject these parameters based on the inferred types of the serialized and parsed JSON objects.

Combining DJCL and DJSON, we implemented a family of emerging IETF standards for JSON cryptography (JOSE), including JSON Web Tokens (JWT) and JSON Web Encryption (JWE) [25]. Our library interoperates with other server-side implementations of JOSE

Program	LOC	Typing	PV LOC	ProVerif
DJCL	1728	300ms	114	No Goal
JOSE	160	36ms	9	No Goal
Sec. AJAX	61	7ms	243	12s
LastPass	43	42ms	164	21s
Facebook	135	42ms	356	43s
ConfChair	80	31ms	203	25s

Table 2: Evaluation of DJS codebase

(notably those implementing OpenID Connect). Using JOSE, we can write security components that exchange encrypted and/or authenticated AJAX requests and responses with trusted servers. More generally, we can build various forms of secure RPC mechanisms between a DJS script and other principals (scripts, frames, browser extensions, or servers.)

6 Applications

We revisit the password manager bookmarklet, single sign-on script, and encrypted storage website examples from Section 2 and evaluate how DJS can help avoid attacks and improve confidence in their security. For each component, we show that DJS can achieve security goals even stronger than those currently believed possible using standard browser security mechanisms. Table 2 summarizes our codebase and verification results.

6.1 Secret-Keeping Bookmarklets

Bookmarklets are fragments of JavaScript stored in a bookmark that get evaluated in the scope of the active page when they are clicked. Password manager bookmarklets (like LastPass Login, Verisign One-Click, Passpack It) contain code that tries to automatically fill in login forms (or credit card details) on the current page, by retrieving encrypted data the user has stored on the password manager’s web server.

For example, the LastPass server authenticates the user with a cookie (she must be currently logged in), authenticates the host website with the Referer or Origin header, and returns the login data encrypted with a secret key (`LASTPASS_RAND`) that is unique to the bookmarklet and embedded in its code. The bookmarklet then decrypts the login data with its key and fills in the login form.

The code in these bookmarklets is typically not defensive against same origin attacks; this leads to a family of *rootkit* attacks, where a malicious webpage can fool the bookmarklet into revealing its secrets [1]; indeed, we found new variations of these attacks (Section 2) even after the original designs were fixed to use frames.

We wrote two, improved versions of the LastPass bookmarklet using DJS that prevent such attacks:

- The first uses DJCL’s AES decryption to decrypt the login data retrieved from the LastPass server.
- The second uses DJCL’s HMAC function to authenticate the bookmarklet (via `postMessage`) to a frame loaded from the LastPass origin; the frame then decrypts and reveals the login data to the host page.

Assuming the host page is correctly authenticated by LastPass, both designs prevent rootkit attacks.

Moreover, both our bookmarklets guarantee a stronger *click authentication* property. The bookmarklet key represents the intention of the user to release data to the current page. If a script on the page could capture this key, it would no longer need the bookmarklet; it could use the password manager server directly to track (and login) the user on subsequent visits, even if the user wished to remain anonymous, and say had erased her cookies for this site. Instead, by protecting the key using DJS, and using the key only once per click, both our designs guarantee that the user must have clicked on the bookmarklet each time her identity and data is released to the webpage.

Evaluation. Our bookmarklets are fully self-contained DJS programs and with a trimmed-down version of DJCL can fit the 2048 bytes length limit of bookmarklets. They require minimal changes to the existing LastPass architecture. More radical redesigns are possible, but even those would benefit from being programmed in DJS. We verified our bookmarklets for defensiveness by typing, and for key secrecy and click authentication by using ProVerif. In ProVerif, we compose the models extracted from the bookmarklets with the WebSpi library and a hand-written model for the LastPass server (and frame).

Click authentication is an example of a security goal that requires DJS; it cannot be achieved using frames for example. The reason is that bookmarklets (unlike browser extensions) cannot reliably create or communicate with frames without their messages being intercepted by the page. They need secrets for secure communication; only defensiveness can protect their secrets.

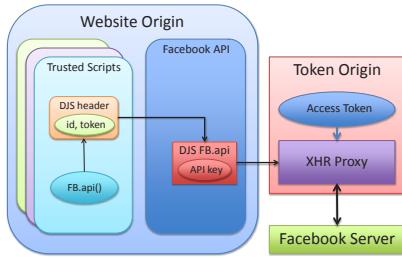
6.2 Script-level Token Access Control

The Facebook login component discussed in Section 2 keeps a secret access token and uses it to authenticate user data requests to the Facebook REST API. However, this token may then be used by any script on the host website, including social plugins from competitors like Twitter and Google, and advertising libraries that may track the user against her wishes. Can we restrict the use of this access token only to selected scripts, say only (first-party) scripts loaded from the host website? Browser-based security mechanisms, like iframes, cannot help, since they operate at the origin level. Even CSP

policies that specify which origins can provide scripts to a webpage cannot differentiate between scripts once they are loaded into the page.

We propose a new design that uses DJS to enforce fine-grained script-level access control for website secrets like access tokens and CSRF tokens. We implement it by modifying the Facebook JavaScript SDK as follows.

We assume that the website has registered a dedicated *Token Origin* (e.g. open.login.yahoo.com) with Facebook where it receives the access token. We assume that the token is obtained and stored securely by this origin.



The token origin then provides a proxy frame to the main website (e.g. *.yahoo.com) that only allows authorized scripts to use the token. The frame listens for requests signed with JWT using an API key; if the signature is valid, it will inject the access token into the request and forward it to the network (using XHR, or JSONP for Facebook), and return the result. An useful extension to this mechanism when privacy is important is to accept encrypted JWE requests and encrypt their result (we leave this out for simplicity).

On the main website, we use a slightly modified version of the Facebook SDK that has no access to the real access token, but still provides the same client-side API to the webpage. We replace the function that performs network requests (`FB.api`) with a DJS function that contains the secret API key, hence can produce signed requests for the proxy frame. This function only accepts requests from pre-authorized scripts; it expects as its argument a serialized JSON Web Token (JWT) that contains the request, an identifier for the source script, and a signature with a script-specific key (in practice, derived from the API key and the script identifier). If the signature is valid, the API request is signed with the API key and forwarded to the proxy frame. This function can also enforce script-level access control; for instance, it may allow cross-origin scripts to only request the user name and profile picture, but not to post messages.

For this design to work, the API key must be fresh for each user, which can be achieved using the user's session or a cookie. Such keys should have a lifetime limit corresponding to the cache lifetime of the scripts that are injected with secret tokens. One may also want to add

freshness to the signed requests to avoid them being replayed to the proxy frame.

Finally, each (trusted) script that requires access to the Facebook API is injected with a DJS header that provides a function able to sign the requests to `FB.api` using its script identifier and a secret token derived from the identifier and API key. We provide a sample of the DJS code injected into trusted scripts below, for basic Facebook API access (`/me`) with no (optional) parameters. Note that only the `sign_request` function is defensive; we put it in the scope of untrusted code using `with` because it prevents the call stack issues of closures:

```

1 with({sign_request: (function(){
2     var djcl = /*...*/;
3     var id = "me.js", tok = "1f3c...";
4     var _ = function(s){
5         return s == "/me" /* || s== "..." */ ?
6             djcl.jwt.create(
7                 djcl.dj.json.stringify({jti: id, req: s}), tok
8             ) : "";
9         return function(s){
10            if(typeof s=="string") return _(s)
11        })(), __proto__:null})
12 {
13     // Trusted script
14     FB.api(sign_request("/me"),
15         function(r){alert("Hello, "+r.name)});
16 }

```

Evaluation. Besides allowing websites to keep the access token secret, our design lets them control which scripts can use it and how (a form of *API confinement*). Of course, a script that is given access to the API (via a script key) may unintentionally leak the capability (but not the key), in which case our design allows the website to easily revoke its access (using a filter in `FB.api`). Our proposal significantly improves the security of Facebook clients, in ways it would be difficult to replicate with standard browser security mechanisms.

We only change one method from the Facebook API which accounts for less than 0.5% of the total code. Our design maintains DOM access to the API, which would be difficult to achieve with frames. Without taking DJCL into account, each of the DJS functions added to trusted scripts is less than 20 lines of code. We typechecked our code for defensiveness, and verified with ProVerif that it provides the expected script-level authorization guarantees, and that it does not leak its secrets (API key, script tokens) to the browser.

6.3 An API for Client-side Encryption

In Section 2 we showed that encrypted cloud storage applications are still vulnerable to client-side web attacks like XSS (e.g. ConfiChair, Mega) that can steal their keys and completely break their security. Finding and eliminating injection attacks from every page is not always

easy or feasible. Instead, we propose a robust design for client-side crypto APIs secure despite XSS attacks.

First, we propose to use a defensive crypto library rather than Java applets (Helios, Wuala, and ConfiChair) or non-defensive JavaScript libraries (Mega, SpiderOak). In the case of Java applets, this also has the advantage of significantly increasing the performance of the application (DJCL is up to 100 times faster on large inputs) and of reducing the attack surface by removing the Java runtime from the trusted computing base.

Second, we propose a new encrypted local storage mechanism for applications that need to store encryption keys in the browser. This mechanism relies on the availability of an embedded *session key* that is specific to the browser session and is embedded into code served by the script server, but not given to the host page.

As a practical example, we show how to use both these mechanisms to make the ConfiChair conference management system more resilient against XSS attacks. ConfiChair uses the following cryptographic API (types shown for illustration):

```
derive_secret_key
  //:(input:string,salt:string)->key:string
base64_encode, base64_decode //:string->string
encryptData, decryptData
  //:(data:string,key:string)->string
encryptKeypurse//:(key:string,keypurse:json)->string
decryptKeypurse//:(key:string,string)->keypurse:json
```

When the user logs in, a script on the login page calls `derive_secret_key` with the password to compute a secret *user key* which is stored in `localStorage`. When the user clicks on a particular document to download (a paper or a review), the conference page downloads the encrypted PDF along with an encrypted *keypurse* for the user. It decrypts the *keypurse* with the user key, stores it in `localStorage`, and uses it to decrypt the PDF. The main vulnerability here is that any same-origin script can steal the user key (and *keypurse*) from local storage.

We write a drop-in replacement for this API in DJS. Instead of returning the real user key and *keypurse* in `derive_secret_key` and `decryptKeypurse`, our API returns keys encrypted (wrapped) under a `sessionKey`. When `decryptData` is called, it transparently unwraps the provided key, never exposing the user key to the page. Both the encrypted user key and *keypurse* can be safely stored in `localStorage`, because it cannot be read by scripts that do not know `sessionKey`. We protect the integrity of these keys with authenticated encryption.

Our design relies on a *secure script server* that can deliver defensive scripts embedded with session keys. Concretely, this is a web service running in a trusted, isolated origin (a subdomain like `secure.confichair.org`) that accepts GET requests with a script name and a target origin as parameters. It authenticates the target origin by

verifying the `Origin` header on the request, and may reject requests for some scripts from some origins. It then generates a fresh `sessionKey`, embeds it within the defensive script and sends it back as a GET response. The `sessionKey` remains the same for all subsequent requests in the same browsing session (using cookies).

Evaluation. Our changes to the ConfiChair website amount to replacing its Java applet with our own cryptographic API and rewriting two lines of code from the login page. The rest of the website works without further modification while enjoying a significantly improved security against XSS attacks. Using ProVerif, we analyzed our API (with an idealized model of the script server and login page) and verified that it does not leak the user key, *keypurse*, or `sessionKey`. Our cryptographic API looks similar to the upcoming Web Cryptography API standard, except that it protects keys from same-origin attackers, whereas the proposed API does not.

7 Related Work

Attacks similar to the ones we describe in Section 2 have been reported before in the context of password manager bookmarklets [1], frame busting defenses [35], single sign-on protocols [6, 36, 41], payment processing components [42], smartphone password managers [9], and encrypted cloud storage [5, 10]. These works provide further evidence for the need for defensive programming techniques and automated analysis for web applications.

A number of works explore the use of frames and inter-frame communication to isolate untrusted components on a page or a browser extension by relying on the same origin policy [2, 7, 8, 27, 44]. Our approach is orthogonal; we seek to protect scripts against same-origin attackers using defensive programming in standard JavaScript. Moreover, DJS scripts require fewer privileges than frames (they cannot open windows, for example) and unlike components written in full HTML, DJS programs can be statically analyzed for security.

A variety of JavaScript subsets attempt to protect trusted web pages from untrusted [20, 26, 28, 29, 31, 32, 34, 39]. Our goal is instead to run trusted components within untrusted web pages, hence our security goals are stronger, and our language restrictions are different. For example, these subsets rely on first-starter privilege, that is, they only offer isolation on web pages where their setup code runs first so that it can restrict the code that follows. Our scripts do not need such privileges.

[21] proves full abstraction for a compiler from f* (a subset of ML) to JavaScript. Their theorem ensures that programmers can reason about deployed f* programs entirely in the semantics of the source language, ignoring JavaScript-specific details. As such, their translation is

also robust against corruption of the JavaScript environment. However, there are also some significant limitations. In particular, their theorems do not account for HTML-level attackers who can, say, open frames and call their functions. We also reported flaws in their translation (since fixed in their online version). In comparison, our programs are written directly in a subset of JavaScript and can defend themselves against stronger threats, including full HTML adversaries that may execute before, after, and concurrently with our programs.

Dynamic information flow analyses for various subsets of JavaScript [3, 17, 24] enforce a security property called noninterference. Our static type system enforces defensiveness and we analyze security by model extraction. Relating defensiveness to noninterference remains future work; we conjecture that DJS may be more suitable than JavaScript to static information flow analysis.

8 Conclusion

Given the complexity and heterogeneity of the web programming environment and the wide array of threats it must contend with, it is difficult to believe that any web application can enjoy formal security guarantees that do not break easily in the face of concerted attack. Instead of relying on the absence of web vulnerabilities, this paper presents a defense-in-depth strategy. We start from a small hardened core (DJS) that makes minimal assumptions about the browser and JavaScript runtime, and then build upon it to obtain defensive security for critical components. We show how this strategy can be applied to existing applications, with little change to their code but a significantly increase in their security. We believe our methods scale, and lifting these results to protect full websites that use HTML and PHP is ongoing work.

Acknowledgements The authors would like to thank David Wagner, Nikhil Swamy and the anonymous reviewers for their helpful comments leading to significant improvements to this paper. We would also like to acknowledge the Mozilla and Facebook security teams for prompt and constructive discussions about our attacks. Bhargavan and Delignat-Lavaud are supported by the ERC Starting Grant CRYSP. Maffeis is supported by EPSRC grant EP/I004246/1.

References

- [1] B. Adida, A. Barth, and C. Jackson. Rootkits for JavaScript environments. In *WOOT*, 2009.
- [2] D. Akhawe, P. Saxena, and D. Song. Privilege separation in HTML5 applications. In *USENIX Security*, 2012.
- [3] T. Austin and C. Flanagan. Multiple facets for dynamic information flow. In *POPL*, pages 165–178, 2012.
- [4] M. Avalle, A. Pironti, D. Pozza, and R. Sisto. JavaSPI: A framework for security protocol implementation. *International Journal of Secure Software Engineering*, 2:34–48, 2011.
- [5] C. Bansal, K. Bhargavan, A. Delignat-Lavaud, and S. Maffeis. Keys to the cloud: Formal analysis and concrete attacks on encrypted web storage. In *POST*, 2013.
- [6] C. Bansal, K. Bhargavan, and S. Maffeis. Discovering concrete attacks on website authorization by formal analysis. In *CSF*, pages 247–262, 2012.
- [7] A. Barth, C. Jackson, and W. Li. Attacks on JavaScript mashup communication. In *W2SP*, 2009.
- [8] A. Barth, C. Jackson, and J.C. Mitchell. Securing browser frame communication. In *USENIX Security*, 2008.
- [9] A. Belenko and D. Sklyarov. “Secure password managers” and “Military-grade encryption” on smartphones: Oh, really? Technical report, Elmcomsoft Ltd., 2012.
- [10] K. Bhargavan and A. Delignat-Lavaud. Web-based attacks on host-proof encrypted storage. In *WOOT*, 2012.
- [11] K. Bhargavan, A. Delignat-Lavaud, and S. Maffeis. Defensive JavaScript website with testbed, technical report and supporting materials. <http://www.defensivejs.com>, 2013.
- [12] K. Bhargavan, C. Fournet, A. D. Gordon, and S. Tse. Verified interoperable implementations of security protocols. In *CSFW*, pages 139–152, 2006.
- [13] B. Blanchet and B. Smyth. *ProVerif: Automatic Cryptographic Protocol Verifier, User Manual and Tutorial*. <http://www.proverif.inria.fr/manual.pdf>.
- [14] P. Canning, W. Cook, W. Hill, W. Olthoff, and J. Mitchell. F-bounded polymorphism for object-oriented programming. In *FPCA*, pages 273–280, 1989.
- [15] L. Cardelli. Extensible records in a pure calculus of subtyping. In *In Theoretical Aspects of Object-Oriented Programming*, pages 373–425. MIT Press, 1994.

- [16] D. Crockford. ADsafe: Making JavaScript safe for advertising. <http://www.adsafe.org/>, 2008.
- [17] W. De Groef, D. Devriese, N. Nikiforakis, and F. Piessens. FlowFox: a web browser with flexible and precise information flow control. In *CCS*, pages 748–759, 2012.
- [18] D. Dolev and A.C. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, IT-29(2):198–208, 1983.
- [19] M. Finifter, A. Mettler, N. Sastry, and D. Wagner. Verifiable functional purity in Java. In *CCS*, pages 161–174. ACM, 2008.
- [20] M. Finifter, J. Weinberger, and A. Barth. Preventing Capability Leaks in Secure JavaScript Subsets. In *BDSS*, 2010.
- [21] C. Fournet, N. Swamy, J. Chen, P. Dagand, P. Strub, and B. Livshits. Fully abstract compilation to JavaScript. In *POPL’13*, 2013.
- [22] P. Haack. JSON hijacking. <http://hhacked.com/2009/06/25/json-hijacking.aspx>, 2009.
- [23] D. Hardt. The OAuth 2.0 authorization framework. IETF RFC 6749, 2012.
- [24] D. Hedin and A. Sabelfeld. Information-flow security for a core of JavaScript. In *CSF*, pages 3–18, 2012.
- [25] IETF. *JavaScript Object Signing and Encryption (JOSE)*, 2012. <http://tools.ietf.org/wg/jose/>.
- [26] S. Maffei, J. C. Mitchell, and A. Taly. Isolating JavaScript with filters, rewriting, and wrappers. In *ESORICS’09*, 2009.
- [27] L. Meyerovich, A. Porter Felt, and M. Miller. Object views: Fine-grained sharing in browsers. In *WWW*, 2010.
- [28] L. Meyerovich and B. Livshits. ConScript: Specifying and enforcing fine-grained security policies for JavaScript in the browser. In *IEEE S&P*, 2010.
- [29] J. Mickens and M. Finifter. Jigsaw: Efficient, low-effort mashup isolation. In *USENIX Web Application Development*, 2012.
- [30] R. Milner. Functions as processes. In *Automata, Languages and Programming*, volume 443, pages 167–180. 1990.
- [31] P. Phung, D. Sands, and D. Chudnov. Lightweight self-protecting JavaScript. In *ASIACCS*, 2009.
- [32] J. Politz, S. Eliopoulos, A. Guha, and S. Krishnamurthi. ADsafety: Type-based verification of JavaScript sandboxing. In *USENIX Security*, 2011.
- [33] F. Pottier. Type inference in the presence of subtyping: from theory to practice. Research Report 3483, INRIA, September 1998.
- [34] C. Reis, J. Dunagan, H. Wang, O. Dubrovsky, and S. Esmeir. BrowserShield: Vulnerability-driven filtering of dynamic HTML. *ACM Transactions on the Web*, 1(3), 2007.
- [35] G. Rydstedt, E. Bursztein, D. Boneh, and C. Jackson. Busting frame busting: a study of clickjacking vulnerabilities at popular sites. In *W2SP’10*, 2010.
- [36] J. Somorovsky, A. Mayer, A. Worth, J. Schwenk, M. Kampmann, and M. Jensen. On breaking SAML: Be whoever you want to be. In *WOOT*, 2012.
- [37] E. Stark, M. Hamburg, and D. Boneh. Symmetric cryptography in JavaScript. In *ACSAC*, pages 373–381, 2009.
- [38] B. Sterne and A. Barth. Content Security Policy 1.0. W3C Candidate Recommendation, 2012.
- [39] A. Taly, Ú. Erlingsson, J. C. Mitchell, M. Miller, and J. Nagra. Automated analysis of security-critical JavaScript APIs. In *IEEE S&P*, 2011.
- [40] Google Caja Team. A source-to-source translator for securing JavaScript-based web. <http://code.google.com/p/google-caja/>.
- [41] R. Wang, S. Chen, and X. Wang. Signing me onto your accounts through facebook and google: A traffic-guided security study of commercially deployed single-sign-on web services. In *IEEE S&P*, pages 365–379. IEEE Computer Society, 2012.
- [42] R. Wang, S. Chen, X. Wang, and S. Qadeer. How to shop for free online - security analysis of cashier-as-a-service based web stores. In *IEEE S&P*, pages 465–480, 2011.
- [43] M. Zalewski. *The Tangled Web*. No Starch Press, November 2011.
- [44] L. Zhengqin and T. Rezk. Mashic compiler: Mashup sandboxing based on inter-frame communication. 2012.

Take This Personally: Pollution Attacks on Personalized Services

Xinyu Xing, Wei Meng, Dan Doozan, Alex C. Snoeren[†], Nick Feamster, and Wenke Lee

Georgia Institute of Technology and [†]UC San Diego

Abstract

Modern Web services routinely personalize content to appeal to the specific interests, viewpoints, and contexts of individual users. Ideally, personalization allows sites to highlight information uniquely relevant to each of their users, thereby increasing user satisfaction—and, eventually, the service’s bottom line. Unfortunately, as we demonstrate in this paper, the personalization mechanisms currently employed by popular services have not been hardened against attack. We show that third parties can manipulate them to increase the visibility of arbitrary content—whether it be a new YouTube video, an unpopular product on Amazon, or a low-ranking website in Google search returns. In particular, we demonstrate that attackers can inject information into users’ profiles on these services, thereby perturbing the results of the services’ personalization algorithms. While the details of our exploits are tailored to each service, the general approach is likely to apply quite broadly. By demonstrating the attack against three popular Web services, we highlight a new class of vulnerability that allows an attacker to affect a user’s experience with a service, unbeknownst to the user or the service provider.

1 Introduction

The economics of the Web ecosystem are all about clicks and eyeballs. The business model of many Web services depends on advertisement: they charge for prime screen real estate, and focus a great deal of effort on developing mechanisms that make sure that the information displayed most prominently is likely to create revenue for the service, either through a direct ad purchase, commission, or at the very least improving the user’s experience. Not surprisingly, malfeasants and upstanding business operators alike have long sought to reverse engineer and exploit these mechanisms to cheaply and effectively place their own content—whether it be items for

sale, malicious content, or affiliate marketing schemes. Search engine optimization (SEO), which seeks to impact the placement of individual Web pages in the results provided by search engines, is perhaps the most widely understood example of this practice.

Modern Web services are increasingly relying upon personalization to improve the quality of their customers’ experience. For example, popular websites tailor their front pages based on a user’s previous browsing history at the site; video-sharing websites such as YouTube recommend related videos based upon a user’s watch history; shopping portals like Amazon make suggestions based on a user’s previous purchases; and search engines such as Google return customized results based upon a wide variety of user-specific factors. As the Web becomes increasingly personal, the effectiveness of broad-brush techniques like SEO will wane. In its place will rise a new class of schemes and outright attacks that exploit the mechanisms and algorithms underlying this personalization. In other words, personalization represents a new attack surface for all those seeking to steer user eyeballs, regardless of their intents.

In this paper, we demonstrate that contemporary personalization mechanisms are vulnerable to exploit. In particular, we show that YouTube, Amazon, and Google are all vulnerable to the same class of cross-site scripting attack, which we call a *pollution attack*, that allows third parties to alter the customized content the services return to users who have visited a page containing the exploit. Although the attack is quite effective, we do not claim that it is the most powerful, broadly applicable, or hard to defeat. Rather, we present it as a first example of a class of attacks that we believe will soon—if they are not already—be launched against the relatively unprotected underbelly of personalization services.

Our attack exploits the fact that a service employing personalization incorporates a user’s past history (including, for example, browsing, searching and purchasing activities) to customize the content that it presents to the

user. Importantly, many services with personalized content log their users’ Web activities whenever they are logged in regardless of the site they are currently visiting; other services track user activities on the site even if the user is logged out (*e.g.*, through a session cookie). We use both mechanisms to pollute users’ service profiles, thereby impacting the customized content returned to the users in predictable ways. Given the increasing portfolio of services provided by major players like Google and Amazon, it seems reasonable to expect that a large fraction of users will either be directly using the service or at least logged in while browsing elsewhere on the Web.

We show that pollution attacks can be extremely effective on three popular platforms: YouTube, Google, and Amazon. A distinguishing feature of our attack is that it does not exploit any vulnerability in the user’s Web browser. Rather, it leverages these services’ own personalization mechanisms to alter user’s experiences. While our implementation employs cross-site request forgery (XSRF) [13], other mechanisms are possible as well.

The ability to trivially launch such an attack is especially worrisome because it indicates the current approach to Web security is ill-equipped to address the vulnerabilities likely to exist in personalization mechanisms. In particular, today’s Web browsers prevent exploits like cross-site scripting and request forging by enforcing boundaries between domains though “same origin” policies. The limitations of these approaches are well known, but our attack represents a class of exploits that cannot be stopped by client-side enforcement: in an attempt to increase the footprint of its personalization engine (*e.g.*, Google recording search queries that a user enters on a third-party page), a service with personalized services is providing the cross-site vector itself. Hence, only the service can defend itself from such attacks on its personalization. Moreover, enforcing isolation between independent Web sessions seems antithetical to the goal of personalization, which seeks to increase the amount of information upon which to base customization attempts.

This paper makes the following contributions:

- We describe pollution attacks against three platforms—YouTube, Google, and Amazon—that allow a third party to alter the personalized content these services present to users who previously visited a Web page containing the exploit.
- We study the effectiveness of our attack on each of these platforms and demonstrate that it (1) can increase the visibility of almost any YouTube channel; (2) dramatically increase the ranking of most websites in the short term, and even have lasting impacts on the personalized rankings of a smaller set of sites, and (3) cause Amazon to recommend reasonably popular products of the attacker’s choosing.

- Our attack and its effectiveness illustrates the importance of securing personalization mechanisms in general. We discuss a number of implications of our study and ways for websites to mitigate similar vulnerabilities in the future.

The rest of the paper is organized as follows. Section 2 provides a general overview of pollution attacks on personalized services. Sections 3, 4, and 5 introduce specific attacks that can be launched against YouTube, Google, and Amazon, respectively, and report on our success. We survey related work in Section 6 and discuss limitations of our work and possible defenses in Section 7 before concluding in Section 8.

2 Overview and Attack Model

In this section, we present a brief overview of personalization as it is used by popular Web services. We then present a model of pollution attacks, which we apply to three different scenarios later in the paper: YouTube, Amazon, and Google.

2.1 Personalization

Online services are increasingly using personalization to deliver information to users that is tailored to their interests and preferences. Personalization potentially creates a situation where both the service provider and the user benefit: the user sees content that more closely matches preferences, and the service provider presents products that the user is more likely to purchase (or links that the user is more likely to click on), thus potentially resulting in higher revenues for the service provider.

The main instrument that a service provider can use to affect the content that a user sees is modifying the *choice set*, the set of results that a user sees on a particular screen in response to a particular query. The size of a choice set differs for different services. For example, YouTube shows the user anywhere from 12–40 videos; Amazon may show the user up to five sets of recommended products; Google’s initial search results page shows the top ten results. Figure 1 shows several examples of choice sets on different sites.

When a user issues a query, a service’s *personalization algorithm* affects the user’s choice set for that query. The choice set that a personalization algorithm produces depends on a user query, as well as a number of auxiliary factors, including the universe of all possible content and the user’s browsing history. Previous work has claimed that many factors, ranging from geography to time of day, may affect the choice set that a user sees. For the purposes of the attacks in this paper, we focus on how changes to a user’s history can affect the choice set,



Figure 1: websites with personalized services (personalized services tailor the data in the red rectangles).

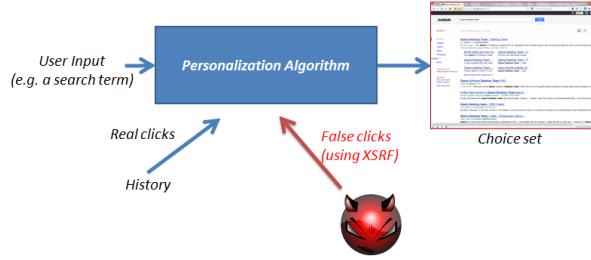


Figure 2: Overview of how history pollution can ultimately affect the user’s choice set.

holding other factors fixed. In particular, we study how an attacker can pollute the user’s history by generating false clicks through cross-site request forgery (XSRF). We describe these attacks in the next section.

2.2 Pollution Attacks

The objective of a pollution attack is to affect a user’s *choice set*, given a particular input. In some cases, a user’s choice set appears before the user enters any input (*e.g.*, upon an initial visit to the page). In this case, the attacker’s goal may be to affect a default choice set. Figure 2 shows an overview of the attacker’s goal: the attacker aims to affect the resulting choice set by altering the user’s history with false clicks, using cross-site request forgery as the attack vector. This attack requires three steps:

1. *Model the service’s personalization algorithm.* We assume that the attacker has some ability to model the personalization algorithm that the site uses to affect the user’s choice set. In particular, the attacker must have some idea of how the user’s past history affects the user’s choice set. This information is often available in published white papers, but in some cases it may require experimentation.
2. *Create a “seed” to pollute the user’s history.* Given some knowledge of the personalization algorithm and a goal for how to affect the choice set, the attacker must design the seed that is used to affect

the user’s choice set. Depending on the service, the seed may be queries, clicks, purchases, or any other activity that might go into the user’s history. A good seed can affect the user’s choice set with a minimal number of “false clicks”, as we describe next.

3. *Inject the seed with a vector of false clicks.* To pollute a user’s history, in most cases we require that the user be signed in to the site. (For some services, pollution can take place even when the user is not signed in.) Then, the attacker can use a mechanism to make it appear as though the user is taking action on the Web site for a particular service (*e.g.*, clicking on links) using a particular attack vector.

In the following sections, we explore how an attacker can apply this same procedure to attack the personalization algorithms of three different services: YouTube, Amazon, and Google search.

3 Pollution Attacks on YouTube

In this section, we demonstrate our attack on YouTube¹. Following the attack steps we described in Section 2, we first model how YouTube uses the watch history of a YouTube user account to recommend videos by reviewing the literature [5]. Second, we discuss how to prepare seed data (*i.e.*, seed videos) to promote target data (*i.e.*, target videos belonging to a specific channel). Third, we introduce how to inject the seed videos to a YouTube user account. Finally, we design experiments and quantify the effectiveness of our attack.

3.1 YouTube Personalization

YouTube constructs a personalized list of recommended videos based upon the videos a user has previously viewed [5]. YouTube attempts to identify the subset of previously viewed videos that the user enjoyed by considering only those videos that the user watched for a long period of time. Typically, YouTube recommends videos that other users with similar viewing histories

¹A demo video is available at <http://www.youtube.com/watch?v=8hi52ws98A>.

have also enjoyed. YouTube tracks the co-visitation relationship between pairs of videos, which reflects how likely a user who watched a substantial portion of video X will also watch and enjoy video Y . In general, there may be more videos with co-visitation relationships than there is display area, so YouTube prioritizes videos with high rankings. YouTube will not recommend a video the user has already watched.

YouTube displays recommended videos in the suggestion list placed alongside with a playing video (*e.g.*, Figure 5) and in the main portion of the screen at the end of a video (Figure 1(a)). A suggestion list appearing next to a video typically contains 20–40 suggested videos, two of which are recommended based upon personalization. At the end of a video, YouTube shows an more concise version of the suggestion list that contains only twelve of the videos from the full list; these videos may or may not contain personal recommendations.

3.2 Preparing Seed Videos

YouTube organizes videos into channels, where each channel corresponds to the set of uploads from a particular user. In our attack, we seek to promote a set of target videos, Ω^T , all belonging to the same YouTube channel, C . To do so, we will use an additional set of seed videos, Ω^S , that have a co-visitation relationship with the target videos. By polluting a user’s watch history with videos in Ω^S , we can cause YouTube to recommend videos in Ω^T . There are two ways to obtain Ω^S : we can identify videos with pre-existing co-visitation relationships to the target videos, or we can create the relationships ourselves.

Existing Relationships. In the simplest version of the attack, the attacker identifies existing videos to use as the seed set. For example, given a target video set Ω^T belonging to channel C , the attacker could consider all of the other videos in the channel, $C - \Omega^T$, as candidate seeds. For every candidate video, the attacker checks which videos YouTube recommends when a fresh YouTube account (*i.e.*, a YouTube account with no history) watches it. YouTube allows its users to view their recommended videos at <http://www.youtube.com/feed/recommended>. If the candidate video triggers YouTube to recommend a video in Ω^T , then the attacker adds the injected video to seed video set Ω^S .

In general, this process allows the attacker to identify seed videos for every target video in Ω^T . The attacker cannot yet launch the attack, though, because a YouTube video in Ω^S may trigger YouTube to also recommend videos not in Ω^T . To address this issue, the attacker can simply add these unwanted videos to the seed video set Ω^S because YouTube does not recommend videos that the user has already watched. As we will show later, the

attacker can convince YouTube that the user watched, but *did not* enjoy, these unwanted videos, so their inclusion in Ω^S will not lead to additional recommendations.

Fabricating Relationships. For some videos, it may be difficult to identify a seed set Ω^S that recommends all of the elements of Ω^T due to lack of co-visitation relationships for some of the target elements. Instead, attackers who upload their own content to use as the seed set can create co-visitation relationships between this content and the target set. In particular, an attacker uploads a set of videos, Ω^0 , and establishes co-visitation relationships between Ω^0 and Ω^T through crowd-sourcing (*e.g.*, Mechanical Turk or a botnet): YouTube visitors need only watch a video in Ω^0 followed by a video in Ω^T . After a sufficient number of viewing pairs, the attacker can use videos in Ω^0 as the seed set. As we will show in Section 3.4.1, a relatively small number of viewing pairs suffices.

3.3 Injecting Seed Videos

To launch the attack and inject seed videos into a victim’s YouTube watch history, an attacker can harness XSRF to forge the following two HTTP requests for each video in the seed set: (1) http://www.youtube.com/user_watch?plid=<value>&video_id=<value>, and (2) http://www.youtube.com/set_awesome?plid=<value>&video_id=<value>, where `plid` and `video_id` correspond to the values found in the source code of the seed video’s YouTube page. The first HTTP request spoofs a request from the victim to start watching the seed video, and the second convinces YouTube that the victim watched the video for a long period of time. Both HTTP requests are required for videos in Ω^S to trigger the recommendation of videos in Ω^T , but only the first HTTP request is needed to prevent the recommendation of unwanted videos.

3.4 Experimental Design

We evaluated the effectiveness of our attack both in controlled environments and against real YouTube users. We first validated the the attack in the simplest scenario, where the attack promoted existing YouTube channels through existing co-visitation relationships. We then considered the scenario where an attack seemed to upload and promote content from a channel that the attacker created. Finally, we conducted a small-scale experiment to demonstrate the effectiveness of the attack against a volunteer set of real YouTube users.

3.4.1 New Accounts

We first promoted existing YouTube channels by launching our attack against victims with fresh YouTube user accounts. This experiment confirms the effectiveness of our approach in the absence of other, potentially countervailing influences, such as recommendations based on a user’s existing history.

We began by selecting 100 existing YouTube channels at random from the list of the top 2,000 most-subscribed channels published by VidStatsX [19]. For each of the selected YouTube channels, we randomly selected 25 videos from the channel as the target video set, used the method described in the previous section to identify a seed video set, and injected the seed videos to a fresh YouTube account.

We then considered promoting new content by creating our own YouTube channel and similarly attacking fresh YouTube accounts. Our YouTube channel contains two 3-minute videos. We selected one of the videos as a one-element target video set and used the other as the seed set. We created a co-visitation relationship by embedding both videos on a web page and recruiting volunteers to watch both videos sequentially. We obtained 65 and 68 views for our seed and target video respectively.

3.4.2 Existing Accounts

We studied the effectiveness of our pollution attack using real YouTube user accounts. We recruited 22 volunteers with extensive pre-existing YouTube watch histories. To limit the inconvenience to our volunteers, we limited our study to attempting to promote one moderately popular YouTube channel based upon existing co-visitation relationships. We selected a moderately popular account because a popular channel may be recommended anyway (regardless of our attack); conversely, an entirely new channel requires a certain amount of effort to establish the co-visitation relationships as described above and we have limited volunteer resources.

Based on these parameters, we arbitrarily selected the channel *OnlyyouHappycamp*. We believe this selection is a reasonable candidate to be promoted using our attack for several reasons. First, compared to popular channels, most videos in *OnlyyouHappycamp* have low view counts (about 2,000 view counts per video on average) and the number of subscribers to the channel is a similarly modest 3,552. Both of these are easily achievable by an attacker at fairly low cost². Second, most videos in *OnlyyouHappycamp* are 22 minutes long, which makes them suitable for promotion. As we will explain in Section 3.5.1, the length of a target video affects its likeli-

²According to the prices in underground markets such as freelancer.com and fiverr.com, 40,000 view counts and 10,000 subscribers cost \$15 and \$30 US dollars, respectively.

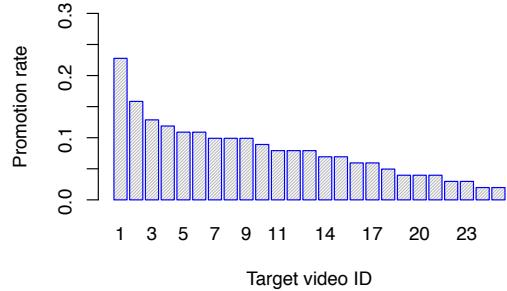


Figure 3: The promotion rate for each of the 25 target videos in channel *lady16makeup*. Two videos were recommended in each of the 114 trials.

hood for being recommended as a result of a co-visitation relationship with another video.

Similar to the experiments with new accounts, we randomly selected 15 target videos from channel *OnlyyouHappycamp*, identified a seed set, and injected the seed videos into the volunteers’ YouTube accounts. After pollution, the volunteers were asked to use their accounts to watch three videos of their choice and report the suggestion list displaying alongside each of their three videos.

3.5 Evaluation

We evaluated the effectiveness of our pollution attacks by logging in as the victim user and viewing 114 representative videos³. We measured the effectiveness of our attack in terms of *promotion rate*: the fraction of the 114 viewings when at least one of the target videos was contained within the video suggestion list. Recall that the list contains at most two personalized recommendations (see Section 3.1); we deem the attack successful if one or both of these videos are videos that were promoted as a result of a pollution attack.

3.5.1 New Accounts

Pollution attacks successfully promoted target videos from each of the 100 selected existing channels: Each time we injected seed videos for a particular channel, we observed the target videos in the suggestion list for each of the 114 videos. Since these are fresh accounts, there is no other history, so our targeted videos always occupy both of the personalized recommendation slots.

In addition, we observed the particular target videos shown in the suggestion video list varied, even when

³We attempted to view 150 videos random from a trace of YouTube usage at our institution over the course of several months. Unfortunately, 36 of the videos were no longer available at the time of our experiment.

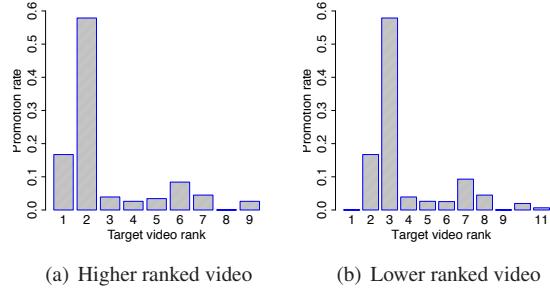


Figure 4: Distribution of the suggestion slots occupied by each of the two successfully promoted target videos.

we were viewing the same video using the same victim YouTube account. In other words, every target video has a chance to be promoted and shown on the suggestion video list no matter which video a victim plays. Figure 3 shows the frequency with which each of the 25 target videos for a representative channel, *lady16makeup*. In an attempt to explain this variation, we computed (1) the Pearson correlation between the showing frequencies and the lengths of the target videos for each channel (ρ_l); (2) the Pearson correlation between the showing frequencies and the view counts of these target videos for each channel (ρ_{cnt}). We found the average Pearson correlation values are medium ($\rho_l = 0.54$) and moderate ($\rho_{cnt} = 0.23$), respectively. This suggests that both the length and view count of a target video influence its recommendation frequency, but the length of a target video is a more significant factor.

Since screen real estate is precious, and users typically focus on the first few items of a list, we report on the position within the suggested video lists that our targeted videos occupied when they were promoted. We observed that the two target videos were usually placed back-to-back on the suggestion list. Figure 4 shows that YouTube usually placed our target videos among the top few spots of a victim’s suggestion list: in our tests with new accounts, the target videos were always recommended and placed on the top 12, which meant they also appeared at the end of viewed videos. This finding is particularly significant because it implies that our target videos are shown even if a victim finishes watching a YouTube video on a third-party website (which typically embeds only the view-screen portion of the YouTube page, and not the full suggestion list).

Our attacks were similarly completely successful in promoting newly uploaded content. As a control, we also signed in as non-polluted fresh YouTube accounts and, unsurprisingly, did not find any of our new content among the videos in the suggestion list. In other words, the videos were recommended exclusively because of our attacks; our experiments were sufficiently



Figure 5: Suggestion lists before (left) and after (right) a pollution attack against a fresh YouTube user account. The video highlighted in red is our uploaded video.

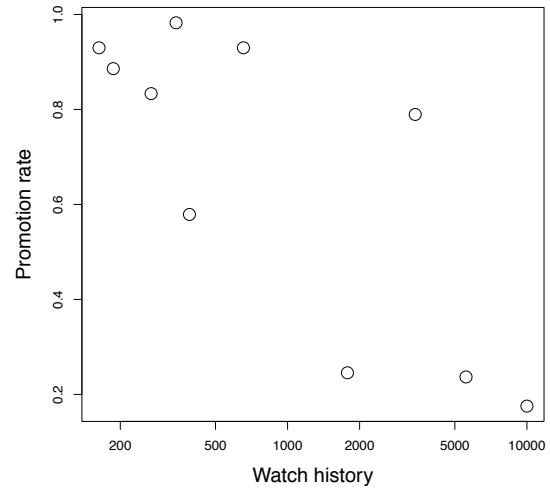


Figure 6: Promotion success rates for 10 real YouTube user accounts with varying watch history lengths.

small that we did not lead YouTube to conclude that our content was, in fact, universally popular. Figure 5 shows a sample screenshot comparing the suggestion lists from a victim account and another, non-exploited fresh account. Finally, we found that one of our target videos occupied the top suggestion slot while viewing 80 out of the 114 test videos.

3.5.2 Existing Accounts

Our attacks were somewhat less successful on real YouTube accounts. We found that 14 out of the 22 volunteer YouTube users reported that at least one of our target videos from channel *OnlyyouHappycamp* appeared in the suggestion list during each of their three video viewings, a 64% promotion rate.

To understand why we were able to exploit some accounts and not others, we asked our volunteers to share their YouTube watch histories. Ten of our volunteers shared their histories with us and allowed us to sign in to

their YouTube accounts to conduct a further study. The number of videos in the watch histories of the ten volunteers ranged from a few hundred to tens of thousands. Figure 6 shows the relationship between the number of watched videos in a watch history and the number of times that at least one of our target videos is displayed along with a playing video. While there appears to be an intuitive decreasing trend (i.e., the longer the history an account has the more resistant it is to pollution), there are obvious outliers. For example, one account with almost 3,500 previous viewings in its history succumbed to our attacks almost 80% of the time.

Consistent with the Pearson coefficients reported earlier, we found that the success of our attacks depends on the rankings and lengths of the videos that are otherwise suggested based upon a user’s history. In particular, we observed that the majority of the videos recommended to users for whom our attacks have low promotion rates have longer lengths and more view counts than our target videos, while the videos that YouTube recommends based on the watch history of the user with 3,500 previous viewings have shorter lengths than our target videos (though they generally have higher view counts than our targets).

Although we believe our attack demonstrates that YouTube’s personalization mechanism is subject to exploit, the persistence of the attack effects is unclear. In our experiments, volunteers watched arbitrary YouTube videos right after being attacked, but we believe our pollution attacks on YouTube are likely to last for some time. Although YouTube does not explicitly disclose how time factors into their recommendation system (if at all) [5], analysis of volunteers’ watch histories indicates that a YouTube video that was watched as long as two weeks prior is still used for generating recommended videos.

4 Google Personalized Search

In this section, we show how history pollution attacks can be launched against Google’s search engine⁴. The goal of our attack is to promote a target webpage’s rank in the personalized results that Google returns for an arbitrary search term by injecting seed search terms into a victim’s search history.

4.1 Search Personalization

Search personalization customizes search results using information about users, including their previous query terms, click-through data and previously visited websites. The details of Google’s personalization algorithms

are not public, but many previous studies have explored aspects of personalized search [2, 4, 6, 7, 9, 10, 14–18]. We describe two classes of personalization algorithms: *contextual personalization* and *persistent personalization*. According to recent reports [11, 12], many search engines including Google, Bing, and Yahoo! apply both types of personalization.

Contextual personalization constructs a short-term user profile based on recent searches and clicks-through [4, 16]. When a user searches for “inexpensive furniture” followed by “maternity clothes,” Google’s contextual personalization algorithm typically promotes search results that relate to “inexpensive maternity clothes” for the next few searches (we provide an analysis of precisely how long this effect lasts in Appendix A.2). In contrast, persistent personalization uses the entire search history—as opposed to only recent searches—to develop a user profile [9, 15]. Personalization that occurs over the longer term may not affect a user’s search results as dramatically, but can have longer-lasting effects for the results that a user sees. For example, searching for “Egypt” using different accounts may result in two distinct result sets: one about tourism in Egypt and one related to the Arab Spring.

4.2 Identifying Search Terms

Given the differing underlying algorithms that govern contextual and persistent personalization, an attacker needs to select different sets of seed search terms depending on the type of attack she hopes to launch.

Contextual Personalization. For the contextual personalization attack, the keywords injected into a user’s search history should be both relevant to the promoting keyword and unique to the website being promoted. In particular, the keywords should be independent from other websites that have similar ranking in the search results, to ensure that only the target website is promoted. Presumably, an attacker promoting a specific website is familiar with the website and knows what keywords best meet these criteria, but good candidate keywords are also available in a website’s `meta` keyword tag. While Google no longer incorporates `meta` tags into their ranking function [3], the keywords listed in the `meta` keyword tag still provide a good summary of the page’s content.

Persistent Personalization. Launching a persistent personalization attack requires a different method of obtaining keywords to inject. In this case, the size of the keyword set should be larger than that used for a contextual attack in order to have a greater effect on the user’s search history. Recall that contextual attacks only affect a user’s current session, while persistent attacks pollute

⁴A demo video is available at <http://www.youtube.com/watch?v=73E5CLFYeu8>.

a user’s search history in order to have a lasting effect on the user’s search results. An attacker can determine suitable keywords using the Google AdWords tool, which takes as an input a search term and URL and produces a list of about one hundred related keywords. Ideally, an attacker could pollute a user’s search history with each of these terms, but a more efficient attack should be effective with a much smaller set of keywords. We determined that an attacker can safely inject roughly 50 keywords a minute using cross-site request forgery; more rapid search queries are flagged by Google as a screen-scraping attack. For this study, we assume an attacker can inject at most 25 keywords into a user’s profile, but the number of keywords can increase if the user stays on a webpage for more than 30 seconds. Not all keyword lists that AdWords returns actually promote the target website. The effectiveness of this attack likely depends on several factors, including the user’s current search history. In Section 4.5, we evaluate the effectiveness of this attack under different conditions.

4.3 Injecting Search Terms

As with the pollution attacks on YouTube, the attack on Google’s personalized search also uses XSRF to inject the seeds. For example, an attacker can forge a Google search by embedding <https://www.google.com/search?hl=en&site=&q=usenix+security+2013> into an invisible iframe. A Web browser will issue an embedded HTTP request, even if Google search response has an enabled `X-Frame-Option` header. Injecting search terms into a Google user’s account affects the search results of the user’s subsequent searches. The number and set of search terms to inject differs depending on whether an attacker can execute a contextual or persistent personalization attack.

4.4 Experimental Design

To cleanly study the effects of our proposed attacks on contextual and persistent search personalization, we conducted most of our experiments using Google accounts with no search history. To validate whether our results apply to real users, we also conducted a limited number of tests using accounts that we constructed to mimic the personae of real users.

To quantify the effectiveness of our attack in general, we must select an unbiased set of target web pages whose rankings we wish to improve. We built two test corpora, one for attacks on contextual personalization, and one for attacks on persistent personalization. We attempted to promote existing web sites using only their current content and link structure; we did not perform any SEO on websites before conducting the attacks. We believe this

represents a conservative lower bound on the effectiveness of the attack, as any individual website owner could engineer the content of their site to tailor it for promotion through search history pollution.

4.4.1 Contextual Pollution

We started by scraping 5,671 shopping-related keywords from `made-in-china.com` to use as search terms. We then entered each of these terms into Google one-by-one to obtain the top 30 (un-personalized) search results for each. Since some of our search terms are related, not all of these URLs are unique. Additionally, we cannot hope to improve the URLs that are already top-ranked for each of the search terms. We obtained 151,363 URLs whose ranking we could hope to improve.

Because we cannot manually inspect each of these websites to determine appropriate seed search terms, we instead focused a subset that include the `meta` keyword tag. For the approximately 90,000 such sites, we extracted the `meta` keywords or phrases from the website. Many of these keywords are generic and will appear in a wide variety of websites. To launch the attack, we require keywords that are unique to the website we wish to promote (at least relative to the other URLs returned in response to the same query), so we ignored any keywords that were associated with multiple URLs in the same set of search results.

This procedure ultimately yielded 2,136 target URLs spanning 1,739 different search terms, for which we had a set of 1–3 seed keywords to try to launch a contextual pollution attack. The average search term has 1.23 results whose ranking we tried to improve. Figure 11 in the Appendix shows the distribution of the original rankings for each of these target websites; the distribution is skewed toward highly ranked sites, perhaps because these sites take care in selecting their `meta` tag keywords.

4.4.2 Persistent Pollution

Once again, we begin by selecting 551 shopping-related search terms and perform Google searches with each of the search terms to retrieve the top 30 search results. As opposed to the contextual attack, where we search for keywords that differentiate the results from one another, we aim to determine search terms that will be associated with the website and search-term pair for the long term.

As described in Section 4.2, we use a tool provided by Google AdWords to obtain a set of keywords that Google associates with the given URL and search term. Constructing related keyword lists for each of the 29 search returns (again excluding the top hit, which we cannot hope to improve) and 551 search terms yields 15,979 distinct URLs with associated lists of keywords.

For each URL, we select 25 random keywords from the AdWords list for 25 distinct trials. If a trial improved a URL’s ranking, we then test the persistence of the attack by performing 20 subsequent queries, each with a randomly chosen set of Google trending keywords. These subsequent queries help us verify that the URL promotion is not just contextual, but does not vanish when a user searches other content. If after all 25 trials we find no keyword sets that promote the URL’s ranking and keep it there for 20 subsequent searches, we deem this URL attempt a failure. If multiple keyword sets succeed, we select the most effective (*i.e.*, the set of 25 keywords that induces the largest ranking improvement) trial to include in the test set.

4.5 Evaluation

In this section, we quantify the effectiveness of search history pollution with attacks that aimed to promote the target websites identified in the previous section. To scope our measurements, we consider the effectiveness of the attacks only for the set of search terms that we identify; it is quite possible, of course, that our pollution attacks also affect the rankings of the targeted URLs for other search terms.

When measuring the effectiveness of our attack, we use two different criteria, depending upon a website’s original position in the search results. In the case of URLs that are already in the first ten search results but not ranked first, we consider the pollution attack successful if it increases the ranking of a URL at all. For URLs subsequent pages, we consider the attack successful only if the attack moves the URL to the first page of search results, since improved ranking on any page that is not the first page is unlikely to have any utility.

4.5.1 Top-Ranked Sites

For the 2,136-page contextual attack test corpus, of the 846 pages that appeared on the front page prior to our attack, we improved the ranking of 371 (44%). The persistent attack was markedly less effective, with only 851 (17%) of the 4,959 test cases that originally appeared on the first page of the search results had ranking improvements surviving the persistence test (*i.e.*, they remained promoted after 20 random subsequent queries). In both cases, however, the probability of success depends greatly on the original ranking of the targeted URL. For example, promoting a second-ranked URL to the top-ranked position for contextual personalization succeeded 1.1% of the time, whereas promoting a tenth-ranked URL by at least one position succeeded 62.8% of the time. Similarly, for attacks on persistent personalization, moving a second-ranked URL to the top suc-

ceeded 4.3% of the time, and moving a tenth-ranked URL to a higher-ranked position succeeded 22.7% of the time. These results make sense, because second-ranked sites can only move into the top-ranked position, whereas sites that are ranked tenth can move into any one of nine higher spots.

To illustrate this effect and illuminate how far each webpage was promoted, Figure 7 shows the PDF of an improved webpage’s rank after contextual history pollution, based upon its position in the non-personalized search results. We observed that contextual pollution was able to promote most webpages by one or two spots, but some low-ranking webpages were also promoted to very high ranks. Similarly, Figure 8 shows the distributions for each result ranking for those websites whose rankings were improved by a persistent history pollution attack. Here, the distributions appear roughly similar (although the absolute probability of success is much lower), but it is difficult to draw any strong conclusions due to the small number of promoted sites of each rank for either class of attack.

4.5.2 The Next Tier

The remaining 1,290 test websites for the contextual attack were initially on the second or third page of search results. By polluting a user’s search history with the unique meta tag keywords associated with each site, we promoted 358 of them (28%) to the front page. Figure 7(j) shows that these websites were more likely to appear at the top of the results than those pages that were initially at the bottom of the first page. We suspect this phenomenon results from the choice of keywords used in pollution: because their original rankings were low, the pollution attack requires a distinguishing keyword to move one of the webpages to the front page at all. If such a keyword can move a search result to the first page, it might also be a good enough keyword to promote the page to a high rank on the first page, as well.

The results from the persistent test set are markedly different. Figure 8(j) shows that sites starting on the second or third page are unlikely to end up at the very top of the result list due to a persistent history attack: Only 80 (less than 1%) of the 11,020 attacks that attempted to promote a website appearing on the 2nd or 3rd page of results was successful in moving it to the front page (and keeping it there). This results shows that persistent search history attacks are generally best launched for sites that are already highly ranked, as opposed to contextual attacks, which can help even lower-ranked sites.

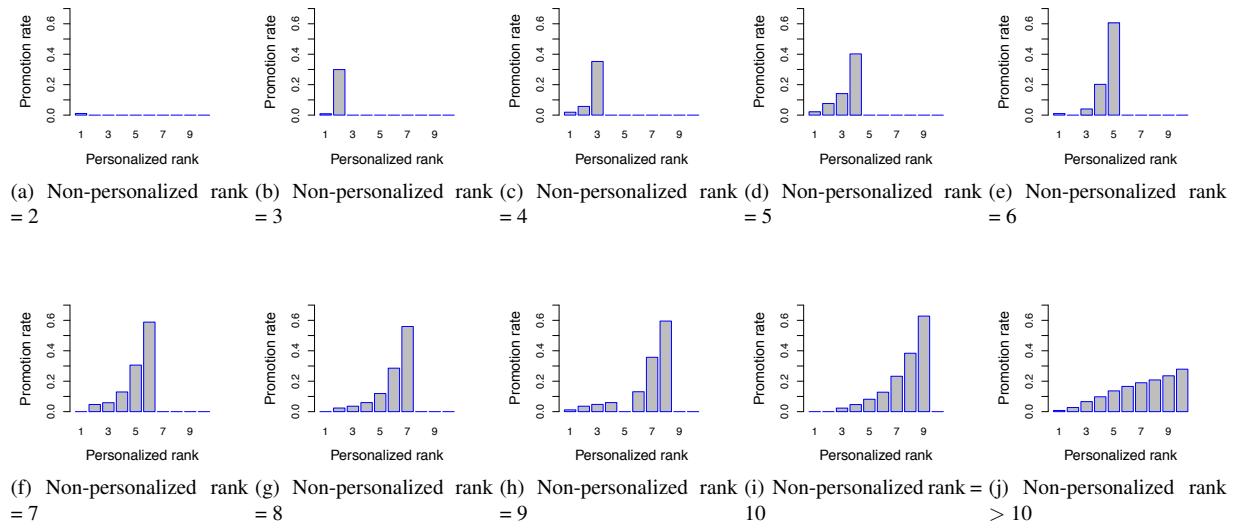


Figure 7: Promotion rates of promoted Google search rankings for successful contextual history pollution attacks.

4.5.3 Real Users

We also evaluate the effectiveness of pollution attacks on ten volunteers’ accounts with extensive pre-existing search histories. We find that, on average, 97.1% of our 729 previously successful contextual attacks remain successful, while only 77.78% of the persistent pollution attacks that work on fresh accounts achieve similar success. We believe that users’ search histories sometimes interfere with the attacks, and that user history interferes more with the attacks on persistent personalization. Contextualized attacks rely only on a small set of recent search terms to alter the personalized search results, which is unlikely to be affected by a user’s search history. In contrast, pollution attacks against persistent personalization rely on more of a user’s search history. If relevant keywords are already present in a user’s search history, keyword pollution may be less effective. In any event, both attacks are relatively robust, even when launched against users with long search histories.

5 Pollution Attacks on Amazon

Of the three services, Amazon’s personalization is perhaps the most evident to the end user. On one hand, this makes pollution-based attacks less insidious, as they will be visible to the observant user. On the other, of the three services, Amazon has the most direct monetization path, since users may directly purchase the goods from Amazon. Therefore, exploitation of Amazon’s personalization may be profitable to an enterprising attacker.

Amazon tailors a customer’s homepage based on the

previous purchase, browsing and searching behavior of the user. Amazon product recommendations consider each of these three activities individually and explicitly labels its recommendations according to the aspect of the user’s history it used to generate them. We focused on the personalized recommendations Amazon generates based on the browsing and searching activities of a customer because manipulating the previous purchase history of a customer may have unintended consequences.

5.1 Amazon Recommendations

Amazon displays five recommendation lists on a customer’s homepage that are ostensibly computed based on the customer’s searching and browsing history. Four of these lists are derived from the products that the customer has recently viewed (view-based recommendation); the fifth is based on the latest search term the customer entered (search-based recommendation). For each of the view-based recommendation lists, Amazon uses relationships between products that are purchased together to compute the corresponding recommended products; this concept is similar to the co-visitation relationship that YouTube uses to promote videos. For the recommendation list that is computed based on the latest search term of a customer, the recommended products are the top-ranked results for the latest search term.

In contrast to the types of personalization used for YouTube and Google Search, Amazon’s personalization is based on history that maintained by the user’s web browser, not by the service. Because customers frequently brows Amazon without being signed in, both the

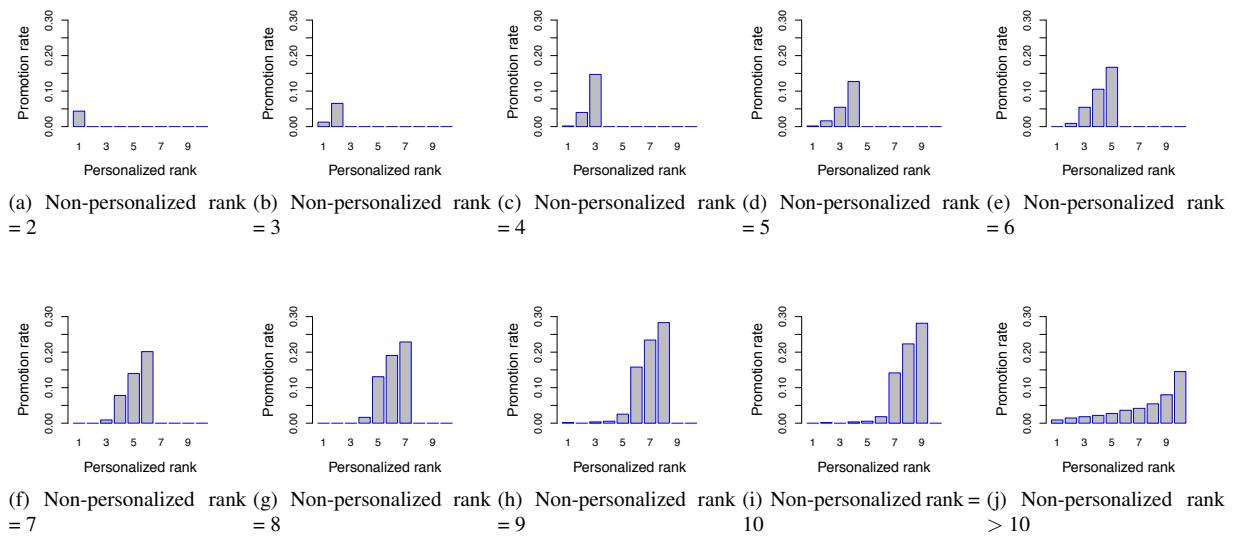


Figure 8: Promotion rates of promoted Google search rankings for successful persistent history pollution attacks.

latest viewed products and search term of the customer are stored in session cookies on the user’s browser rather than in profiles on Amazon servers.

5.2 Identifying Seed Products and Terms

Because Amazon computes the view and search-based recommendation lists separately, the seed data required to exploit each list must also be different.

Visit-Based Pollution. To promote a targeted product in a view-based recommendation list, an attacker must identify a seed product as follows. Given a targeted product that an attacker wishes to promote, the attacker visits the Amazon page of the product and retrieves the related products that are shown on Amazon page of the targeted product. To test the suitability of these related products, the attacker can visit the Amazon page of that product and subsequently check the Amazon home page. If the targeted product appears in a recommendation list, the URL of the candidate related product can serve as a seed to promote the targeted product.

Search-Based Pollution. To promote a targeted product in a search-based recommendation list, it suffices to identify an appropriate search term. If automation is desired, an attacker could use a natural language toolkit to automatically extract a candidate keyword set from the targeted product’s name. Any combination of these keywords that successfully isolates the targeted product can be used as the seed search term for promoting the targeted product. For example, to promote product “Breville BJE200XL Compact Juice Fountain 700-Watt Juice

Extractor”, an attacker can use XSRF to inject the search term “Breville BJE200XL” to replace an Amazon customer’s latest search term.

5.3 Injecting Views and Searches

As with the attacks on the previous two services, the attacker embeds the Amazon URLs of the desired seed items or search queries into a website that the victim’s browser is induced to visit with XSRF. For example, if one seed search term is “Coffee Maker”, the seed URL would be something like <http://www.amazon.com/s/?field-keywords=Coffee+Maker>. Similarly, an attacker could embed the URL of a seed product into an invisible *img* tag as the *src* of the image. When a victim visits the attacker’s website, Amazon receives the request for that particular query or item and customizes the victim’s Amazon website based on that search.

5.4 Experiment Design

To evaluate the effectiveness of the pollution attack against, we conducted two experiments. The first experiment measured the effectiveness of our attack when targeted toward popular items across different categories of Amazon products. The second quantified the effectiveness of our attack on randomly selected, mostly unpopular Amazon products.

5.4.1 Popular Products

Amazon categorizes sellers’ products into 32 root categories. To select products from each category, we

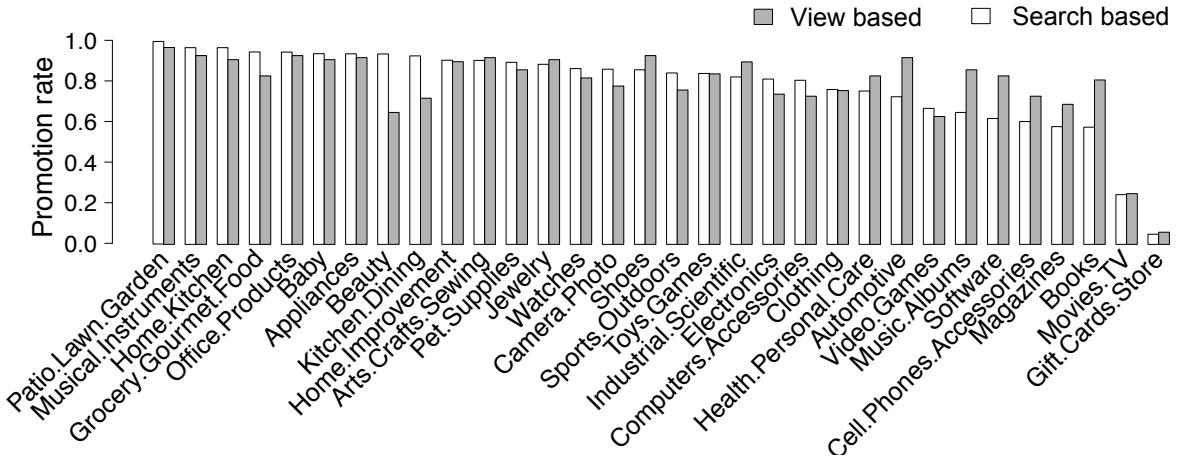


Figure 9: Promotion rates across Amazon categories.

scraped the top 100 best-selling products in each category in January 2013 and launched a separate attack targeting each of these 3,200 items.

5.4.2 Random Products

To evaluate the effectiveness of the pollution attack for promoting arbitrary products, we also selected products randomly. We downloaded a list of Amazon Standard Identification Number (ASIN) [1] that includes 75,115,473 ASIN records. Because each ASIN represents a Amazon product, we randomly sampled ASINs from the list and constructed a set of 3,000 products currently available for sale. For every randomly selected product in the list, we recorded the sale ranking of that product in its corresponding category.

5.5 Evaluation

Because Amazon computes search and visit-based recommendations based entirely upon the most recent history, we can evaluate the effectiveness of the pollution attack without using Amazon accounts from real users. Thus, we measured the effectiveness of our attack by studying the success rate of promoting our targeted products for fresh Amazon accounts.

5.5.1 Promoting Products in Different Categories

To evaluate the effectiveness of the pollution attack for each targeted product, we checked whether the ASIN of the targeted product matches the ASIN of an item in the recommendation lists on the user's customized Amazon homepage.

Figure 9 illustrates the promotion rate of target products in each category. The view-based and search-based

attacks produced similar promotion rates across all categories, about 78% on average. Two categories had significantly lower proportion rates: Gift-Cards-Store and Movies-TV (achieving 5% and 25%, respectively).

To understand why these categories yielded lower promotion rates, we analyzed the top 100 best selling products for each category. For Gift-Cards-Store, we found that there were two factors that distinguish gift cards from other product types. First, the gift cards all had similar names; therefore, using the keywords derived from the product name resulted in only a small number of specific gift cards being recommended. Second, we found that searching any combination of keywords extracted from the product names always caused a promotion of Amazon's own gift cards, which may imply that it is more difficult to promote product types that Amazon competes with directly.

Further investigation into the Movies-TV category revealed that Amazon recommends TV episodes differently. In our attempts to promote specific TV episodes, we found that Amazon recommends instead the first or latest episode of the corresponding TV series or the entire series. Because we declared a promotion successful only if the exact ASIN appears in the recommendation lists, these alternate recommendations are considered failures. These cases can also be considered successful because the attack caused the promotion of very similar products. Therefore, we believe that for all categories except for Gift-Cards-Store, an attacker has a significant chance of successfully promoting best-selling products.

5.5.2 Promoting Randomly Selected Products

We launched pollution attacks on 3,000 randomly selected products. We calculated the *Cumulative Success Rate* of products with respect to their rankings. The Cu-

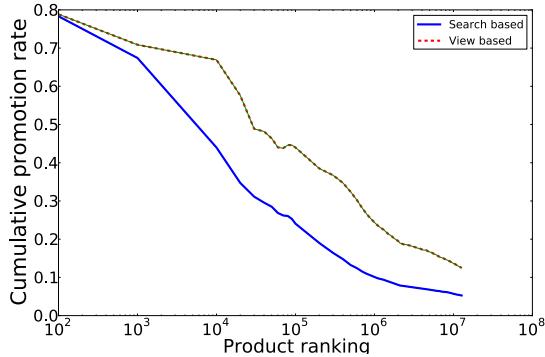


Figure 10: Cumulative promotion rates across varying product ranks for different Amazon pollution attacks.

mulative Success Rate for a given range of product rankings is defined as the ratio of the number of successfully promoted products to the number of target products in that range.

Figure 10 shows the cumulative promotion rate for different product rankings for the two different types of pollution attacks. As the target product decreases in popularity (*i.e.*, has a higher ranking position within its category) pollution attacks become less effective, but this phenomenon reflects a limitation of Amazon recommendation algorithms, not our attack. Products with low rankings might not be purchased as often; as a result, they may have few and weak co-visit and co-purchase relationships with other products. Our preliminary investigation finds that products which rank 2,000 or higher within their category have at least a 50% chance of being promoted by a visit-based pollution attack, and products with rankings 10,000 and higher have at least a 30% chance to be promoted using search-based attacks.

6 Related Work

To the best of our knowledge, the line of work most closely related to ours is black-hat search engine optimization (bSEO). Although sharing a common goal as search history pollution—illicitly promoting website rankings in search results—bSEO follows a completely different approach, exploiting a search engine’s reliance on crawled Web content. Blackhat SEO engineers the content of and links to Web pages to obtain a favorable ranking for search terms of interest [8]. Thus, techniques that address bSEO are unlikely to be effective against pollution attacks. On the other hand, because bSEO targets the general indexing and ranking process inside search engines, any successfully promoted website will be visible to all search engine users, potentially significantly boosting the volume of incoming traffic. Yet, effective bSEO campaigns typically involve support from

a complex network infrastructure, which may consist of hundreds of search-indexed websites (preferably with non-trivial reputations at established search engines) to coordinate and form a link farm [20]. These infrastructures not only require a considerable amount of money to build and maintain, but also take time to mature and reach their full effectiveness [8]. By contrast, launching a search history pollution attack is significantly easier.

We showed in Section 4 that a user’s personalized search results can be manipulated simply by issuing crafted search queries to Google. Without requiring any external support, the entire process happens instantly while the user is visiting the offending Web page. Although our attack targets individual search users (*i.e.*, the polluted result is only visible to individual victims), it by no means limits the scale of the victim population, especially if an exploit is placed on a high-profile, frequently visited website.

7 Discussion

Our current study has several limitations. Most notably, the scale of our experiments is modest, but because we typically randomly select the target items, we believe that the results of our experiments are representative, and that they illustrate the substantial potential impacts of pollution attacks. Similarly, our specific pollution attacks are fragile, as each service can take relatively simple steps to defend against them.

A possible defense against pollution attacks arises from the fact that cross-site request forgery can be stopped if requests to a website must carry tokens issued by the site. Enforcing this constraint, however, also prevents information and behaviors at third-party sites from being harvested for personalization and hampers the current trend of increasing the scope of data collection by websites for improved personalization. One short-term effect from this study may be that (some) websites will begin to consider the tradeoffs between the security and benefits of personalization.

YouTube in particular uses two separate HTTP requests to track a YouTube’s user viewing activity that are independent from the act of streaming of the video. One straightforward defense against pollution attacks is to monitor the time between the arrivals of the two HTTP requests. If YouTube finds the interval is substantially less than the length of the video, it could ignore the signal. An attacker can still always inject a short video or control the timing of the HTTP requests in an effort to bypass such a defense mechanism. We did notice that an injected short video can be used to promote multiple longer videos; for example, watching a single two-

second video⁵ causes YouTube to recommend several long videos.

8 Conclusion

In this paper, we present a new attack on personalized services that exploits the fact that personalized services use a user’s past history to customize content that they present to the user. Our attack pollutes a user’s history by using cross-site request forgery to stealthily inject and execute a set of targeted browsing activities in the user’s browser, so that when the user subsequently accesses the associated service specific content is promoted. We illustrate how an attacker can pollute a user’s history to promote certain content across three platforms. While our attack is simple, its impact can be significant if enough users’ histories are compromised.

As personalization algorithms and mechanisms increasingly control our interactions with the Internet, it is inevitable that they will become the targets of financially motivated attacks. While we demonstrate pollution attacks on only YouTube, Google, and Amazon, we believe that our methods are general and can be widely applied to services that leverage personalization technologies, such as Facebook, Twitter, Netflix, Pandora, etc. The attacks we present here are just the first few examples of potentially many possible attacks on personalization. With increasingly complex algorithms and data collection mechanisms aiming for ever higher financial stakes, there are bound to be vulnerabilities that will be exploited by motivated attackers. The age of innocence for personalization is over; we must now face the challenge of securing it.

Acknowledgments

This research was supported in part by the National Science Foundation under grants CNS-1255453, CNS-1255314, CNS-1111723, and CNS-0831300, and the Office of Naval Research under grant no. N000140911042. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation or the Office of Naval Research.

References

- [1] Amazon.com product identifiers. http://archive.org/details/asin_listing.
- [2] BENNETT, P. N., RADLINSKI, F., WHITE, R. W., AND YILMAZ, E. Inferring and using location metadata to personalize web search. In *Proceedings of the 34th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval* (2011).
- [3] CUTTS, M. Does Google use the “keywords” meta tag? <http://www.youtube.com/watch?v=jK7IPbnmvVU>.
- [4] DAOUD, M., TAMINE-LECHANI, L., AND BOUGHANEM, M. A session based personalized search using an ontological user profile. In *Proceedings of The 24th Annual ACM Symposium on Applied Computing* (2009).
- [5] DAVIDSON, J., LIEBALD, B., LIU, J., NANDY, P., VAN VLEET, T., GARGI, U., GUPTA, S., HE, Y., LAMBERT, M., LIVINGSTON, B., AND SAMPATH, D. The YouTube video recommendation system. In *Proceedings of the 4th ACM Conference on Recommender Systems* (2010).
- [6] DOU, Z., SONG, R., AND WEN, J.-R. A large-scale evaluation and analysis of personalized search strategies. In *Proceedings of the 16th ACM International Conference on the World Wide Web* (2007).
- [7] LIU, F., YU, C., AND MENG, W. Personalized web search by mapping user queries to categories. In *Proceedings of the 11th ACM International Conference on Information and Knowledge Management* (2002).
- [8] LU, L., PERDISCI, R., AND LEE, W. Surf: detecting and measuring search poisoning. In *Proceedings of the 18th ACM Conference on Computer and communications security* (2011).
- [9] MATTHIJS, N., AND RADLINSKI, F. Personalizing Web search using long term browsing history. In *The Fourth ACM International Conference on Web Search and Data Mining* (2011).
- [10] QIU, F., AND CHO, J. Automatic identification of user interest for personalized search. In *Proceedings of the 15th ACM International Conference on the World Wide Web* (2006).
- [11] SEARCH ENGINE LAND. Bing results get localized & personalized. <http://searchengineland.com/bing-results-get-localized-personalized-64284>.
- [12] SEARCH ENGINE LAND. Google now personalizes everyones search results. <http://searchengineland.com/google-now-personalizes-everyones-search-results-31195>.
- [13] SHIFLETT, C. Cross-site request forgeries. <http://shiflett.org/articles/cross-site-request-forgeries>, 2004.
- [14] SIEG, A., MOBASHER, B., AND BURKE, R. Web search personalization with ontological user profiles. In *Proceedings of the 16th ACM Conference on Conference on Information and Knowledge Management* (2007).
- [15] SONTAG, D., COLLINS-THOMPSON, K., BENNETT, P. N., WHITE, R. W., DUMAIS, S., AND BILLERBECK, B. Probabilistic models for personalizing Web search. In *Proceedings of the 5th ACM International Conference on Web Search and Data Mining* (2012).

⁵<http://www.youtube.com/watch?v=UPXK3AeRvKE>

- [16] SRIRAM, S., SHEN, X., AND ZHAI, C. A session-based search engine. In *Proceedings of the 27th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval* (2004).
- [17] TAN, C., GABRILOVICH, E., AND PANG, B. To each his own: personalized content selection based on text comprehensibility. In *Proceedings of the 5th ACM International Conference on Web Search and Data Mining* (2012).
- [18] TEEVAN, J., DUMAIS, S. T., AND HORVITZ, E. Personalizing search via automated analysis of interests and activities. In *Proceedings of the 28th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval* (2005).
- [19] VIDSTATSX. Youtube channel, subscriber, & video statistics. <http://vidstatsx.com/>.
- [20] WU, B., AND DAVISON, B. D. Identifying link farm spam pages. In *Proceedings of the Special Interest Tracks and Posters of the 14th ACM International Conference on the World Wide Web* (2005).

A Appendix

Here we provide more details regarding the actual exploit and test corpora for the search personalization attack.

A.1 Search Term Variance

As with the various product categories on Amazon, it is reasonable to expect that the effectiveness of search history pollution depends on the value of the search term being polluted. In other words, just as Amazon tightly controls the gift cards it recommends, it might be the case that a website cannot be promoted in Google’s search results as easily for a highly competitive search term, such as “laptop”, as it can for relatively uncontested search terms. To obtain an estimate of the value of different search terms, we again turned to Google’s AdWords Keyword Tool. The tool provides a function that associates a given search term with a level of competition. The competition level is a measure of how expensive it would be for URL to consistently pay enough to be ranked at the top of the list of advertisers for a particular search term. Competition level is expressed as a value from 0 to 1, with 0 having no competition and 1 having fierce competition.

Recall that out of the 2,136 webpages that we attempted to promote using a contextual pollution attack, 729 were successful. It is important to note that some of the promoted results were for the same initial search terms. Therefore, the number of search terms associated with the webpages are 1,740 and 606, respectively. As an example, we attempted to promote both `made-in-china.com` and `DHgate.com` with respect to

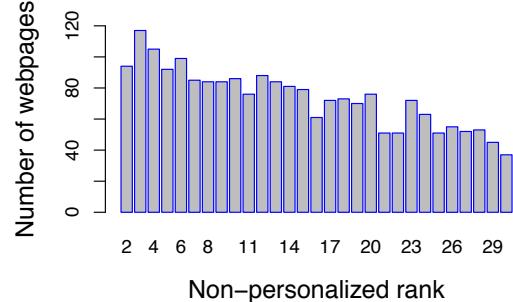


Figure 11: Google’s original rank distribution for the 2,136 webpages whose ranking we attempt to improve with contextual search history pollution.

the original search term “watch”. The keywords injected by the pollution attack differ, however, and are “China” and “China wholesale” respectively. For the persistent attacks, we were successful in promoting at least one returned website for 247 out of the 551 search terms.

Figure 12 shows the competition level distribution for both types of attacks. Figures 12(a) and 12(b) correspond to the 1,740 search terms associated with our entire contextual test corpus and the 606 search terms for which there was a website we could promote. Likewise, Figures 12(c) and 12(d) plot the competitiveness of the search terms for the 551 tested and the 247 successful persistent pollution attacks. Although the distributions are different between test corpora, in both cases, the distributions suggest there is no obvious correlation between search term competition or value and the likelihood of being able to launch a search history pollution attack.

A.2 Robustness

Because a contextual history pollution attack uses only a few recent search history entries to promote a website, the lifetime of this attack is limited to the period when Google’s personalization algorithm considers this contextual information. We empirically determine Google’s timeout threshold by injecting sets of contextual keywords into a Google search profile and then pausing Google’s history collection. We then search alternatively for two distinct search terms—one that we know is affected by the injected keywords, and another we know is not. We continue to search for these two terms, recording and time stamping all the search returns.

Our analysis of many such tests with different sets of search terms indicates that Google appears to enforce a ten-minute threshold on context-based personalized search, which thereby limits the scope of the contextual pollution attack. Similarly, there are limits on how many different searches can be conducted before the

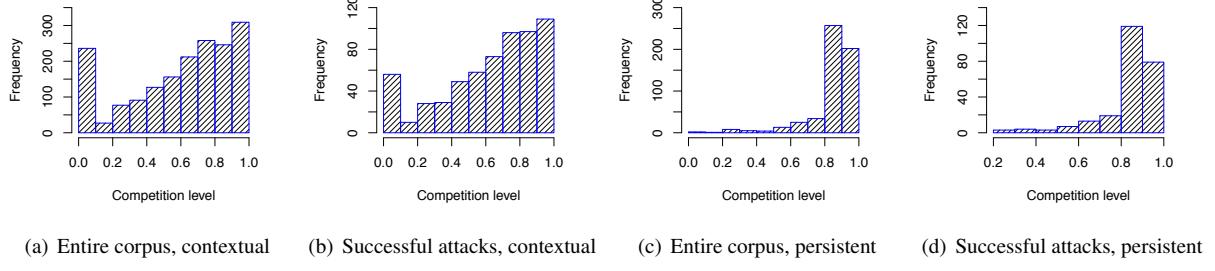


Figure 12: Distribution of search-term competition levels.

injected context is no longer used to personalize subsequent queries. Our initial testing indicates that personalization falls off after the fourth search. Hence, we conclude that the pollution attack can last for at most four subsequent queries or ten minutes, whichever comes first.

Our testing of persistent attacks shows that if a webpage remains promoted after several search terms, it will remain promoted for a long time. To determine how

long, we identified a set of 100 webpages and search terms on which we launch a successful persistent pollution attack. We then inject additional randomly selected trending keywords one-by-one and continually check whether the promotion remains. 72% of the websites remain promoted after 60 additional keywords, indicating that, when successful, persistent pollution attacks are likely to remain effective for quite some time.

Steal This Movie - Automatically Bypassing DRM Protection in Streaming Media Services

Ruoyu Wang^{1,2}, Yan Shoshitaishvili¹, Christopher Kruegel¹, and Giovanni Vigna¹

¹University of California, Santa Barbara, CA, USA

{fish, yans, chris, vigna}@cs.ucsb.edu

²Network Sciences and Cyberspace, Tsinghua University, Beijing, China

Abstract

Streaming movies online is quickly becoming the way in which users access video entertainment. This has been powered by the ubiquitous presence of the Internet and the availability of a number of hardware platforms that make access to movies convenient. Often, video-on-demand services use a digital rights management system to prevent the user from duplicating videos because much of the economic model of video stream services relies on the fact that the videos cannot easily be saved to permanent storage and (illegally) shared with other customers. In this paper, we introduce a general memory-based approach that circumvents the protections deployed by popular video-on-demand providers. We apply our approach to four different examples of streaming services: Amazon Instant Video, Hulu, Spotify, and Netflix and we demonstrate that, by using our technique, it is possible to break DRM protection in a semi-automated way.

1 Introduction

Digital Rights Management (DRM) is used by content distributors to restrict the way in which content may be used, transferred, and stored by users. This is done for several reasons. To begin with, content creators try to prevent content from reaching non-paying users through pirated copies of the content. While estimates of the cost impact of piracy are considered to be hugely inaccurate and research on this issue is inconclusive [41], they vary from \$446 million [14] to \$250 billion [8] for the movie and music industries in the US alone, and are far from insignificant in other parts of the world [12, 27]. Consequently, DRM is used to protect the media distributed through subscription-based services. In these services, such as Netflix, Spotify, Hulu, and Amazon Prime Instant Video, a user pays a recurring fee for access to a large database of media. This media can be played as much and as often as the user wishes, but becomes unavailable when a user stops paying for the service. The need to protect content in this scenario is obvious: if users can save the content for playback later

and simply cancel their account, the streaming service will lose substantial amounts of money.

DRM protection of media, especially passive media such as movies and music, has a fundamental difficulty. In order to enable the viewing of content, such content must at some point be decrypted. Different DRM schemes put this decryption at various stages of the media playback pipeline. Schemes such as High-bandwidth Digital Copy Protection (HDCP) [9] attempt to put this decryption outside of the reach of software and into the media playback hardware itself. However, use of these schemes are not always feasible. Specifically, many mobile devices, virtual machines, and lower-end computers do not support schemes such as HDCP. To function on such devices, DRM schemes must carry out decryption in software. On top of this limitation, hardware DRM schemes suffer from a problem of being too brittle against attacks. This was demonstrated, in the case of HDCP, with the compromise of the HDCP master keys [30], which rendered that DRM scheme useless. DRM schemes that do not rely on special hardware support are much more flexible in recovering from such compromises.

In order for an effective DRM scheme to be implemented, the possible attacks that it could succumb to must be well understood. In this paper, our goal is to examine one such attack: the identification of the transition between encrypted and decrypted data in the media player software.

To this end, we introduce MovieStealer, an approach for the automatic defeating of DRM in media playing programs. This approach takes advantage of several central intuitions. Our first intuition is that most data-processing operations, and specifically decryption operations, are carried out on buffers of data. This allows us to concentrate our analysis on the flow of data between buffers, making the analysis task considerably more tractable. Secondly, we observe that all popular media services of which we are aware utilize existing media codecs. We believe that this is because coming up with new codecs is a very complicated task, and many of the technologies behind efficient codecs

are patented. Additionally, high-definition codecs are extremely performance-intensive, and many media player devices rely on hardware support to decode them. This reliance on hardware support makes changing these codecs extremely difficult, making it far easier to license an existing codec than to create a new one. Utilizing this observation, we are able to identify buffers that contain data similar to what we would expect to be present in popular codecs. Our final observation is that we can distinguish three distinct classes of data by carrying out a statistical analysis: encrypted data (which will possess high randomness and high entropy), encoded media data (which will possess low randomness and high entropy), and other data (which will possess lower randomness and entropy).

We utilize these observations and develop an approach that tracks the flow of data from buffer to buffer within a program and identifies, using information theoretical techniques, the point in the program at which the data is decrypted. After automatically identifying this location in the program, MovieStealer dumps the decrypted stream. This stream can then be reconstructed into an unprotected media file and played back in an unauthorized media player.

Furthermore, we design optimizations that allow this online approach to be carried out on a running media player. Such optimizations are necessary due to the performance-demanding nature of the services that we target.

We implemented this approach and evaluated it on several streaming services, namely Netflix, Amazon Instant Video, Hulu, and Spotify. The latter is a music streaming service, while the others are video streaming services. All of these services are real-time, high-performance products which must be analyzed with low overhead in order to function. In all cases, MovieStealer is able to successfully pinpoint the decryption location and dump the decrypted stream. After this point, we consider the DRM protection to be *broken*. We have also implemented media file reconstructors to recover a playable media file.

To showcase our optimizations, we have also evaluated our approach against GPG, an open-source cryptographic suite.

The task of dumping the decrypted stream is completely automated. MovieStealer dynamically analyzes a program while it is used to play media, and dumps the decrypted streams. However, the final step of reconstruction requires a component to be developed for each protocol. We have implemented three such components to cover our four target streaming services. Since we consider the DRM to be bypassed as soon as we recover the decrypted data, automating this last step is out of the scope of our DRM analysis.

MovieStealer was developed in order to gain insight into the weaknesses of cryptographic DRM schemes. The implementation and utilization of such an approach for

piracy purposes is, of course, illegal. Our intention is not to aid illegal activity, and we present a discussion on ethics and legality in Section 7.

In summary, we make the following contributions:

1. We present an approach capable of automatically identifying and exploiting weaknesses in DRM scheme implementations by identifying cryptographic functionality in real-time, with no offline analysis, and duplicating the decrypted data.
2. To make such an approach work on performance-demanding applications and to reduce the amount of time the approach requires to locate the decrypted data, we utilize a set of optimizations that would be useful for any similar dynamic analysis approaches.
3. We show the effectiveness of this approach on four popular streaming services (Amazon Instant Video, Hulu, Netflix, and Spotify) and a general-purpose encryption tool (GPG).
4. To the best of our knowledge, we demonstrate the first publicly-described approach to duplicate PlayReady-protected content (such as modern versions of Netflix) without the use of a screen scraper. While we have been informed that there have been other attacks on PlayReady, we have been unable to find any public evidence of this fact.
5. Finally, we suggest several countermeasures that vendors of content protection schemes could employ to resist an attack such as MovieStealer. These range from technical solutions, attacking the technical details of our approach, to social solutions, such as increased use of watermarking to make piracy more prosecutable.

2 Background and Related Work

Over the last several decades, there has been an arms race between content owners, wishing to restrict the use of their content, and content consumers, who wish to use such content in an unrestricted way. New Digital Rights Management techniques are created on a regular basis, and new workarounds are quickly found to counter them. In this section, we survey several popular DRM techniques to better frame the research presented in this paper.

DRM schemes can generally be split into two classes: non-cryptographic DRM schemes and cryptographic DRM schemes. The former relies on verifying that the user is authorized to use the protected content by somehow utilizing a physical aspect of this content. Of course, this requires that the content ships with something like a manual, disk, or hardware dongle to use for verification. With the advent of digital distribution for software and multimedia, non-cryptographic DRM schemes have fallen in popularity.

On the other hand, cryptographic DRM schemes work by cryptographically verifying that the user attempting to access the content is authorized to do so. This approach is usable for digital distribution of content, and

is the paradigm according to which modern DRM schemes are developed.

In this paper, we include *link-protection* schemes, such as HDCP, which protect content in transit from being intercepted, with true *Digital Rights Management* systems, which ensure that only an authorized user is accessing the content in question. From the viewpoint of removing the protection, these two categories of content protection schemes are quite similar, and our system is general enough to handle both.

2.1 Cryptographic DRM Techniques

One of the early examples of cryptographic DRM techniques was the DVD Content Scramble System [44]. CSS is an encryption scheme to prevent DVDs from being played on unauthorized devices. It functioned by assigning a limited number of keys to manufacturers of DVD playing devices. These keys would then be used to decrypt the key chain of a given DVD and play back the video. CSS was broken in 1999 through cryptanalysis by a group of security researchers including Jon Lech Johansen (afterwards known as DVD Jon) [48]. This was done by reverse engineering a software DVD player to identify the encryption algorithm.

CSS was a forerunner of the type of copy protection that MovieStealer was created to analyze. While DRM schemes have since evolved to be more flexible, the basic premise remains the same: content is shipped in an encrypted form (whether through physical media or as a download), and is decrypted by an authorized player.

2.2 Hardware-based DRM

Hardware-based DRM has been around since the early days of copy protection. Early examples of this class of approaches are copy-protection dongles shipped with software [49]. Software protected by such dongles does not run without the presence of the dongle, and duplication of the dongle is infeasible. While early dongles simply contained static information that would be checked in software, modern dongles are complex cryptographic co-processors that actually carry out operations, such as decrypting the program code, on behalf of the protected program.

A specific adaptation of this into the realm of multimedia is HDCP [9], a link protection scheme which moves the decryption of media content outside of the computer. In a perfect implementation of this scheme, all content handled by the computer is always encrypted [11], and the decryption occurs in the media playback hardware (such as the monitor) itself. This would be problematic for our approach, but is not a problem in practice for several reasons. To begin with, all of our surveyed streaming services allow playback without HDCP. This is necessary because systems such as netbooks and virtual machines lack support for HDCP, and these services attempt to re-

DRM Platform	Encryption type		
	Connection	File	Stream
PlayReady	No	No	Yes
RTMPE	Yes	No	No
Spotify	Yes	Yes	No

Table 1: The present encryption locations for our analyzed platforms.

main compatible with them. Additionally, HDCP does not integrate seamlessly with the encryption used in the media streaming services of which we are aware. Encrypted content streamed from these services must first be decrypted, usually in memory, before being re-encrypted with HDCP. While some media devices exist that can handle this step in dedicated hardware, thus disallowing any access to the unencrypted stream, general purpose consumer devices are not among them. This means that on such devices, even in the presence of HDCP, MovieStealer can intercept the protected content on such devices while it is unencrypted. Finally, HDCP has been irrevocably broken with the leak of the HDCP master key. Hardware-based DRM schemes like HDCP are very hard to patch because they need to work on many devices that are not easily upgradeable. While the upgradeability of these devices might be improved in the future, there is currently no clear solution to this issue.

2.3 Streaming DRM Platforms

We analyze three different DRM schemes used by four platforms in this paper: Microsoft PlayReady (used by Netflix) [10], RTMPE (a link protection mechanism used by Adobe’s Flash streaming platforms such as Amazon Instant Video and Hulu) [3], and Spotify’s content protection [15].

We stress that our approach, as implemented by MovieStealer, does not exploit any particular vulnerability inherent to any single platform. Instead, these DRM schemes are vulnerable due to their inherent design, and not the inadequacies of any specific vendor or organization.

In this section, we provide some details about how these schemes function, in order to better frame our approach.

2.3.1 PlayReady

Microsoft’s PlayReady DRM, as implemented in its Silverlight streaming platform, which is used most prominently by Netflix, is a cross-platform content protection mechanism. PlayReady supports *individualization*, meaning that the media is encrypted with a content key, which is then encrypted with different keys for every user. Every time a user streams content on Silverlight, PlayReady provides an individualized *license*, ensuring that the content key can be decrypted and protected content viewed only by the intended recipient. The process to play back PlayReady-protected media using Silverlight comprises

several steps. To improve understanding, we present a high-level overview of these steps.

Metadata. To initialize playback, the Silverlight client requests metadata from the media server provider (such as Netflix). This metadata is a file that contains available resolutions and bit rates for the content, whether the payload is encrypted or not, the domain name of the license server, and the expiration time of the request.

License. If the metadata specifies that the payload is encrypted, the Silverlight client must acquire the license (containing the decryption key) from the license server, which is specified in the metadata. When a client sends the license request to the license server, the license server responds with the *Individualized Black Box* (IBX). The IBX is a custom, easily-upgradeable, and highly-obfuscated DLL that can be customized by individual content providers. Using the IBX, the client generates an individualized request to the license server.

The license server verifies this request and responds with a license. The client uses the IBX to decrypt the license and extract the content key, which is a 128-bit AES key.

Data. Having acquired the license, the client can now play back the protected content. This content takes the form of a fragmented MPEG-4 file transferred from the service provider. The protection works by encrypting the media stream data, while leaving the headers and stream metadata unencrypted. The data is encrypted using AES and is decrypted using the key acquired from the license server.

Performance. PlayReady has several performance requirements. To begin with, as with any network service, the client must be able to communicate with the server without letting the connection time out. Additionally, as a security measure against piracy, the IBX and corresponding license request have an expiration time, and the license will stop working after this timeout has elapsed. Finally, the media player (Netflix) itself has a minimum performance threshold, below which it will stop processing the stream and display an error. A successful online analysis of a PlayReady-protected media player must have a low-enough overhead to allow the player to meet these performance obligations.

2.3.2 RTMPE

RTMPE is a lightweight link protection mechanism developed by Adobe on top of the Real Time Messaging Protocol (RTMP) [2]. The addition to RTMP is a simple encryption layer.

Encryption layer. RTMPE generates a stream key using a Diffie-Hellman [29] key exchange. Once this key is agreed upon, the entire communication stream is encrypted using RC4 [1]. No extra encryption is done on the media itself.

Performance. Any online analyzer running against

RTMPE must be fast enough to allow the processing of the data stream without dropping the connection.

2.3.3 Spotify

Spotify implements a custom protection scheme to prevent duplication of their content. This scheme was reverse-engineered by the Despotify Project in their attempt to create an interoperable client [5]. The scheme uses a stream cipher to protect its communication, and, in addition, it encrypts each individual song.

Stream cipher. The Spotify client performs a key exchange with the server to create a key to be used for the remainder of the session. After the key is generated, the session is encrypted using a Shannon stream cipher [42].

Song encryption. Individual music files sent by Spotify in the encrypted stream are themselves encrypted with AES. The keys to this encryption are sent in the stream along with the music files. Upon receipt of a music file and its corresponding key, the Spotify client decrypts the file for playback. For offline playback, Spotify can cache this data.

Performance. An online analysis of Spotify must be fast enough to process the data stream without dropping the connection. Additionally, if the Spotify client runs too slow, it will mistakenly perceive that the connection to the server has been lost.

2.4 Bypassing DRM

As noted above, DRM methods tend to have unique workarounds, depending on their specific characteristics. For non-interactive multimedia, one general approach is called the Analog Hole [47]. The Analog Hole is a “flaw” in any DRM scheme, which is due to the fact that any media must eventually be consumed by a human. For example, a video will eventually have to be displayed on a screen and seen by someone’s eyes. In the simplest setting, a human could just record the protected music or movie with a microphone or a camcorder. Programs [4] exist that will even record a movie as it is playing on the screen by scraping the screen’s pixels.

However, since all the streaming media platforms known to us use lossy encoding for space and bandwidth-saving reasons, this type of DRM bypassing has the downside of a loss of quality due to the necessity to re-encode the captured audio and video. The only way to duplicate such content without quality loss is to capture the decrypted content after decryption but before decoding. There are two ways to do this: recovery of the keys used in the cryptographic process and the interception of the decrypted content. The former method requires approaches that may vary widely based on the DRM scheme and the type of encryption and key management used. Additionally, white-box cryptography [24] could be used to greatly complicate such an implementation by obscuring the usage of the cryptographic keys. The latter approach, which MovieStealer uses, allows us to intercept decrypted content irrespective

of the underlying encryption protocols. By doing this, it is possible to recover the original high-quality media sent by the media originator in a general way.

2.5 Cryptographic Function Identification

Since the identification of cryptographic functions is relevant to many other fields of study, and particularly relevant to malware analysis, other works have looked into identifying cryptographic routines.

An early approach to detecting decryption in memory is detailed by Noe Lutz [39]. This approach slows down the instrumented program by a factor of 2,400, an unacceptable slowdown for a high-performance media streaming application. Additionally, this approach detects encrypted data by measuring entropy. Such a detection would be unable to distinguish between encrypted and compressed (or, in our case, encoded) data.

Another approach, ReFormat [46], functions by detecting the flow of input data from a decryption routine to a handling routine followed by a flow of output data to an encryption routine. This approach does not work for our application domain for two reasons. Depending on the protocol, an analyzed media player might not necessarily encrypt a response. For example, the actual communication protocol of Microsoft’s Silverlight streaming platform is not encrypted. As such, the client only decrypts the encrypted stream data, but does not have to encrypt any responses. Furthermore, ReFormat detects the transition from encrypted to decrypted data based on the percentage of arithmetic and bitwise functions processing it. However, since the decrypted stream in a media player is passed on to the decoding step, this heuristic does not necessarily hold true.

Dispatcher [20] is an approach that analyzes the data flow of bots to determine their communication protocol. To find the decrypted data, the system uses a similar method to ReFormat. Dispatcher also functions through offline analysis, and would be unsuitable for our application.

Another interesting approach is presented by Caballero, et al [21]. This approach is geared toward *removing* the decryption and decoding functionality in a malware program to easier interact with it (in the paper, the authors interacted with the malware to find bugs). This is not applicable to our case, since we gain no benefit from interacting with a media player directly. In any case, the overhead in this approach likely makes it unfeasible for use on large programs such as media players, although we are unable to verify this as the authors did not publish the tool itself.

BCR [19] is a tool that implements an algorithm, similar to MovieStealer’s buffer detection, for detecting cryptographically-relevant loops and buffers. However, this approach has very heavy overhead, requires several similar executions, and relies heavily on offline analysis, which makes it ineffective for our target applications.

Aside from the performance issue, this is also due to the fact that streaming media players are not completely deterministic because of changes in behavior due to network latency, user interaction, and other factors. These factors are often hard or impossible to control between executions, especially with complicated DRM platforms such as PlayReady. Additionally, by avoiding this requirement in MovieStealer, we are able to simplify our approach by not worrying about buffers being relocated by ASLR.

The approach described by Grobert, et al [32] also detects cryptographic primitives, but is another offline analysis and would not be performant enough for a media player. Additionally, this approach, along with other similar approaches that check for cryptographic primitives, would be sensitive to white-box obfuscation.

Finally, a recent result in this area of research is Aligot [22]. Aligot works by identifying loops in programs, identifying data flow between such loops, and comparing the result against reference implementations of cryptographic primitives. However, it also functions in an offline manner and (considering the amount of time its offline phase requires) would be too slow for a media player to function. Additionally, Aligot requires that the program being analyzed utilize a standard implementation of cryptographic primitives, while our approach avoids such an assumption.

These existing approaches are not adequate for breaking DRM in media players. Since the media services that we analyze have real-time requirements, any approach must have minimal overhead to function. However, these approaches were mostly designed to be run against small, non-demanding malicious programs. In general, they have high overhead and rely on offline analysis while MovieStealer is designed to be a fast, online approach. Furthermore, these approaches do not address the distinction between encrypted and encoded/compressed data with the regards to randomness as opposed to entropy, which is necessary to locate the appropriate buffer from which to extract the decrypted-but-encoded media stream.

3 Approach

Our intuition is that the authors of a media player would reuse existing, proven codecs for ease of deployment, performance, and reliability. Thus, at some point during the processing, one should see the data decrypted and sent to the media codec for decoding. By examining the data as it flows through the authorized media player, one can detect the point at which the player transforms the data from an encrypted stream to an encoded stream. Once this location in the program is detected, the decrypted stream can be dumped and reconstructed. Our approach leverages this observation and provides an automatic mechanism to break the DRM schemes of several popular streaming services.

The process of copying protected content can be divided

into three separate phases:

1. Analyze the way in which the authorized media player handles the encrypted stream and identify the point at which the stream is decrypted.
2. Dump this decrypted stream.
3. Reconstruct the original media file from the decrypted stream.

Normally, the first step would have to be done once per media player (or, depending on the DRM implementation, once per media codec), while the second and third steps would be repeated for each dumped movie.

Given an authorized media player executable, MovieStealer will execute the binary, trace its execution flow, monitor and log its data access, recover loops and buffers (defined as consecutive bytes of data), recognize the decryption step, dump the decrypted data, and construct a media file with the unprotected content.

3.1 Stream Decryption Analysis

The first step in the copying of protected content is the analysis of the authorized media player’s processing of the encrypted stream. Of course, much of the code dealing with Digital Rights Management is heavily obfuscated, packed, or protected, and so our approach must be able to work with countermeasures such as dynamically generated functions. Therefore, the stream decryption detection is based on the dynamic analysis of the player application.

A media player processes a substantial amount of data in the course of downloading, decrypting, decoding, and playing media. Intuitively, such data, whether encrypted or decrypted, is stored in buffers in memory. While this data could conceivably be stored in evasive schemes (for example, splitting up buffers so that no two bytes are adjacent), we have not observed such evasiveness in the real-world applications that we have analyzed. Moreover, this would complicate the development process and would impede performance.

Thus, our goal in this step is to identify the location in the program where an encrypted buffer is turned into a decrypted buffer.

3.1.1 Loop Detection

The intuitive way to access data buffers is through a loop (or a loop-equivalent CPU instruction). As data decryption involves accessing the encrypted buffers, we would expect (and, indeed, this is what we have observed) it to be done using loops. Our intuition here is that a loop will exist that carries out a decryption operation on a small chunk of data. This loop (or, more precisely, its output) is what we are looking for. Hence, the first step of our solution is to automatically identify loops in the program.

Subsequent parts of our analysis work on loops rather than either functions or individual instructions for several reasons. First, loops are more likely to access a small

```
        mov eax, 0
.head:
        mov ebx, (0x1000, eax, 4)
        mov (0x2000, eax, 4), ebx
        inc eax
        cmp eax, 5
        jne .head
```

Table 2: An example of a loop.

```
        mov eax, 0xBAADF00D
        xor dword ptr [esp], eax
```

Table 3: An example of an implicit read by a loop.

number of buffers for a single purpose, while functions might access many buffers for several purposes. Secondly, a single instruction might only carry out a partial operation on the buffer. For example, a loop might carry out an entire decryption step while a single instruction in the loop might simply XOR two words together. Thus, by performing our analysis at the loop level, we can better see individual actions that a program carries out on its buffers. Thirdly, identifying functions within a program, without symbol tables and in the presence of obfuscation, is a complicated and error-prone task. We bypass this problem by operating on loops, which are more straightforward to identify. A loop can usually be identified as long as its basic blocks are executed at least twice.

Although we perform our analysis on loops, our approach is inspired by some basic concepts taken from function analysis. A good example is the input and output of a loop. We mark all data that a given loop reads as its input, and all data that it writes as its output. Table 2 gives an example of a loop that reads 5 dwds from the buffer starting at 0x1000 as input and writes them to the buffer at 0x2000 as output.

It is important to note that, in the x86 architecture, data can be an input to a loop without being explicitly read by that loop. For example, Table 3 demonstrates such a case, where *esp*, despite not being explicitly read by the code in question, is an input to the XOR operation.

Our approach assumes that the decryption process happens inside a loop. More specifically, we expect to find a loop in the authorized media player that has at least one encrypted buffer as an input and at least one decrypted buffer as an output. We expect this decryption to be done in a loop because such DRM schemes on media must process large volumes of data, and the most efficient way of processing such data is through a loop or loop-like instruction.

Detecting the loops. Our approach to detecting loops is mainly inspired by LoopProf [40]. LoopProf maintains a Basic Block Stack (BBLStack) per thread. A BBLStack is a stack of basic block addresses. Whenever a basic block

is executed, its start address is pushed to this stack, and when the basic block exits, the start address is popped.

Our analysis routine is called every time a Basic Block (BBL) is executed. The analysis routine attempts to find the same BBL by tracing back in the BBLStack. If the same BBL is found in the BBLStack, the basic blocks between this BBL and the top of the stack are considered to be a loop.

Note that when using this approach, some additional care must be taken to avoid misdetection of recursive calls as loops.

Although the basic idea of loop detection is simple, much attention was given to performance. We explain our optimizations in detail in Section 4.2.

Maintaining a call stack. As described in LoopProf, loop detection by BBLStack can cause our program to identify loops that occur across function boundaries, which is often the case with recursive function calls. While this would not break our approach, we have chosen to detect and remove these loops to improve the performance of the analysis, given that we have not seen any example of decryption being done in a recursive fashion.

Using our call stack, we only check BBL inside the current frame when searching for loops. We maintain this call stack for every thread by instrumenting every call and return instruction. Of course, functions do not have to use these instructions, in which case one would still detect the blocks as a loop. In the cases we have examined cases, this is acceptable for our approach.

Apart from aiding in loop detection, the presence of a call stack allows us to identify loops that are used for multiple purposes. For example, one loop could be used both to encrypt and to decrypt buffers. In this case, if the loop is called by one function, it behaves like a decryption routine, and has a random input as well as a non-random output. However, when called by a different function, the loop might behave like an encryption routine, and would throw off our detection if we did not differentiate between these two cases. Table 4 illustrates this scenario. Differentiating between these two scenarios is important for our analysis, since we analyze all of the data read and written by each loop in aggregate across several runs. Thus, we must differentiate between the two execution paths of this loop in order to distinguish the two different cases. Therefore, a loop is identified not only by its basic blocks, but also by the top several functions on the call stack at the time it was called.

Detecting unrolled loops. Loops are frequently unrolled for increased performance. Specifically, the first or last few iterations are often unrolled, with the rolled loop present in case more data needs to be processed. In order to detect unrolled loops, we take note of the basic blocks that were executed between any two loops. We later check if these basic blocks do operations on the same buffer as

```
void crypto_loop(void *key, void *in,
                 void *out, int len);

void encrypt() {
    crypto_loop("key", decrypted,
                encrypted, len);
}

void decrypt() {
    crypto_loop("key", encrypted,
                decrypted, len);
}
```

Table 4: An example of both the encryption and decryption being done in one loop.

either of the two loops.

3.1.2 Buffer Identification

According to prior work in the field of data reverse-engineering, most buffers are accessed in loops [43]. Thus, having identified loops, we must then identify the buffers on which they operate. For the sake of performance, and unlike the approaches outlined in Howard [43] and REWARDS [37], which track the base pointers of and offsets into buffers by instrumenting every instruction, our approach is based on recording and analyzing reading and writing operations inside a loop. This is similar to what is implemented in BCR [19]. In addition, several heuristic methods are applied to improve the detection of the buffers. By applying these heuristics, even complex buffers such as the key permutation array used in RC4, which is accessed neither consecutively nor completely in most cases, can be identified by our approach.

Fetching memory access patterns. When MovieStealer is analyzing a loop, it dynamically instruments each read and write within that loop. For each such read and write, we record the target memory location that it accesses, the instruction pointer where the access occurs, and the size of the read or write. Note that some instructions, when called with specific operands, execute both a read and a write operation.

Whenever control flow leaves the loop, we move on to analyzing the loop’s memory access patterns.

Analyzing memory access patterns. A loop can access a buffer in one of several different access patterns. Our approach focuses on detecting the following ones:

1. Consecutively accessing the buffer byte-by-byte.
2. Consecutively accessing the buffer dword-by-dword.
3. Consecutively accessing the buffer at single-byte offsets and reading a dword at a time.
4. Consecutively accessing the buffer using multimedia CPU extensions, such as SSE instructions [45].

Address	Step 1	Step 2	Step 3
0x1000	O (size 4)	C (element size 4)	C (element size 4)
0x1004	O (size 4)		
0x1008	O (size 4)	C (element size 4)	C (element size 4)
0x100c	O (size 4)	C (element size 4)	
0x1010	O (size 4)	O (size 4)	
0x1010	O (size 8)	O (size 8)	C (element size 8)
0x1014			

Table 5: An example of the creation of composite buffers (C) from the memory read operations (original buffers O) of the code in Table 2.

5. Accessing the buffer in a predictable pattern. For example, two first bytes out of every three consecutive bytes are read in a buffer.
6. Accessing the buffer in an unpredictable pattern. In most such cases, the buffer is not fully accessed during the execution of a loop. For instance, accessing the key permutation array of the RC4 [1] algorithm.

To identify a read or write buffer, we perform our analysis in several steps. First, we classify each memory region affected by an individual memory access as an *original* buffer and sort them by their starting memory addresses. Then, we merge these buffers into *composite* buffers by recursively applying the following steps until there are no more candidates for merging. As we merge the buffers, we attempt to determine the size of the elements in each buffer.

- Two *original* buffers are merged if they are adjacent and are of equal size. In this case, the element size for the resulting composite buffer is set to the size of the two original buffers.
- Two *original* buffers are merged if they overlap, are of equal size, and their size is divisible by the size of the overlapping portion. In this case, the element size for the resulting compound buffer is set to the size of the overlapping region between the two original buffers.
- An *original* buffer is merged with a *composite* buffer if they are adjacent and the element size of the composite buffer is equal to the size of the original buffer.

This is applied recursively until there are no more *original* buffers that can be merged. At this point, any remaining *original* buffers are reclassified as *compound* buffers with an element size equal to their length. An example of this is detailed in Table 5.

This step merges the individual memory accesses into a preliminary representation of buffers. The sizes of these *composite* buffers will vary, but will be divisible by their element size. This representation is finalized in the next step, where the composite buffers are merged.

Merging composite buffers. Due to the way in which some buffers are accessed, they will be split into several

composite buffers in the previous step. One example of this is the key permutation array used in RC4 [1]. This buffer usually has a size of 256 bytes, and is not likely to be completely read or written if there are less than 100 bytes to be decrypted. One approach is to aggregate the memory accesses over several different calls to the function to identify the buffer, but that brings up questions of when to terminate such an analysis. Therefore we use a simple heuristic to better identify such buffers: Given two existing composite buffers C and D, where buffer C starts at $addr_c$ and has a size of $size_c$, while buffer D starts at $addr_d$ and has a size of $size_d$, and $addr_d > addr_c$. We define the term gap ratio as the size of gap between buffers C and D divided by the sum of sizes of buffers C and D:

$$gratio(C, D) = \frac{addr_d - (addr_c + size_c)}{size_c + size_d}$$

We then perform the following algorithm:

1. If C and D have the same element size, and they are adjacent, they will be merged into a larger buffer.
2. If C and D are not adjacent, and they have the same element size, they will be merged if the gap ratio is less than 0.2. We determined this number experimentally. Of course, setting this threshold to a value too large will create false positives in the buffer detection (and will add noise to our subsequent statistical testing), while leaving it too small will cause us to miss parts of the buffers.

This algorithm is applied on the set of composite buffers until no more buffers can be merged.

Tracking unrolled loops. After the composite buffers are merged, we add any memory accesses done by blocks that are adjacent to the buffers and are identical to the blocks inside a loop. This allows us to catch the marginal parts of buffers that are modified by unrolled loops.

Data paths. After this step, we will have obtained a full list of buffers that are accessed inside each loop. We define a data path as an input-output buffer pair within a loop. A loop could have multiple data paths, as shown in Table 6. In the absence of detailed data-flow analysis, we conclude that every input buffer and every output buffer in a loop make a data path. Thus, in a loop with N input and M output buffers, we will have $N \times M$ data paths.

3.1.3 Decryption Detection

After identifying the buffers and the paths between them, the next step is to identify the buffer that holds the decrypted content. While a full analysis of every data path in a real-world application could be unfeasible due to the complexity of modern media players, we can utilize several heuristics to identify the data path that contains the decryption of the protected content. First of all, the data path that

```

        mov eax, 0
.head:
        inc eax
        mov ebx, (0x1000, eax, 4)
        mov (0x2000, eax, 4), ebx
        mov ebx, (0x3000, eax, 4)
        mov (0x4000, eax, 4), ebx
        cmp eax, 10
        jne .head

```

Table 6: An example of a loop with four data paths: 0x1000 to 0x2000, 0x1000 to 0x4000, 0x3000 to 0x2000, and 0x3000 to 0x4000.

Stage	Input		Output	
	E	R	E	R
Download	high	high	high	high
Decrypt	high	high	high	low
Decode	high	low	low	low

Table 7: The entropy (E) and randomness (R) of data paths when playing a protected media file.

we are looking for should have a similar throughput to the size of the media file. Additionally, since we are looking for a data path that has an encrypted input and a decrypted (but encoded with a media codec) output, we can utilize information theoretical properties of the buffers to improve our analysis.

We perform this step on the aggregated input and the aggregated output buffers of each data path. That is, we append all of the input and all of the output of a given data path across multiple executions of the loop in question, resulting in an overall input buffer and an overall output buffer. This allows us, for example, to analyze all of the output of a given operation across the runtime of the program. In the case of a decryption function, this will allow us to collect all of the decrypted content.

Entropy test. The data path in which we are interested will have an encrypted input buffer and a decrypted but encoded output buffer. The input buffer, being encrypted, will have very high entropy. The output buffer, being encoded (and effectively compressed), will also have very high entropy. We use this property to further filter out unrelated data paths.

This also helps filter out the decoding step. Media codecs are highly compressive functions, resulting in high-entropy buffers. On the contrary, a buffer of, for example, YUV color frames is likely to have a comparatively low entropy.

Randomness test. A fundamental property of encrypted data is that it is indistinguishable from random data. This is called *ciphertext indistinguishability*, and is a basic requirement for a secure cryptosystem [31]. Further-

more, randomness is very difficult to achieve, and is not a feature of data encoding algorithms. Such algorithms, which are essentially specialized compression algorithms, produce data with high entropy but low randomness. Thus, as shown in Table 7, we can distinguish between the encrypted and decrypted stream by using a randomness test.

The Chi-Square randomness test is one such test, designed to determine if a given input is random. Often used to test the randomness of pseudo-random number generators, we use it to determine whether or not the content of a buffer is encrypted. The implementation details of the Chi-Square randomness test is detailed by Donald Knuth [33] and its application to randomness testing is presented by L’Ecuyer, et al [36]. Our approach does not rely on the implementation details of the randomness test, and we have omitted them in the interest of space. Furthermore, the Chi-Square randomness test is not the only one that can be used; any measure of randomness of a buffer can be utilized for this purpose.

One important consideration is the amount of data that we should collect before performing our randomness test. A commonly accepted rule for the Chi-Square randomness test, mentioned by Knuth [33], is that given n , the number of observations, and p_s , the probability that n is observed to be in category s , the expected value $n \times p_s$ is greater than 5 for all categories s . We consider the contents of each buffer one byte at a time, giving us 256 categories of s . According to calculations presented by Knuth, we would need to collect 320 kilobytes of data for a reliable test. In fact, we carried out an empirical analysis of the minimum amount of data that needed to be analyzed to be confident of avoiding misdetection. The analysis determined that a safe threshold to avoid misclassifying random data as non-random is 800 kilobytes, and a safe threshold to avoid misclassifying non-random media data as random is 3800 bytes, both of which are easily feasible for any sort of media playback.

We have observed that the Chi-Square randomness test returns extremely low values (very close to 1.0) for encrypted data, and very high values (in the thousands) for encoded data.

3.2 Dumping the Stream

After the previous steps, we are able to identify the specific data path that has the encrypted input and the decrypted, but decoded, output. Then, our system instruments the authorized media player and dumps the output buffer.

3.3 Reconstructing the File

Finally, with the decrypted data available, the last step is to reconstruct the media file. In the trivial case, the DRM scheme works by encrypting the entire media file wholesale. This is simple to recover because the decrypted buffer

that we dump will then contain the whole, unprotected file. However, this is not the case in general. For example, the approach used in Microsoft’s PlayReady DRM encrypts just the media stream, leaving the headers and metadata decrypted. Thus, the decrypted stream will contain the raw media stream, which cannot be directly played by a media player. In the general case, this is a problem of program analysis and writing an automated tool to reconstruct the file given an unknown protocol is quite complicated.

In order to recreate the media file in these circumstances, knowledge of the streaming/DRM protocol is required. For example, knowing that PlayReady encrypts the media stream, we wrote a reconstruction plugin to reconstruct the file with the newly decrypted media stream (and the already decrypted headers and metadata) so that it would be playable in conventional media players.

Depending on the protocol and the expertise of the operator, this stage can involve reading documentation, reverse engineering, and file analysis. However, at this point the automated decryption of the content, which is the central aim of our paper, is already completed. While the media content will need to be reconstructed for every dumped file, the development process is only required once per DRM platform (or, depending on the implementation, once per DRM platform/media codec combination).

4 Implementation

We implemented our system using the PIN [38] Dynamic Binary Instrumentation framework. We chose this tool for its ease of development, but our approach can be implemented on top of a full-system emulator such as QEMU in order to avoid anti-DBI techniques by the media-playing applications. However, the use of QEMU would raise the question of performance, since it is not clear if QEMU’s dynamic recompilation of binaries can match the performance of PIN. Additionally, though our system is implemented under the x86 architecture, the approach is easily translatable to other architectures as well.

We will detail some of the specific implementation details related to the individual DRM platforms that we analyzed. Additionally, in the course of implementing our approach, we made several implementation decisions, which we will discuss hereinafter. After describing these, we will also detail optimizations that we used to increase the speed of our approach.

4.1 DRM Platforms

We specifically investigated three DRM platforms: Flash video with Amazon Instant Video and Hulu, Microsoft PlayReady with Netflix, and Spotify. Here we will give an overview of the protocols and the tools we developed to support them.

4.1.1 Flash RTMPE

Amazon Instant Video and Hulu both use the RTMPE protocol, developed by Adobe, to transmit video. RTMPE works by encrypting the whole media file on the fly before sending it across the network.

Since the entire file is encrypted, reconstructing it did not present a challenge because it was decrypted in a continuous manner in one function.

4.1.2 Microsoft PlayReady

Netflix uses Microsoft’s Silverlight PlayReady DRM to protect its content. PlayReady presents several challenges.

Relocating code. In Silverlight, the actual routine used for decrypting AAC audio, WMV video and H.264 video is frequently relocated inside the process’ memory space. We assume that this is done to frustrate would-be pirates. An additional benefit, given the required flexibility of the surrounding code, would be the ability to dynamically update the decryption routine over the network. However, we did not observe the latter ever occurring. Ironically, this evasive behavior gives us a clear signal that such code is *interesting*, and could enable us to prioritize it in our analysis.

To cover the case of relocated code, we identify such loops based on the non-relocating portions of their call stack and the hash of their basic blocks. This allows us to handle relocating code automatically as part of the normal analysis.

Disabling adaptive streaming. Netflix automatically adjusts the quality of the video stream to compensate for bandwidth and CPU inadequacies. This can result in a varied quality in the generated media file, which would lead to a confusing subsequent media consumption experience. Furthermore, because the MovieStealer implementation is extremely CPU-intensive, such adaptive streaming features will invariably select the stream with the worst available quality.

Our solution to this problem, specifically for Silverlight-based streaming services, is to use a Winsock introspection tool named Esperanza [7] to inspect the browser’s traffic and filter the lower-bandwidth stream options out of the metadata. While this is a protocol-specific fix, a generalized version of this would be outside of the scope of this paper.

Partial encryption. PlayReady is hard to work with because it only encrypts the raw stream data of its media files. Header information and meta-data is not encrypted. Because of this, the decrypted file must be pieced back together by combining the original metadata and the dumped stream. Furthermore, some of the headers have to be modified to reflect the fact that the file is no longer encrypted.

4.1.3 Spotify

Spotify’s distinguishing factor is the use of the Themida packer to frustrate our DBI platform. Since our instrumen-

tation is done dynamically, we would normally be able to copy protected content of packed programs. However, because Themida contains some evasive behavior that is able to confuse PIN, we had to use the OllyDBG debugger to first neutralize Themida’s evasiveness by hiding PIN’s presence. After this, we were able to extract music from Spotify, despite it being packed with the Themida packer. While this is not automated in *our* implementation, automating such anti-debugging practices is quite feasible.

Spotify encrypts its music files as a whole, so reconstructing them is straightforward.

4.2 Optimization

A basic implementation of our approach is able to detect and duplicate decrypted data in a program, but is not yet performant enough to analyze media players. To remedy this, we developed several optimizations. There are three stages that can be optimized: the selection of loops to analyze, the instrumentation and analysis of the loops themselves, and general performance optimizations.

Admittedly, some of the optimizations presented here are not automated. Specifically, limiting code coverage requires some domain knowledge to determine which code should be instrumented. However, MovieStealer can still function, albeit at a reduced speed, without this optimization.

4.2.1 Intelligent Loop Selection

Due to the overhead involved in instrumenting the memory operations of a loop and keeping track of the data that a loop is accessing, MovieStealer instruments a limited number of loops at a time. While a loop is instrumented, data from its buffers is saved and passed to our analyses. When a loop is determined by the randomness test to not be the decryption loop, or it is eliminated by one of the optimizations mentioned below, we discard the tracked state and instrument the next loop. To minimize the time necessary for MovieStealer to find the decryption routine, we need to select these loops in the most optimal manner.

Limit code coverage. Code coverage greatly influences the execution speed of MovieStealer. In most cases it is not necessary to instrument every module of the target process. For example, only the core libraries of Silverlight need to be instrumented to bypass PlayReady DRM, rather than the whole set of libraries of the browser. To reduce the number of loops that need to be analyzed, we only select the ones in the suspected DRM code, cutting out a significant amount of overhead.

On-demand instrumentation. Although we limit code coverage, there are still many instructions that are executed only once during initialization, which have nothing to do with decryption. Instrumenting and analyzing such loops would be a waste of resources. Inspired by PrivacyScope [50], we have designed MovieStealer to start after the program has initially loaded. After we load the

authorized media player, we start MovieStealer and begin the media playback process. Thus, the initialization code will not be analyzed and MovieStealer will immediately begin zeroing in on the actual decryption functionality. We have observed that this significantly reduces the amount of loops that MovieStealer has to instrument and analyze.

Loop execution frequency. Additionally, we have observed that, in a streaming media player, the decryption routine is usually one of the most frequently-executed loops. This is because additional media is constantly being loaded over the network and must be constantly decrypted. On the other hand, loops pertaining to other functionality (for example, UI processing), are executed comparatively less frequently. To take advantage of this, we prioritize these loops for analysis ahead of less-frequently executed loops.

Static instruction analysis. As described in prior works [20, 46], code that carries out cryptographic functionality tends to utilize a large amount of certain types of operations. To optimize our analysis, we statically analyze the amount of arithmetic and bitwise operations in every loop and de-prioritize loops that lack such operations.

Additionally, we have observed that decryption routines often contain unrolled loops for increased performance. As such, we assign a higher priority to loops that are unrolled. We statically detect unrolled loops by detecting a repeating pattern of instructions before or after a loop body. While this is a very simplistic approach to unrolling detection, we feel that it is adequate. It works for the code that we have observed in our analyses, and if it fails to detect an unrolled loop, such a loop would still be analyzed later.

Loop hashing. In order to allow MovieStealer to function over several executions of a program, we save the results of our analyses for analyzed loops. We identify loops using a tuple consisting of the offsets of the basic blocks comprising the loop from the base address of their module, and the name of the module. When the analysis of a loop is finished, the results are saved before the state for the loop is discarded. This allows us to keep results over multiple executions of MovieStealer in case it takes an exceptionally long time to identify the decryption point. While this optimization can be useful, we did not run into any cases where we had to rely on it.

The astute reader will note that the relocating loops of DRM schemes such as Microsoft PlayReady will not be successfully recorded by this approach. However, this optimization would still allow us to avoid reanalyzing the majority of loops in a program, and being able to thus focus on just the relocating ones will greatly reduce the time required for MovieStealer to identify the decryption loop.

4.2.2 Improved Instrumentation

Intelligently selecting loops to instrument greatly improves MovieStealer’s performance, but lots of time is

Stage	Input bandwidth	Output bandwidth
Download	roughly S	roughly S
Decrypt	roughly S	roughly S
Decode	roughly S	greater than S

Table 8: The bandwidth of data paths when playing a protected media file of size S .

still spent analyzing loops that turn out to be unrelated to decryption. For loops that handle a lot of data, this data needs to be analyzed in a performant fashion. However, when instrumenting loops that do not handle much data, much time is spent waiting to acquire enough data for the statistical tests. To further optimize this, we created several approaches to increase the performance of loop instrumentations and to decrease the time necessary to arrive at a classification.

Bandwidth filtering. Since protected media needs to be decrypted before being played, we should be able to find the decryption loop more efficiently by examining its data throughput. We define the input bandwidth of a data path as the amount of data in the aggregated input buffer and the output bandwidth of a data path as the amount of data in the aggregated output buffer. In Table 8, we detail the steps that an authorized media player takes when playing protected content, along with the expected input and output bandwidth of these functions. Intuitively, a loop that is decrypting the network traffic should have a similar bandwidth to the network traffic itself.

We carry out a bandwidth check on each instrumented loop every two seconds and compare it against the network traffic (for streaming media players) or the disk traffic (for GPG). Empirically, we determined that it’s safe to discard a loop after 20 seconds if it fails the bandwidth test at least 60% of the time. A loop is considered to have failed a bandwidth test if its bandwidth is not within 60% of the expected bandwidth.

Avoiding unnecessary data copying. For the randomness test, the entropy test and the data dumping, we must record data chunks that are read or written during the loop execution, as described before. Since memory operations happen very frequently, performance is critical in tracking these reads and writes. Our approach must fulfill these basic requirements:

1. Moving, copying and modifying data as little as possible.
2. Imposing as little overhead as possible for addressing the buffer.

We did not include thread safety as one of the basic requirements, as in real-world media players few buffers are accessed simultaneously by multiple threads. We assume that programs that do access buffers concurrently will handle their own synchronization.

We have different strategies for reading and writing. For written data, rather than logging what is written, two variables holding the starting address and the ending address are maintained for every buffer. Each time a buffer write occurs, we update the starting address and ending address so that they correctly reflect the start and end positions that are written. As we expect these buffers to be consecutive, there is no problem with expanding the margins over bytes that are not read yet. For the randomness and entropy tests, MovieStealer analyzes every byte in the buffers between the start and end positions.

For content that is read out of buffers, we have a different strategy. As data being read during a loop might be overwritten inside the same loop, our write-buffer strategy does not always work. Hence it is necessary for MovieStealer to not only record the memory ranges, but also record the data located at the memory ranges at the time that reading happens. It is important to note that memory reading is not always consecutive nor always starts from the beginning of the buffer. Thus, through the single run of a loop, only parts of a buffer might be updated. To achieve better performance, we try to avoid re-copying unchanged data. This is done by treating each buffer as a concatenation of 4,096-byte blocks. As a loop executes, we mark the blocks that it modifies, and copy only the modified blocks when it exits. Our copied-off buffer is an array of pointers to these blocks. Any unchanged blocks on a new run are stored as pointers to previously-copied versions of that data.

4.2.3 Other Optimizations

Call stack optimization. To improve performance, a *call stack key* is maintained for each thread, and is updated each time a call or ret instruction is executed. When a new function is called, its start address is XORed onto the call stack key when the function is added to the call stack. When the program is about to return from a function, we pop the function from the call stack and wipe it from the call stack key by XORing its start address again. This way, we can use the call stack key instead of the whole call stack to identify a given loop. A dword comparison has considerably less overhead than a list comparison and, in practice, we have not seen any call stack key collisions due to this in our experiments.

This optimization is especially useful in loop selection, loop analysis, and data dumping.

5 Evaluation

In the course of our evaluation, we strived to demonstrate two things: that our optimizations work and are effective at improving performance, and that MovieStealer is an effective tool for bypassing the DRM of streaming media services. Since most of the streaming media services do not function at all without our optimizations, we

ran the optimization evaluation on GPG, an open source cryptography suite. GPG has fewer real-time processing requirements than real-world media players and as such works despite high overhead from unoptimized analyses.

We evaluated MovieStealer’s effectiveness on a series of online streaming services, including Netflix, Hulu, Amazon Video, and Spotify. Our experiments consisted of loading the streaming application (in all cases except for Spotify, this was done by visiting the appropriate webpage in the browser. Spotify is a stand-alone application), starting MovieStealer, and playing a video or a song. MovieStealer would then pinpoint the decryption location and, on future runs, would begin dumping the media file. The reconstructor would then be run to create a playable media file. We verified that the media file was playable by playing it in a different, unauthorized player.

We carried out three experiments for each DRM platform, treating Hulu and Amazon Video as a single platform. For each experiment, we started MovieStealer from scratch. We recorded the number of loops identified, the loops analyzed before MovieStealer zeroed in on the sensitive loop, the total amount of analyzed loops that contain detected buffers, the total number of buffers identified, the total number of decryption loops that MovieStealer identified, and the total time until data could start being dumped. In all of the experiments, the loop responsible for decrypting the encrypted content was partially unrolled as a performance optimization.

To the best of our knowledge, MovieStealer is the first publicly described approach with the ability to successfully copy content protected by Microsoft PlayReady DRM without screen scraping techniques, as well as the first implementation to do cryptographic identification and copying of content at runtime.

MovieStealer was able to function on all DRM approaches that we evaluated.

Effect of optimizations. We carried out our optimization evaluations by executing MovieStealer against GPG as it decrypted a video file. First, we measured the performance of MovieStealer with all optimizations enabled, then measured the performance of first the callstack key optimization and then the code coverage limit optimization by running MovieStealer with all other optimizations enabled, and finally enabled some of our optimizations one-by-one to demonstrate their effects. The results can be seen in Table 9.

Necessary optimizations. Some of our optimizations were necessary to get the media players to function at all. As described in Section 2.3, these media players are high-performance pieces of software with some real-time requirements. For example, Netflix implements content expiration and has minimum performance requirements below which it will not play videos, and an unoptimized approach fails to meet such requirements. We have found

Optimizations enabled	LT	S
All	7	31
All but callstack key	6	47
All but limit code coverage	10	34
Only limit code coverage	9	65
Only static instruction analysis	10	49
Only bandwidth filtering	35	180
Only execution frequency	40	3,480

Table 9: Results for GPG. LT = loops traced, S = total seconds before the decryption loop was identified.

Experiment no.	1	2	3
Loops identified	1,529	1,258	1,647
Buffers identified	14	6	1
Loops traced	46	35	62
Seconds elapsed	281	146	175

Table 10: Results for Amazon Video and Hulu

Experiment no.	1	2	3
Loops identified	2,876	2,274	2,950
Buffers identified	88	80	152
Loops traced	8	58	54
Seconds elapsed	86	110	191

Table 11: Results for Netflix

Experiment no.	1	2	3
Loops identified	2,305	1,845	1,667
Buffers identified	60	69	63
Loops traced	224	204	138
Seconds elapsed	536	739	578

Table 12: Results for Spotify

that it is possible to analyze the streaming media players by enabling, at minimum, all of the loop selection optimizations.

Non-determinism. Non-determinism is introduced into the results from several sources. To begin with, the programs in question are complex and multi-threaded, and rely on external resources to function. This means that the sequence that code is executed (and that MovieStealer analyzes it) in varies between runs.

Additionally, MovieStealer starts on demand, so it might begin analyzing different parts of the program in different runs. This will also make it analyze code in different order. Finally, the code relocations used by PlayReady DRM adds extra indeterminism to the mix. However, this does not have an effect on the final, successful decryption result.

6 Discussion

The expected use of an approach such as MovieStealer would be to save streamed movies either for later watching or for sharing with others. The latter approach is, of course, illegal. Our intention is not to aid illegal activity, and we discuss this further in Section 7.

It is also important to stress that in order for MovieStealer to function, the user must be authorized to play the content in the first place.

One possibility for future direction is a look into automatically cracking HDCP-protected content. Since the master keys are leaked, it might be possible to analyze encrypted-encrypted data paths and attempt to automatically use the HDCP keys to decrypt the content further. With the relatively low amount of buffers identified in the video experiments, this might be feasible from a performance standpoint. This would allow MovieStealer to function on devices with dedicated hardware for hiding content as it's re-encrypted for HDCP.

Another potential direction would be use MovieStealer to automatically recover encryption keys from running software. After detecting a decryption loop, MovieStealer could check the other inputs to that loop or to other loops that touched the encrypted buffer to determine if such inputs are the keys to the encryption.

Furthermore, it would be interesting to investigate the use of our approach to inform systems such as Inspector Gadget [34] in order to automatically export the encryption/decryption functionality of programs.

6.1 Countermeasures

Although our approach proved to be effective on current online streaming services, there are steps that could be taken by the authors of the DRM schemes to protect themselves against MovieStealer.

Anti-debugging. Applying extreme anti-debugging and anti-DBI techniques would prevent our implementation, *in its current form*, from working. However, nothing prevents one from implementing MovieStealer in a full-system emulator such as QEMU [16], rendering the approach immune to such evasions.

Attacking our loop detection. There are several ways to prevent MovieStealer from properly detecting loops within a program. A full unrolling of relative loops could effectively prevent the real loop from being detected by MovieStealer. However, full unrolling will result in loss of flexibility of the loop, and detection might still be possible using pattern matching approaches. Alternatively, protecting sensitive program modules by using virtual machine interpreted instructions would be very effective, as most of our loop identification approaches would not work. However, the performance penalty for doing this would likely be unacceptable.

Attacking the buffer detection. We cannot properly analyze a buffer that has a nonconsecutive layout in memory. For example, if a buffer only occupies one byte every three bytes, these bytes will not be identified as a valid byte array, let alone a buffer. We have not seen these techniques being used, and implementing them will likely carry an overhead cost. However, it is a definite possibility with modern hardware.

Along these lines, an effective countermeasure would be a functional hardware DRM scheme. However, it is not clear how to implement this in a way flexible enough to be resistant to events such as key leaks while being secure enough to be resistant to bypass.

Attacking the decryption detection. One very effective countermeasure would be to intersperse non-random data in the encrypted buffers, and to insert random data into the decrypted buffer. This would lower the randomness of the encrypted buffer and raise the randomness of the decrypted buffer, possibly defeating our analysis. The decoder would then be modified to ignore the inserted random bytes so that it can successfully replay the video. It is important to note that this approach would require a modification of the decoder, as removing the random bytes beforehand (and reducing the randomness of the buffer in question) would trigger MovieStealer's decryption detection.

Attacking the pirates. Watermarking has proven to be incredibly effective in tracking piracy. The originator could watermark the media [28, 17, 18], and in the event of piracy, the pirates could be identified by this watermark. This is a very effective technique, and it has been used to successfully track down pirates [13, 6]. While some research has been done toward the circumvention of watermarks [26, 35], a watermark-related arms race might be easier for content providers than the design of mechanisms to counteract approaches similar to MovieStealer.

7 Ethical and Legal Issues

In this section, we discuss the ethical and legal implications of our work.

First of all, obviously our work was never motivated by the desire to obtain protection-free copies of the media for re-distribution (piracy) or to create and distribute tools that would allow others to bypass content protection mechanisms.

Our goal was to analyze the security of the cryptographic mechanisms used by these emerging services, and to develop an approach that would demonstrate the general fallacy behind these schemes, in the hope that our findings would prompt the development of new, more secure approaches to content protection that are not vulnerable to our attack. This is especially important if cryptography-based protection mechanisms are touted to "protect" user-generated content (e.g., independent movies distributed

exclusively through streamed media) and give to the content authors (i.e., the users of the distribution service) a false sense of security with regards to the possibility of malicious third parties stealing their content.

The legality of this research is tightly related to the location where the research is performed. For example, there are some subtle but important differences between the laws in the United States and the laws of the European Union and Italy [23].

The research was carried out in the United States, and hence, it falls under the Digital Millennium Copyright Act [25]. The DMCA prohibits the circumvention of content protection mechanisms, but includes explicitly protection of security research (referred to as “Encryption Research” – see Section 1201(g) of the DMCA.) We feel that this research falls under this protection and is therefore legal. Citing from the DMCA document: “Factors in determining exemption: In determining whether a person qualifies for the exemption under paragraph (2), the factors to be considered shall include the information derived from the encryption research was disseminated, and if so, whether it was disseminated in a manner reasonably calculated to advance the state of knowledge or development of encryption technology, versus whether it was disseminated in a manner that facilitates infringement under this title or a violation of applicable law other than this section, including a violation of privacy or breach of security.”

We feel that the way in which this research is disseminated is clearly focused on advancing research and not to facilitate infringement. In fact, we have chosen not to publicly distribute the source code of our tool or to provide ways to easily attack specific technologies. In addition, with the help of the Electronic Frontier Foundation, we contacted each of the companies involved in order to disclose these DRM workarounds responsibly. Microsoft was notified because they are the vendor of the Silverlight DRM used in Netflix. Adobe was notified because they are the vendor the RTME implementation for Amazon and Hulu. Netflix, Amazon, and Hulu were notified because the DRM being bypassed is used by their services. Spotify was in the unique position of falling into both categories. Of course, we contacted them as well.

Of the companies contacted, Netflix, Amazon, and Hulu did not respond to our initial or follow-up contacts, nor when contacted through EFF’s channels of communication. However, Microsoft, Adobe, and Spotify responded, acknowledged the issues, and discussed workarounds. All three companies reviewed our work, provided comments for this paper, and encouraged its publication, for which we are grateful.

In summary, our goal is to improve the state-of-the-art in cryptographic protection and not to create and distribute tools for the violation of copyright laws.

8 Conclusions

In this paper, we have proposed MovieStealer, a novel approach to automated DRM removal from streaming media by taking advantage of the need to decrypt content before playing. Additionally, we have outlined optimizations to make such DRM removal feasible to do in real-time, and have demonstrated its effectiveness against four streaming media services utilizing three different DRM schemes.

Acknowledgements We would like to thank representatives from Microsoft, Spotify, and Adobe for their feedback in regards to the drafts that we sent them. Additionally, we are eternally grateful to the EFF and UCSB’s legal counsel for their help with legal and ethical concerns during the publication process. Finally, we thank Dr. Jianwei Zhuge for his advice.

This work was supported by the Office of Naval Research (ONR) under Grant N00014-12-1-0165 and under grant N00014-09-1-1042, and the National Science Foundation (NSF) under grants CNS-0845559 and CNS 0905537, and by Secure Business Austria. This work was partly supported by Project 61003127 supported by NSFC.

References

- [1] RC4, 1994. <http://web.archive.org/web/20080404222417/http://cyberpunks.venona.com/date/1994/09/msg00304.html>.
- [2] RTMP, 2009. http://wwwimages.adobe.com/www.adobe.com/content/dam/Adobe/en/devnet/rtmp/pdf/rtmp_specification_1.0.pdf.
- [3] Adobe RTMPE, 2012. <http://lkcl.net/rtmp/RTMPE.txt>.
- [4] Audials software, 2012. http://audials.com/en/how_to_record_stream_capture_music_videos_movies_from/netflix.html.
- [5] The despotify project, 2012. <http://despotify.se/>.
- [6] E-city nabs pirates using thomson watermarking tech, 2012. <http://businessofcinema.com/bollywood-news/ecity-nabs%2Dpirates-using-thomson%2Dwatermarking%2Dttech/27167>.
- [7] Esperanza project, 2012. <http://code.google.com/p/esperanza>.
- [8] Freakonomics - How Much Do Music And Movie Piracy Really Hurt the U.S. Economy?, 2012. <http://www.freakonomics.com/2012/01/12/how-much%2Ddo-music%2Dand-movie%2Dpiracy%2Dreally-hurt%2Dthe-u-s%2Deconomy/>.
- [9] High-bandwidth Digital Content Protection System - Interface Independent Adaptation - 2.2, 2012. http://www.digital-cp.com/files/static_page_files/6FEA6756-1A4B%2DB294\%2DD0494084C37A637F/HDCP\%20Interface\%20Independent\%20Adaptation\%20Specification\%20Rev2_2_FINAL.pdf.
- [10] Microsoft PlayReady DRM, 2012. [http://msdn.microsoft.com/en-us/library/cc838192\(VS.95\).aspx](http://msdn.microsoft.com/en-us/library/cc838192(VS.95).aspx).
- [11] Microsoft protected media path, 2012. http://scholar.google.com/scholar?hl=en&q=protected+media+path&btnG=&as_sdtp=1%2C5&as_sdtp=1.
- [12] Nation of unrepentant pirates costs \$900m, 2012. <http://www.smh.com.au/technology/technology-news/nation-of%2Dunrepentant-pirates-costs\%2D900m-20110305\%2D1bix5.html>.

- [13] Porn studio awarded 1.5 million from man who shared 10 movies, 2012. http://www.slate.com/blogs/trending/2012/11/02/kywan_fisher_flava_works_wins_1_5_million_in_biggest_ever_torrent_judgement.html.
- [14] SOPA: How much does online piracy really cost the economy?, 2012. http://www.washingtonpost.com/blogs/ezra-klein/post/how-much-%2Ddoes-online-piracy%2Dreally-cost-the%2Deconomy/2012/01/05/gIQAXknNdP_blog.html.
- [15] Spotify DRM, 2012. <http://www.defectivebydesign.org/spotify>.
- [16] F. Bellard. QEMU, a fast and portable dynamic translator. USENIX, 2005.
- [17] J. Bloom and C. Polyzois. Watermarking to track motion picture theft. In *Signals, Systems and Computers, 2004. Conference Record of the Thirty-Eighth Asilomar Conference on*, volume 1, pages 363–367. IEEE, 2004.
- [18] L. Boney, A. Tewfik, and K. Hamdy. Digital watermarks for audio signals. In *Multimedia Computing and Systems, 1996., Proceedings of the Third IEEE International Conference on*, pages 473–480. IEEE, 1996.
- [19] J. Caballero, N. M. Johnson, S. McCamant, and D. Song. Binary code extraction and interface identification for security applications. Technical report, DTIC Document, 2009.
- [20] J. Caballero, P. Poosankam, C. Kreibich, and D. Song. Dispatcher: Enabling active botnet infiltration using automatic protocol reverse-engineering. In *Proceedings of the 16th ACM conference on Computer and communications security*, pages 621–634. ACM, 2009.
- [21] J. Caballero, P. Poosankam, S. McCamant, D. Song, et al. Input generation via decomposition and re-stitching: Finding bugs in malware. In *Proceedings of the 17th ACM conference on Computer and communications security*, pages 413–425. ACM, 2010.
- [22] J. Calvet, J. M. Fernandez, and J.-Y. Marion. Aligot: Cryptographic function identification in obfuscated binary programs. 2012.
- [23] R. Caso. *Digital Rights Management: Il commercio delle informazioni digitali tra contratto e diritto d'autore*. CEDAM, 2006.
- [24] S. Chow, P. Eisen, H. Johnson, and P. C. Van Oorschot. A white-box des implementation for drm applications. In *Digital Rights Management*, pages 1–15. Springer, 2003.
- [25] U. S. Congress. Digital Millennium Copyright Act, October 1998.
- [26] I. Cox and J. Linnartz. Some general methods for tampering with watermarks. *Selected Areas in Communications, IEEE Journal on*, 16(4):587–593, 1998.
- [27] G. Danby. Key issues for the new parliament 2010 - copyright and piracy, 2010. http://www.parliament.uk/documents/commons/lib/research/key_issues/Key%20Issues%20Copyright%20and%20piracy.pdf.
- [28] E. Diehl and T. Furun. © watermark: Closing the analog hole. In *Consumer Electronics, 2003. ICCE. 2003 IEEE International Conference on*, pages 52–53. IEEE, 2003.
- [29] W. Diffie and M. Hellman. New directions in cryptography. *Information Theory, IEEE Transactions on*, 22(6):644–654, 1976.
- [30] Engadget. HDCP ‘master key’ supposedly released, unlocks HDTV copy protection permanently, 2010. <http://www.engadget.com/2010/09/14/hdcp-master%2Dkey-supposedly%2Dreleased-unlocks%2Dhdtv-copy%2Dprotect/>.
- [31] O. Goldreich. *Foundations of Cryptography: Volume 2, Basic Applications*, volume 2. Cambridge university press, 2004.
- [32] F. Gröbert, C. Willems, and T. Holz. Automated identification of cryptographic primitives in binary programs. In *Recent Advances in Intrusion Detection*, pages 41–60. Springer, 2011.
- [33] D. Knuth. *The art of computer programming*. addison-Wesley, 2006.
- [34] C. Kolbitsch, T. Holz, C. Kruegel, and E. Kirda. Inspector gadget: Automated extraction of proprietary gadgets from malware binaries. In *Security and Privacy (SP), 2010 IEEE Symposium on*, pages 29–44. IEEE, 2010.
- [35] G. Langelaar, R. Lagendijk, and J. Biemond. Removing spatial spread spectrum watermarks. In *Proceedings of the European Signal Processing Conference (EUSIPCO98), Rodes, Greece*, 1998.
- [36] P. L'Ecuyer. Testing random number generators. In *Winter Simulation Conference: Proceedings of the 24 th conference on Winter simulation*, volume 13, pages 305–313, 1992.
- [37] Z. Lin, X. Zhang, and D. Xu. Automatic reverse engineering of data structures from binary execution. 2010.
- [38] C. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *ACM SIGPLAN Notices*, volume 40, pages 190–200. ACM, 2005.
- [39] N. Lutz. Towards revealing attackers intent by automatically decrypting network traffic. *Master's thesis, ETH Zuerich*, 2008.
- [40] T. Moseley, D. Grunwald, D. Connors, R. Ramamujam, V. Tovinkere, and R. Peri. LoopProf: Dynamic techniques for loop detection and profiling. In *Proceedings of the 2006 Workshop on Binary Instrumentation and Applications (WBIA)*, 2006.
- [41] M. Peitz and P. Waelbroeck. Piracy of digital products: A critical review of the economics literature. 2003.
- [42] C. Shannon. Communication theory of secrecy systems. *Bell system technical journal*, 28(4):656–715, 1949.
- [43] A. Slowinska, T. Stancescu, and H. Bos. Howard: a dynamic excavator for reverse engineering data structures. In *Proceedings of NDSS*, 2011.
- [44] F. A. Stevenson. Cryptanalysis of contents scrambling system, 2000. http://web.archive.org/web/2000030200206/www.dvd-copy.com/news/cryptanalysis_of_contents_scrambling_system.htm.
- [45] S. Thakur and T. Huff. Internet streaming SIMD extensions. *Computer*, 32(12):26–34, 1999.
- [46] Z. Wang, X. Jiang, W. Cui, X. Wang, and M. Grace. Reformat: Automatic reverse engineering of encrypted messages. *Computer Security—ESORICS 2009*, pages 200–215, 2009.
- [47] Wikipedia. Analog Hole - Wikipedia, the free encyclopedia, 2012. [Online; accessed 09-Nov-2012].
- [48] Wikipedia. DeCSS - Wikipedia, the free encyclopedia, 2012. [Online; accessed 09-Nov-2012].
- [49] Wikipedia. Software protection dongle - Wikipedia, the free encyclopedia, 2012. [Online; accessed 09-Nov-2012].
- [50] Y. Zhu, J. Jung, D. Song, T. Kohno, and D. Wetherall. Privacy scope: A precise information flow tracking system for finding application leaks. Technical report, Tech. Rep. EECS-2009-145, Department of Computer Science, UC Berkeley, 2009.