Davide Balzarotti
Salvatore J. Stolfo
Marco Cova (Eds.)

LNCS 7462

Research in Attacks, Intrusions, and Defenses

15th International Symposium, RAID 2012
Amsterdam, The Netherlands, September 2012
Proceedings

Springer

Lecture Notes in Computer Science 7462

Commenced Publication in 1973
Founding and Former Series Editors:
Gerhard Goos, Juris Hartmanis, and Jan van Leeuwen

Davide Balzarotti Salvatore J. Stolfo
Marco Cova (Eds.)

Research in Attacks, Intrusions, and Defenses

15th International Symposium, RAID 2012
Amsterdam, The Netherlands, September 12-14, 2012
Proceedings

Springer

Volume Editors

Davide Balzarotti
Institut Eurécom
2229 Route des Cretes
06560 Sophia-Antipolis Cedex, France
E-mail: davide.balzarotti@eurecom.fr

Salvatore J. Stolfo
Columbia University
Department of Computer Science
1214 Amsterdam Avenue, M.C. 0401
New York, NY 10027-7003, USA
E-mail: sal@cs.columbia.edu

Marco Cova
University of Birmingham
School of Computer Science
Edgbaston, Birmingham, B15 2TT, UK
E-mail: m.cova@cs.bham.ac.uk

ISSN 0302-9743 e-ISSN 1611-3349
ISBN 978-3-642-33337-8 e-ISBN 978-3-642-33338-5
DOI 10.1007/978-3-642-33338-5
Springer Heidelberg Dordrecht London New York

Library of Congress Control Number: 2012946390

CR Subject Classification (1998): C.2.0, D.4.6, K.6.5, K.4.4, H.2.7, C.2, H.4, H.5.3

LNCS Sublibrary: SL 4 – Security and Cryptology

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India

Printed on acid-free paper

Springer is part of Springer Science+Business Media (www.springer.com)

Preface

On behalf of the Program Committee, it is our pleasure to present the proceedings of the 15th International Symposium on Research in Attacks, Intrusions, and Defenses (RAID 2012), which took place in Amsterdam, The Netherlands, during September 12–14, 2012.

For its 15th anniversary, the RAID symposium changed its name from "Recent Advances in Intrusion Detection" to "Research in Attacks, Intrusions and Defenses." The new name reflects the broader scope of the conference that now aims at bringing together leading researchers and practitioners from academia, government, and industry to discuss novel research contributions related to any area of computer and information security.

This year, there were six technical sessions presenting full research papers on virtualization security, attacks and defenses, host and network security, fraud detection and underground economy, Web security, and intrusion detection systems. Furthermore, there was an invited talk to present the most influential paper presented in the first five years of the RAID conference and a poster session presenting emerging research areas and case studies.

The RAID 2012 Program Committee received 84 full paper submissions from all over the world. All submissions were carefully reviewed by independent reviewers on the basis of technical quality, topic, novelty, and overall balance. The final decision took place at a Program Committee meeting on May 24 in San Francisco, California, where 18 papers were eventually selected for presentation at the conference and publication in the proceedings. The symposium also accepted 12 poster presentations, reporting early-stage research, demonstration of applications, or case studies. An extended abstract of each accepted poster is included in the proceedings.

The success of RAID 2012 depended on the joint effort of many people. We would like to thank all the authors of submitted papers and posters. We would also like to thank the Program Committee members and additional reviewers, who volunteered their time to carefully evaluate all the submissions. Furthermore, we would like to thank the General Chair, Bruno Crispo, for handling the conference arrangements; Marco Cova for handling the publication process; William Robertson and Sotiris Ioannidis for publicizing the conference; Stefano Ortolani for maintaining the conference website and helping with the local arrangements; and the Vrije Universiteit in Amsterdam for hosting the conference. We would also like to thank our sponsor Symantec, for supporting the conference.

September 2012

Davide Balzarotti
Salvatore Stolfo

Organization

Organizing Committee

General Chair

Bruno Crispo	University of Trento, Italy

Program Chair

Davide Balzarotti	Eurecom, France

Program Co-chair

Salvatore Stolfo	Columbia University, USA

Publication Chair

Marco Cova	University of Birmingham, UK

Publicity Chairs

Sotiris Ioannidis	FORTH, Greece
William Robertson	Northeastern University, USA

Local Chair

Stefano Ortolani	Vrije Universiteit, The Netherlands

Program Committee

Anil Somayaji	Carleton University, Canada
Michael Bailey	University of Michigan, USA
Mihai Christodorescu	IBM T.J. Watson Research Center, USA
Juan Caballero	IMDEA Software Software Institute, Spain
Srdjan Capkun	ETH Zurich, Switzerland
Marco Cova	University of Birmingham, UK
Nick Feamster	Georgia Tech, USA
Debin Gao	Singapore Management University, Singapore
Guofei Gu	Texas A&M, USA
Guillaume Hiet	Supelec, France
Thorsten Holz	Ruhr University Bochum, Germany
Sotiris Ioannidis	FORTH, Greece
Gregor Maier	ICSI, USA

Christian Kreibich	ICSI, USA
Christopher Kruegel	UC Santa Barbara, USA
Andrea Lanzi	EURECOM, France
Corrado Leita	Symantec Research, France
Benjamin Livshits	Microsoft Research, USA
Fabian Monrose	University of North Carolina at Chapel Hill, USA
Benjamin Morin	ANSSI, France
Roberto Perdisci	University of Georgia, USA
William Robertson	Northeastern University, USA
Abhinav Srivastava	AT&T Labs-Research, USA
Angelos Stavrou	George Mason University, USA
Dongyan Xu	Purdue, USA
Charles Wright	MIT Lincoln Laboratory, USA

External Reviewers

Elias Athanasopoulos
Ryad Benadjila
Gehana Booth
Shakeel Butt
Zhui Deng
Zhongshu Gu
Amin Hassanzadeh
Sharath Hiremagalore
Johannes Hoffmann
George Kontaxis
Lazaros Koromilas
Kyu Hyung Lee
Zhiqiang Lin
Luc des trois Maisons
Saran Neti
Antonis Papadogiannakis
Fei Peng
Junghwan Rhee
R. Scott Robertson
Zacharias Tzermias
Pu Shi
Seungwon Shin
Dannie Stanley
George Vassiliadis
Tielei Wang
Zhaohui Wang
Chao Yang
Jialong Zhang

Steering Committee

Chair

Marc Dacier	Symantec, USA

Members

Herve Debar	Telecom SudParis, France
Deborah Frincke	Pacific Northwest National Lab, USA
Ming-Yuh Huang	Northwest Security Institute, USA
Somesh Jha	University of Wisconsin, USA
Erland Jonsson	Chalmers, Sweden
Engin Kirda	Northeastern University, USA
Christopher Kruegel	UC Santa Barbara, USA
Wenke Lee	Georgia Tech, USA

Richard Lippmann	MIT Lincoln Laboratory, USA
Ludovic Me	Supelec, France
Robin Sommer	ICSI/LBNL, USA
Alfonso Valdes	SRI International, USA
Giovanni Vigna	UC Santa Barbara, USA
Andreas Wespi	IBM Research, Switzerland
S. Felix Wu	UC Davis, USA
Diego Zamboni	HP Enterprise Services, Mexico

Table of Contents

Virtualization

Trusted VM Snapshots in Untrusted Cloud Infrastructures 1
Abhinav Srivastava, Himanshu Raj, Jonathon Giffin, and Paul England

Secure and Robust Monitoring of Virtual Machines through Guest-Assisted Introspection 22
Martim Carbone, Matthew Conover, Bruce Montague, and Wenke Lee

Assessing the Trustworthiness of Drivers 42
Shengzhi Zhang and Peng Liu

Attacks and Defenses

Industrial Espionage and Targeted Attacks: Understanding the Characteristics of an Escalating Threat 64
Olivier Thonnard, Leyla Bilge, Gavin O'Gorman, Seán Kiernan, and Martin Lee

Memory Errors: The Past, the Present, and the Future 86
Victor van der Veen, Nitish dutt-Sharma, Lorenzo Cavallaro, and Herbert Bos

A Memory Access Validation Scheme against Payload Injection Attacks 107
Dongkyun Ahn and Gyungho Lee

Host and Network Security

Dione: A Flexible Disk Monitoring and Analysis Framework 127
Jennifer Mankin and David Kaeli

AK-PPM: An Authenticated Packet Attribution Scheme for Mobile Ad Hoc Networks 147
Zhi Xu, Hungyuan Hsu, Xin Chen, Sencun Zhu, and Ali R. Hurson

Fraud Detection and Underground Economy

Paying for Piracy? An Analysis of One-Click Hosters' Controversial Reward Schemes 169
Tobias Lauinger, Engin Kirda, and Pietro Michiardi

Proactive Discovery of Phishing Related Domain Names 190
Samuel Marchal, Jérôme François, Radu State, and Thomas Engel

Evaluating Electricity Theft Detectors in Smart Grid Networks 210
Daisuke Mashima and Alvaro A. Cárdenas

Web Security

PoisonAmplifier: A Guided Approach of Discovering Compromised Websites through Reversing Search Poisoning Attacks 230
Jialong Zhang, Chao Yang, Zhaoyan Xu, and Guofei Gu

DEMACRO: Defense against Malicious Cross-Domain Requests 254
Sebastian Lekies, Nick Nikiforakis, Walter Tighzert, Frank Piessens, and Martin Johns

FlashDetect: ActionScript 3 Malware Detection 274
Timon Van Overveldt, Christopher Kruegel, and Giovanni Vigna

Intrusion Detection

ALERT-ID: Analyze Logs of the Network Element in Real Time for Intrusion Detection... 294
Jie Chu, Zihui Ge, Richard Huber, Ping Ji, Jennifer Yates, and Yung-Chao Yu

A Lone Wolf No More: Supporting Network Intrusion Detection with Real-Time Intelligence 314
Johanna Amann, Robin Sommer, Aashish Sharma, and Seth Hall

GPP-Grep: High-Speed Regular Expression Processing Engine on General Purpose Processors 334
Victor C. Valgenti, Jatin Chhugani, Yan Sun, Nadathur Satish, Min Sik Kim, Changkyu Kim, and Pradeep Dubey

N-Gram against the Machine: On the Feasibility of the N-Gram Network Analysis for Binary Protocols 354
Dina Hadžiosmanović, Lorenzo Simionato, Damiano Bolzoni, Emmanuele Zambon, and Sandro Etalle

Poster Abstracts

Online Social Networks, a Criminals Multipurpose Toolbox (Poster Abstract) 374
Shah Mahmood and Yvo Desmedt

The Triple-Channel Model: Toward Robust and Efficient Advanced Botnets (Poster Abstract) 376
Cui Xiang, Shi Jinqiao, Liao Peng, and Liu Chaoge

Network Security Analysis Method Taking into Account the Usage Information (Poster Abstract) 378
Wu Jinyu, Yin Lihua, and Fang Binxing

Automatic Covert Channel Detection in Asbestos System (Poster Abstract) 380
Shuyuan Jin, Zhi Yang, and Xiang Cui

EFA for Efficient Regular Expression Matching in NIDS (Poster Abstract) 382
Dengke Qiao, Tingwen Liu, Yong Sun, and Li Guo

Distress Detection (Poster Abstract) 384
Mark Vella, Sotirios Terzis, and Marc Roper

Trie Data Structure to Compare Traffic Payload in a Supervised Anomaly Detection System (Poster Abstract) 386
Jenny Andrea Pinto Sánchez and Luis Javier García Villalba

Towards Automated Forensic Event Reconstruction of Malicious Code (Poster Abstract) 388
Ahmed F. Shosha, Joshua I. James, Chen-Ching Liu, and Pavel Gladyshev

Accurate Recovery of Functions in a Retargetable Decompiler (Poster Abstract) 390
Lukáš Ďurfina, Jakub Křoustek, Petr Zemek, and Břetislav Kábele

Improvement of an Anagram Based NIDS by Reducing the Storage Space of Bloom Filters (Poster Abstract) 393
Hugo Villanúa Vega, Jorge Maestre Vidal, Jaime Daniel Mejía Castro, and Luis Javier García Villalba

Concurrency Optimization for NIDS (Poster Abstract) 395
Jorge Maestre Vidal, Hugo Villanúa Vega, Jaime Daniel Mejía Castro, and Luis Javier García Villalba

Malware Detection System by Payload Analysis of Network Traffic (Poster Abstract) 397
Luis Javier García Villalba, Jaime Daniel Mejía Castro, Ana Lucila Sandoval Orozco, and Javier Martínez Puentes

Author Index 399

Trusted VM Snapshots in Untrusted Cloud Infrastructures

Abhinav Srivastava[1], Himanshu Raj[2], Jonathon Giffin[3], and Paul England[2]

[1] AT&T Labs–Research
[2] Microsoft Research
[3] School of Computer Science, Georgia Institute of Technology
abhinav@research.att.com, {rhim,pengland}@microsoft.com,
giffin@cc.gatech.edu

Abstract. A cloud customer's inability to verifiably trust an infrastructure provider with the security of its data inhibits adoption of cloud computing. Customers could establish trust with secure runtime integrity measurements of their virtual machines (VMs). The runtime state of a VM, captured via a snapshot, is used for integrity measurement, migration, malware detection, correctness validation, and other purposes. However, commodity virtualized environments operate the snapshot service from a privileged VM. In public cloud environments, a compromised privileged VM or its potentially malicious administrators can easily subvert the integrity of a customer VMs snapshot. To this end, we present *HyperShot*, a hypervisor-based system that captures VM snapshots whose integrity cannot be compromised by a rogue privileged VM or its administrators. HyperShot additionally generates trusted snapshots of the privileged VM itself, thus contributing to the increased security and trustworthiness of the entire cloud infrastructure.

1 Introduction

The lack of verifiable trust between cloud customers and infrastructure providers is a basic security deficiency of cloud computing [2, 14, 25, 31]. Customers relinquish control over their code, data, and computation when they move from a self-hosted environment to the cloud. Recent research has proposed various techniques to protect customers' resources in the cloud [10, 19, 34]. We consider an alternate way to establish trust and provide customers with control: runtime verification of their rented virtual machines' (VMs) integrity. The runtime state of a VM, captured via a *snapshot*, can be used for runtime integrity measurement [3, 5, 12], forensic analysis [42], migration [8], and debugging [39]. The snapshot allows customers to know the state of their VMs and establishes trust in the cloud environment.

Today's commodity virtualization environments such as Xen [4], VMware [41], and Hyper-V [24] capture a consistent runtime state of a virtual machine at a point in time via the *snapshot* service. Each of these virtualized environments generates a snapshot from a privileged VM, such as dom0 [4] or the root VM [24]. Since the privileged VM, together with the hypervisor and the hardware, comprises the infrastructure platform's trusted computing base (TCB), the snapshot generation service and the resulting snapshot stored in the privileged VM are inherently trusted.

D. Balzarotti, S. Stolfo, and M. Cova (Eds.): RAID 2012, LNCS 7462, pp. 1–21, 2012.

Such unconditional trust in privileged VMs is sensible for a self-hosted private virtualized infrastructure, but it does not generalize to today's virtualization-based public cloud computing platforms due to different administrative control and restrictions [6]. Cloud customers or tenants must trust the root VM to generate a true snapshot of their VMs. Given that the root VM runs a full fledged operating system and a set of user-level tools with elevated privileges, any vulnerability present in that substantial software collection may be exploited by attackers or malware to compromise the integrity of the snapshot process or the snapshot information itself. Customers must hence also rely on an infrastructure provider's administrators and administration policies to properly manage the security of privileged VMs. Yet, the problem of malicious administrators is serious enough that researchers have proposed technical measures to support multiple-person control for administration [30].

In this work, we focus on the trust issues between customers and providers and propose a system called *HyperShot* that generates trusted snapshots of VMs *even in the presence of adversarial root VMs and administrators*. HyperShot only relies on a hypervisor and does not include the root VM in its TCB. Trust in the hypervisor itself is established via hypervisor launch using Trusted Execution Technology (TXT) [15]. Our design departs from the existing snapshot generation approaches in that HyperShot operates from the hypervisor rather than from the root VM. Since the hypervisor executes at a higher privilege level than the root VM, this design protects the snapshot service from the compromised root VM. HyperShot protects the integrity of the generated snapshot with a cryptographic signature from a Trusted Platform Module (TPM) that incorporates measurements of all trusted components and a hash of the snapshot itself. Since the TPM is exclusively under the control of our hypervisor, these measurements enable a verifying entity or customer to establish trust in a VM's snapshot. Since HyperShot does not trust the root VM and its administrators, it extends the same snapshot generation functionality and trust guarantees to the root VM itself.

To allow customers and providers to obtain verifiable snapshots of VMs executing in the cloud, we present a snapshot protocol that operates with *minimal trust in the cloud infrastructure itself*. The design of HyperShot decouples the snapshot generation process from the verification or analysis process. A customer can generate the snapshot in the cloud and can perform analysis at its own end or assign the verification duties to a third party. This design reduces burden on cloud infrastructure providers since customers can choose analysis software independent of the cloud provider. HyperShot's snapshot of the root VM enables customers to verify the integrity of a provider's management VMs in a measurable way with the help of third parties trusted by both providers and customers. We believe that this significantly contributes to the trustworthiness and reliability of the cloud infrastructure provider as a whole.

To demonstrate the feasibility of our ideas, we have implemented a HyperShot prototype based on Microsoft's Hyper-V virtualization infrastructure; any other hypervisor such as Xen or VMware is equally suitable for our work. The platform TCB in commodity virtualized environments is large and contains substantially more code than just the hypervisor due to the inclusion of one or more privileged VMs to perform tasks like I/O virtualization, peripheral access, and management [37]. Our design significantly reduces the platform TCB for the snapshot service by removing the privileged VM and its administrators from the TCB, an approach that is similar to the VMware ESXi architecture [40]. HyperShot only adds $\sim 4K$ lines to the hypervisor codebase, which coupled with the eviction of privileged VMs, nets a much smaller TCB.

Our choice of a commodity virtualization environment is motivated by the reliance of today's cloud infrastructures on full featured hypervisors, rather than on small security-specific hypervisors like SecVisor [36] and TrustVisor [20]. These small hypervisors do not support many features essential to public cloud environments, such as support of multiple VMs and VM migration. Nevertheless, our trusted snapshot service is general, and it can easily support security-specific hypervisors if cloud providers decide to move towards these smaller hypervisors in the future.

To demonstrate the usefulness of trusted snapshots, we integrated HyperShot with the open source forensic utility *Volatility* [42] and with runtime integrity measurement software called *KOP* [5]. Our security evaluation demonstrates HyperShot's effectiveness at mitigating various attacks on the snapshot service and on the snapshot itself. Our detailed performance evaluation shows that HyperShot incurs a modest runtime overhead on a variety of benchmarks.

In summary, we make the following contributions:

- We investigate the trust issues between cloud customers and providers and motivate the need to reduce the effective platform TCB in the cloud by showing a concrete attack on the snapshot originating from a malicious root VM.
- We propose trusted snapshots as a new cloud service for customers and design a mechanism for trusted snapshot generation across all VMs, including the root VM. Our design mitigates various attacks that compromise the snapshot service and integrity of the generated snapshot. We implemented our design in the Hyper-V hypervisor.
- We associate a hardware rooted trust chain with the generated snapshot by leveraging trusted computing technologies. In particular, the hypervisor *signs* the snapshot using a TPM. Trusted boot using TXT establishes trust in the hypervisor.
- We present a snapshot protocol allowing clients to request and verify VM snapshots. We demonstrate the use of trusted snapshots in various end-to-end scenarios by integrating the trusted snapshot with forensic analysis and runtime integrity measurement software.

2 Overview

2.1 Threat Model

Our threat model considers attacks that compromise the integrity of the snapshot either by tampering with the snapshot file's contents or with the snapshot service. We refer to all entities that could perpetrate this attack from the root VM—whether it is a malware instance running in the root VM or a malicious administrator—collectively as a *malicious root VM*. In particular, we consider the following types of attacks:

- **Tampering:** A malicious root VM may modify the generated snapshot file or the runtime memory and CPU state of a target VM during the snapshot process to

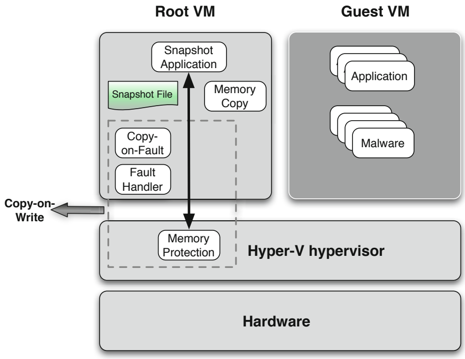


Fig. 1. The existing Hyper-V snapshot design places all components other than memory protection inside the root VM, which makes the snapshot service vulnerable to attacks by the root VM

remove memory regions that include evidence of malware or other activity undesired by the customer.

- **Reordering:** A malicious root VM may reorder the content of memory pages in the snapshot file without modifying the contents of individual pages. This may fool integrity checking or forensic analysis utilities that expect a snapshot file's contents to be in certain order, leading to failures to locate security-relevant data in the file.
- **Replaying:** A malicious root VM may provide an old snapshot of a target VM that does not contain any malicious components.
- **Masquerading:** A malicious root VM may try to intercept the snapshot request and modify the request's parameters, such as the target VM, to provide a snapshot of a VM different than the one intended.

Note that attacks meant to compromise the snapshot process via introduction of a bad hypervisor, such as bluepill [32] and SubVirt [17], are addressed by the use of trusted computing techniques, as described later. We do not consider hardware attacks, such as cold boot attacks on RAM and side channel attacks, specifically since these attacks may compromise the confidentiality of a VM's state while HyperShot's goal is to protect the integrity of the snapshot. Finally, we do not consider availability attacks, such as deleting the snapshot file, crashing the root VM, or crashing the customer VMs during the snapshot, and DMA-based attacks from rogue devices or rogue device drivers. DMA-based attacks can be thwarted using IOMMU-based protection methods, as successfully demonstrated by Nova [37] and TrustVisor [20]. We also assume *minimal trust in the cloud infrastructure* to assign a globally unique identifier (guid) to a customer VM and enforce the invariant that a VM with a particular guid can only be executing on one physical machine at any given time. This infrastructure is independent of customers, maintained by cloud providers, and it can be achieved with a much restricted core set of machines. This assumption is already required in today's cloud environments such as EC2 and Azure for proper functioning of network-based services.

2.2 Threats to Existing Hyper-V Snapshot Mechanisms

A typical virtualization infrastructure includes a hypervisor, multiple guest VMs, and a privileged management VM, such as the root VM or dom0. The current virtualization architecture supported by Hyper-V, Xen, and VMware ESX server allows the snapshot service to operate from the privileged management VM. As shown in Figure 1, the root VM in Hyper-V takes a guest VM's snapshot with only minimal support from the hypervisor. More specifically, the root VM only relies on the hypervisor to protect guest memory pages from writes performed by the target guest VM being snapshotted. These writes trigger the root VM's copy-on-write (CoW) mechanism, where the root VM handles faults (using a fault handler), copies the content of the page (using copy-on-fault) before removing the protection, and resumes the guest VM's execution. Concurrently, the snapshot application also copies other guest memory pages. After completion of the snapshot, the snapshot file is stored in the root VM and CoW protection on guest memory pages is removed.

We evaluated the security of the existing Microsoft Hyper-V [24] snapshot mechanism in a cloud environment under the threat model described above. We developed a concrete tampering attack on a customer's snapshot file by removing from it evidence of malware infection and other important information that a malicious administrator may want to hide from a customer. To launch the attack, we utilized a forensic analysis utility called *Volatility* to extract information such as the list of running processes, loaded drivers, opened files, and connections. We first opened the snapshot file in analysis mode and listed all running processes, and we then chose a process from the list to remove—in a real threat scenario, this could be malware. Next, we used Volatility to alter the list of running processes by rewriting the linked list used by Windows to store all running processes. We repeated this experiment to remove a loaded driver from the list of drivers. These malicious modifications will not be detected by the consumers of this snapshot due to the lack of any measurable trust associated with the generated snapshot.

3 HyperShot Architecture

In this section, we present the design goals of HyperShot and the detailed description of its various components as shown in Figure 2.

3.1 Design Goals

We designed HyperShot to satisfy the following goals:

- **Security:** HyperShot creates trusted snapshots by protecting both the snapshotting code and the generated snapshot data files against manipulation by malicious root VMs. To secure the snapshot service, HyperShot deploys its components inside the hypervisor. Since the hypervisor runs in a high-privileged mode, the untrusted root VM cannot alter HyperShot's components either in memory or in persistent storage. To preserve the integrity of the snapshot files stored in the root VM, HyperShot hashes the snapshot data of the target VM[1] from the hypervisor and signs the hash using the TPM. While manipulation is not directly prevented, the signed hashes

[1] We use the term target VM to refer to any VM that is being snapshotted without distinguishing whether it is a guest VM or the root VM. Any distinction is explicitly qualified.

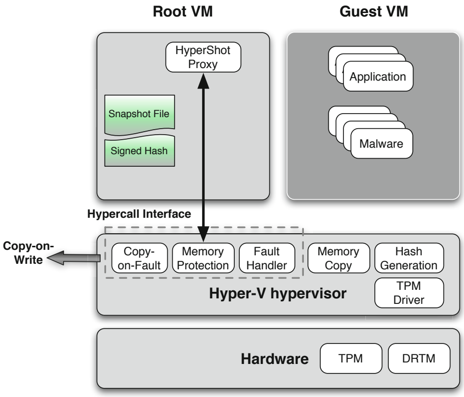


Fig. 2. HyperShot's design contains all components inside the hypervisor. It runs only a thin proxy in the root VM to forward snapshot requests to the hypervisor. The generated snapshot and its signature are stored in the root VM.

allow HyperShot or a customer to detect malicious alterations by the infrastructure provider. Unlike existing snapshot generation techniques that include the root VM in the platform TCB, HyperShot excludes the root VM and its administrators from the TCB; hence it extends the same snapshot generation functionality and trust guarantees to the root VM.

- **Consistency:** To capture a consistent state of the target VM, HyperShot does not allow any modification to the target VM state without first recording the state as it was at the time of the snapshot request.
- **Performance:** To keep the performance overhead moderate, HyperShot only pauses the target VM for a minimal duration and then uses copy-on-write (CoW) to allow the VM to continue execution during the snapshot. This design incurs low overhead on applications executing inside the target VM.

HyperShot deploys all components inside the hypervisor; it keeps only a thin proxy client in the root VM. HyperShot also protects the integrity of the snapshot from attacks that originate due to a compromise of the proxy software executing inside the root VM, as explained in Section 5.1. HyperShot moves only snapshot service related functionality into the hypervisor, leaving the rest of the code inside the root VM. The additional code added into the hypervisor totals $\sim 4K$ lines of source.

3.2 Enhanced Copy-on-Write Protection

HyperShot creates trusted and consistent snapshots of VMs executing in a virtualization based cloud environment. One possible approach to enable consistency is to pause the target VM during the entire snapshot process. This approach may impose severe runtime overhead on applications running in the target VM during the snapshot generation. Further, it still does not guarantee consistency since a malicious root VM may modify the state of the target VM while the target VM's snapshot is in progress. *To offer both consistency and security*, HyperShot utilizes an enhanced copy-on-write (CoW) mechanism inside the hypervisor.

To set up the enhanced CoW on a guest VM at the beginning of a snapshot, the hypervisor pauses the guest VM and marks its memory pages read-only by iterating across its address space. To protect the guest VM's state from untrusted modifications by the root VM during the snapshot, HyperShot also write-protects the corresponding memory pages mapped in the root VM's page tables. For snapshots of the root VM, the CoW setup is performed only on the root VM's address space because guest VMs cannot access the root VM's memory pages. Our CoW mechanism is different from those used in live VM migration [8]. In particular, we copy memory page content prior to modification rather than iteratively copying dirty pages after alteration.

3.3 Access Mediation

CoW setup allows HyperShot to mediate writes performed by the target VM to write-protected memory pages. This design fulfills HyperShot's goals of *security and consistency* since it allows HyperShot to copy the content of memory pages before they are modified. In particular, at each page fault on a write-protected page, HyperShot copies the content of the faulted page before restoring the original access permissions. These copies are stored in protected memory regions belonging to the hypervisor. In our implementation, this memory is *deposited* by the root VM on-demand before the snapshot of a VM is started. Once deposited, this memory becomes inaccessible from the root VM, hence the snapshot data stored in this memory cannot be modified by the malicious root VM. HyperShot keeps the content of the faulted and copied pages inside the hypervisor until the snapshot process is completed. *This design is required both for security and correctness* because HyperShot allows changes to occur on the faulted pages after copying the pages' contents. Multiple guest physical addresses (GPAs) may map to the same system physical address (SPA), so if a write-protected GPA faults and it mapped to an SPA whose page has already been processed, then HyperShot takes the snapshot of the faulted GPA and its hash from the stored copy rather than directly from the target VM's memory as the memory content may have been modified.

For the root VM's snapshot, faults originate only from the root VM. However, for a guest VM's snapshot, HyperShot receives faults on a write-protected page both from the guest and root VM. The faults occur from the guest VM as part of its own execution and from the root VM as part of I/O and other privileged operations. The stock Hyper-V hypervisor does not expect the root VM to generate page faults due to page protection bits because the root VM has full access to all guest VMs' memory pages. To handle these new faults, HyperShot adds a new page fault handler in the hypervisor to facilitate the copying of page contents and restoring of the original access permissions in the root VM's address space. The same page fault handler is used during the snapshot of the root VM to handle page faults due to CoW setup on the root's address space.

Finally, HyperShot provides persistent protection to the runtime environment of the target VM. HyperShot mediates operations that map and unmap memory pages in the target VM's address space maintained by the hypervisor. If these changes involve a page that is protected and not already copied, HyperShot copies the contents of the memory page before it allows the operation to complete. In addition to recording memory pages, HyperShot also snapshots the virtual CPU (vCPU) state associated with a target VM at the beginning of a snapshot and stores all vCPU register values inside the hypervisor.

3.4 Memory Copy

The memory copier is a component that resides in the hypervisor and closely works with the HyperShot proxy to move memory contents into a generated snapshot file. The proxy invokes a series of hypercalls in a loop during which it sequentially requests the contents of each memory page by invoking the memory copier and writes to the file. The copier is responsible for two tasks. First, it passes the contents of pages already snapshotted via CoW to the proxy. Second, it snapshots remaining memory pages that were not modified during the CoW, as the CoW mechanism would not have recorded the contents of unwritten pages.

On a request from the HyperShot proxy, if the requested page is write-protected and not already copied, the copier extracts the contents of the page from the target VM's memory, calculates a hash over the page's content, and stores only its hash in the hypervisor for the future use. It passes a copy of the content back to the HyperShot proxy. If the requested page had already been copied during the CoW, the copier calculates the hash on the previously stored content inside the hypervisor and sends the content back to the proxy. Finally, if a requested memory page is not mapped in the target VM, all zeroes are returned and a hash of the zero page is stored for the future use.

3.5 Hash Generation

To protect the integrity of the snapshot file from a malicious root VM, HyperShot creates message digests or hashes of the target VM's memory pages before the snapshot content is sent to the root VM. These hashes are stored inside the hypervisor, and thus are not accessible to the root VM.

The hashing process works in two steps. In the first step, HyperShot generates the SHA-1 hash of each individual memory page present in the target VM's address space. In the second step, it creates the composite hash of all the individual hashes calculated in the first step:

$$\mathtt{H_{composite}} = \mathtt{SHA\text{-}1}(\mathtt{H_1}||\mathtt{H_2}||\mathtt{H_3}......||\mathtt{H_M})$$

where M is the total number of guest memory pages and $\mathtt{H_i}$ is the SHA-1 hash of the i^{th} memory page.

To generate the composite hash, HyperShot follows a simple ordering: it always concatenates individual hashes starting from the hash of the first memory page in the target VM and continues up to the hash of the last page. This ordering is important—as described earlier in Section 2.1, an attacker may launch a reordering attack by altering the snapshot file. To detect the page reordering attempts, HyperShot always expects the snapshot to be performed in sequential order, and it generates the composite hash in the similar fashion. This design detects a page ordering attack as the composite hash will not match during verification. We describe the snapshot verification process in Section 4.2. We used a linear hash concatenation approach for simplicity, though other efficient techniques, such as Merkle hash trees [22], could be used to generate a composite.

3.6 Integrity Protection of the Snapshot

HyperShot keeps the contents of CoW memory pages and the hashes of all pages in the hypervisor memory until the snapshot is over; the individual hashes are used to generate the composite. Merely generating $\mathtt{H}_{\mathtt{composite}}$ inside the hypervisor is insufficient to maintain the integrity of the snapshot because malware or malicious administrators can easily subvert the hash when it is transferred to the root VM. We must protect the integrity of $\mathtt{H}_{\mathtt{composite}}$ itself so that it cannot be modified inside the root VM. We use hardware-supported *signing* in order to detect malicious tampering of $\mathtt{H}_{\mathtt{composite}}$ and leverage trusted computing technologies to establish a trusted environment for the signing operation.

Trusted computing technologies provide a basis for trusting a platform's software and hardware configurations based on a *trust chain that is rooted in hardware*. In particular, it provides the *Trusted Platform Module* (TPM) [38], which is available today on most commodity hardware platforms. Each TPM has multiple 20-byte *Platform Configuration Registers* (PCRs) used to record measurements related to the platform state, such as those of various software components on the platform. These measurements form a chain of trust, formed either *statically* at the platform's boot time, or *dynamically* at any time during execution. Most PCRs cannot be reset by software but only *extended* from their previous values. The TPM also contains an asymmetric *endorsement key* (EK) that uniquely identifies the TPM and the physical host it is on. Due to security and privacy concerns, a separate asymmetric *attestation identity key* (AIK) is used as an alias for EK when signing *remote attestations* of the platform state (as recorded in various PCRs). The process of establishing an AIK based on EK is described in detail in [27]. The remote attestation process is based on the TPM's *quote* functionality, which signs a 20-byte sized data value using $\mathtt{AIK}_{\mathtt{priv}}$ and a set of PCRs [38]. A more detailed description of trusted computing technologies for today's commodity platforms is provided by Parno et al. [28].

HyperShot leverages a research prototype version of Hyper-V that provides a trusted boot of the hypervisor. In particular, the hypervisor is launched via *Intel's TXT* [15] technology, and it is started *before* the root VM (compared to the stock Hyper-V architecture where the hypervisor starts after the root VM). This late hypervisor launch architecture is very similar to Flicker [21] and trusted Xen [7]. The trusted boot measures the platform TCB and records the measurements in a non-repudiable fashion in the TPM's PCRs 17, 18, and 22. The AIK is also loaded into the TPM before a signature can be requested from the TPM in the form of a quote.

After the snapshot process is completed, the hypervisor records $\mathtt{H}_{\mathtt{composite}}$ as part of the system configuration in the resettable PCR 23. The quote request (described in detail in Section 4) includes PCRs 17, 18, 22, and 23 in the signing process to ensure (a) the integrity of the platform TCB via PCRs 17, 18, and 22; and (b) the integrity of the generated snapshot itself via PCR 23. Note that in HyperShot, the TPM is under the control of hypervisor and only a para-virtualized interface is provided to the root VM to request a quote. A malicious root VM cannot send arbitrary raw TPM commands to corrupt the TPM state. Also, the para-virtualized TPM interface presented to the root VM does not allow resetting of PCR 23; only hypervisor is allowed to reset PCR 23. This design protects the integrity of the measurement from the malicious root VM.

3.7 DMA Considerations

A DMA operation occurring during a snapshot may alter memory pages of a target VM. To ensure snapshot consistency, we modify the guest DMA processing code inside the root VM at a point prior to the actual DMA. We add a hypercall in this code path to pass the GPAs of memory pages being locked for the DMA. This ensures that the original memory page is processed prior to the DMA if it had not already been copied for the snapshot. We intercept guest DMA operations for consistency, and not for security reasons. Our DMA interception code resides inside the root VM and is prone to attacks. Our current implementation does not yet support interception of the root VM's DMA operations. Though it is possible by modifying the root's DMA handler, it would also not be secure. A future integration of IOMMU with CoW based protection will ensure that any DMA write operation that bypasses the snapshot will fault and be either aborted or restarted after the copy is made.

4 HyperShot Protocol

The snapshot protocol allows clients or challenging parties to securely retrieve trusted snapshot and validation information from the providers or attesting system. HyperShot generates snapshots of VMs at the request of a client. The protocol used by HyperShot involves the following entities:

- **Challenger:** A customer or challenger is an entity who wishes to measure the integrity of her VM running in the cloud. In HyperShot, a cloud infrastructure provider can act as the challenger when it requests snapshots of the root VM of a particular machine. Challengers can act as verifiers or can assign verification duty to third parties whom they trust.
- **HyperShot Service Front-End:** The front-end routes the snapshot request to the physical machine where the VM to be snapshotted is currently executing. It also sends the snapshot file and TPM quote along with the `AIK` certificate back to the customer. The HyperShot service front-end is a part of the minimal trusted cloud infrastructure.
- **Forwarder:** The HyperShot proxy receives the request from the HyperShot service front-end and uses the hypercall interface to pass the request to the hypervisor.
- **Generator/Attestant:** The generator is the HyperShot component inside the hypervisor that performs most of the work—it sets up proper CoW protections, copies memory pages, protects the integrity of individual memory pages and the composite snapshot, and provides an attestation using the TPM. This is the key component of the HyperShot TCB.

4.1 Snapshot Generation Protocol

Figure 3 depicts the HyperShot protocol with steps involved in the snapshot generation process. The protocol starts when a challenger sends a snapshot request to the HyperShot service front-end in the cloud, identifying the VM to be snapshotted by the guid

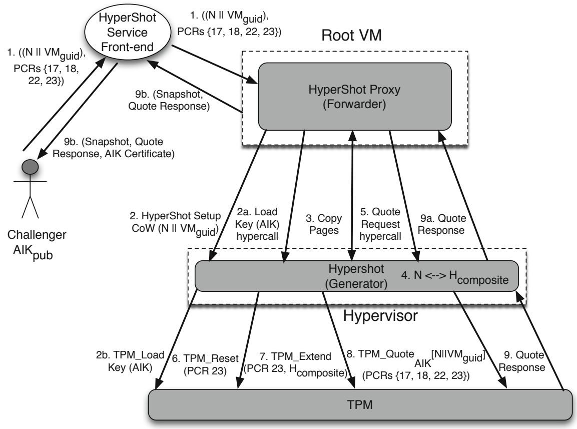


Fig. 3. HyperShot protocol to request VM snapshots

(assigned by the trusted part of the cloud infrastructure at the VM creation time). In particular, the challenger first creates a non-predictable random nonce $\mathtt{N}$, concatenates it with the $\mathtt{VM_{guid}}$, and sends it to the HyperShot service (1). The nonce guarantees the freshness of the request and *thwarts replay attacks*, whereas the VM identifier *defeats masquerading attacks*.

On receiving a nonce and $\mathtt{VM_{guid}}$, the service front-end finds the physical host on which the $\mathtt{VM_{guid}}$ is running and forwards the request to the HyperShot proxy running on that host. The proxy further forwards the request to the hypervisor and starts the CoW setup process using the hypercall (2). Masquerading attempts in which a malicious root VM modifies the $\mathtt{VM_{guid}}$ that is being passed to the hypervisor can easily be detected by the customer as the $\mathtt{VM_{guid}}$ value will be used to create the final quote by the TPM. Defenses against other forms of masquerading attacks such as "evil clone" are described in Section 5.1.

After initiating the CoW, the forwarder also requests the hypervisor to load the protected key blob for AIK into the TPM, which is used later by the TPM to sign the snapshot (2a-2b). After the CoW setup, HyperShot starts copying the target VM's memory pages either through CoW faults or by servicing memory copy requests from the forwarder (3). Once the memory copying process is over, the hypervisor generates the composite hash $\mathtt{H_{composite}}$ and associates it with the nonce (4), and stores it for future use.

Next, the forwarder requests a quote over the nonce and the VM identifier from the hypervisor (5). The hypervisor resets PCR 23 (6), and then *extends* it with $\mathtt{H_{composite}}$ corresponding to the nonce and the VM identifier (7):

$$\mathtt{PCR}_{23} = \mathtt{Extend}(0||\mathtt{H_{composite}})$$

The hypervisor then generates the following *TPM_Quote* request (8):

$$\mathtt{TPM\_Quote}_{\mathtt{AIK}}(\mathtt{N}||\mathtt{VM_{guid}})[\mathtt{PCRs}]$$

where $\mathtt{AIK}$ is the handle for an $\mathtt{AIK_{priv}}$ already loaded in the TPM, and $\mathtt{PCRs}$ is the set of $\mathtt{PCRs} = \{17, 18, 22, 23\}$ to be included in the quote. As described earlier, PCRs 17, 18, and 22 report the integrity of the platform TCB, while PCR 23 reports the integrity measure for the snapshot.

The generated quote is sent back to the forwarder (9, 9a), which sends it to the HyperShot service front-end, which in turn sends it to the challenger along with the snapshot file and AIK certificate (9b). The snapshot verification process performed at the verifier/challenger site is explained in the next section.

4.2 Snapshot Verification Protocol

The verification of the snapshot is straightforward, and it is performed completely in software. In the first phase, the verifier ensures that the quote obtained from the forwarder is valid. This requires completing the following steps:

- The verifier checks that $\mathtt{AIK}_{\mathtt{pub}}$ is indeed a valid AIK based on a certificate associated with the AIK obtained. Next, it verifies the signature from the quote process.
- It determines if the values of PCRs 17, 18, and 22 included in the quote correspond to a known good hypervisor. The known good values of a hypervisor are known to the verifier a priori. This informs the challenger that the hypervisor's integrity was maintained during the snapshot.
- It extracts $\mathtt{H}_{\mathtt{sent}}$ as the value of PCR 23 included in the quote.

In the next phase, the verifier computes the composite hash, $\mathtt{H}_{\mathtt{local}}$, over the memory contents contained in the snapshot file by using the same algorithm as used by HyperShot (described in Section 3.5). Next, the verifier performs the extend operation (in software):

$$\mathtt{H}_{\mathtt{final}} = \mathtt{Extend}(0||\mathtt{H}_{\mathtt{local}})$$

If $\mathtt{H}_{\mathtt{final}} = \mathtt{H}_{\mathtt{sent}}$, then the snapshot received by the verifier is trusted. Otherwise, the verifier discards the snapshot and takes remedial action, such as informing the provider or moving its work to an alternate provider.

5 Evaluation

HyperShot's functionality extends the platform hypervisor with $\sim 4K$ lines of C code, a modest increase compared to the original size of the hypervisor. This design results in a large TCB reduction from the point of view of the snapshot service as HyperShot does not rely on the root VM and its administrators [37]. Next, we provide a detailed security and performance evaluation of HyperShot and its deployment strategy.

5.1 Security Analysis and Evaluation

We analyze the security of HyperShot against threats described in Section 2.1, such as tampering, reordering, replaying, and masquerading. We first considered the tampering attack and used the same attack as described in Section 2.2 to compromise the integrity of the snapshot file. After the completion of the snapshot process but before the snapshot is sent to the client, we used Volatility [42] to alter the generated snapshot file stored in the root VM. We deleted a process from the list of running processes. The modified snapshot file and the TPM quote were sent to the verifier, which followed the protocol described in Section 4.2. The verification failed since the hash calculated by the verifier did not match the hash included in the trusted quote. Note that this attack was successful on a system with stock Hyper-V snapshotting in the root VM.

Next, we analyzed an attacker's ability to manipulate the snapshot via manipulation of the proxy. An attacker may launch reordering attacks either by reshuffling the content of the snapshot file or by compromising the code and data of the HyperShot proxy. HyperShot mitigates page reordering attacks by following a simple ordering when generating the composite hash. This design forces the proxy to copy memory pages in the snapshot file in the same order. If this ordering is changed by malware instances or malicious administrators, then the generated composite hash in the hypervisor will not match with the hash calculated at the verifier. HyperShot also prevents other attacks originating from the proxy software inside the root VM. An untrusted root VM may force the proxy to skip the snapshot of some memory pages that they want to hide. To defeat these attacks, the hypervisor ensures that the set of pages protected at the beginning of the CoW setup is the same as the set of pages hashed, and if not, it does not generate the TPM quote over the composite hash.

HyperShot thwarts replaying and masquerading attacks by using a nonce and VM identifiers in the request from the clients. The nonce allows HyperShot to distinguish between a new and the old request, while a VM identifier allows it to identify the target VM. Since the TPM quotes the generated snapshot using the nonce and the VM identifier that it receives in the request, the verifier can check whether the snapshot corresponds to a recent request and for the correct VM.

A different version of the masquerading attack may be launched by the root VM by setting up an "evil" clone of the customer VM with malware in it. The root VM may start this clone along with the customer VM (with a different guid, since the hypervisor enforces the uniqueness of VM guids). Next, the root VM may divert all snapshot requests to the correct VM and actual service requests that operate on customer private data to the evil clone. A variant of the same attack has the root VM shutting down the correct customer VM after a snapshot, starting the evil clone with the customer VM's guid, and forwarding service requests to the clone. These attacks can be mitigated by using a communication protocol enhanced with attestation, such as a quote based on the VM's recent snapshot, in a manner similar to the one proposed by Goldman et al. [13]. They addressed the lack of linkage between the endpoint identity determination and remote platform attestation, which is precisely the root of the evil clone problem. In short, the solution requires generating an SSL certificate for the VM and incorporating the hash of this certificate in the TPM quote, along with the VM's snapshot hash. Both of these values can be captured in vPCRs by our hypervisor.

Our security analysis demonstrates that HyperShot's design effectively mitigates attacks in a cloud environment. Although we have not explored any responses to an untrusted snapshot other than to discard it, it is plausible that in an operational cloud environment, customers would escalate this security threat with the infrastructure provider.

We also evaluated the usefulness of a trusted snapshot by integrating the snapshot generated by HyperShot with two runtime integrity management tools, Volatility [42] and KOP (Kernel Object Pinpointer) [5]. Volatility is an open source forensic tool that analyzes OS memory images. It recognized the snapshot generated by HyperShot without any modifications to its image parsing logic. We extracted information such as a list of running processes, drivers, files, dlls, and sockets from a snapshot of a Windows XP guest VM. With this information, customers could identify potentially malicious software present in their VMs at the time of snapshot, *without the possibility of any tampering from the cloud infrastructure provider*.

KOP uses offline static analysis of a kernel's source code to enable systematic kernel integrity checking by mapping kernel data objects in a memory snapshot. In its current implementation, KOP supports Windows Vista and requires the memory image to be in the *crash dump format*. In our experiment, we ran a kernel malware instance that hides its presence by deleting itself from the list of loaded drivers inside a Vista guest VM. We used an offline utility to convert a HyperShot snapshot into the crash dump format, and we then ran KOP on the dump. KOP successfully found the hidden malicious driver object in the kernel memory. Our integration demonstrates the usefulness of HyperShot as an integral component for managing end-to-end runtime integrity management of VMs in a public cloud environment.

5.2 Performance Evaluation

In this section, we present micro- and macro-benchmark results to quantify (1) the performance impact of HyperShot's trusted snapshot generation on target VMs, (2) the demand put on system resources, specifically memory, and (3) the performance of end-to-end snapshot generation and verification. All experiments are performed on an Intel Quad Core 2.53 GHz machine with 4 GB of memory and Extended Page Table (EPT) support [16]. EPT allows HyperShot to modify the GPA-to-SPA map using the second level page table kept in the hypervisor, while the OS inside the VM manages the traditional first level page tables to map virtual address to GPA. For simplicity, we ran the challenger application and the forwarder inside the root VM, which runs 64-bit Windows Server 2008 R2, and obviated the service front-end. The guest VM was allocated 1GB RAM, 1 CPU, and ran 32-bit Windows XP with service pack 3. For macro-benchmark experiments, we used the Passmark Performance Test benchmark suite [29] inside the guest VM.

HyperShot uses CoW to protect memory pages that generate faults whenever the guest or root writes on these protected pages. On each CoW fault, besides changing the page protection, HyperShot also copies the contents of the page if needed. *The median time to perform the additional copy operation is 6.45μs.*

Next, we measured HyperShot's overhead on different workloads using the PassMark benchmark suite running inside a guest VM. We used 5 representative benchmarks to measure HyperShot's impact on CPU, memory read and write, and sequential disk read and write performance respectively. Each benchmark was executed for 30 times in a loop, and we snapshotted the VM during this loop's execution. Each iteration of the loop ran for a short period of time, so that multiple iterations (but not all) were impacted by the snapshot process. The overall loop's execution encompassed the whole snapshot process. Mean results in Figure 4 indicate that HyperShot imposes no

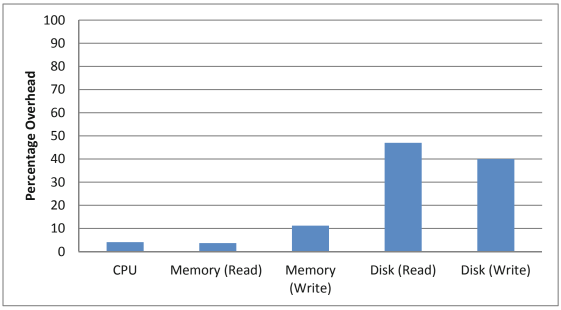


Fig. 4. Percentage overhead incurred by HyperShot on various workloads. Smaller measurements are better

discernible overhead on CPU and memory read benchmarks, and a modest $\sim$12% overhead on memory writes. This is expected, since CPU and memory read benchmarks do not generate many CoW faults, while the memory write benchmark generates many CoW faults when it writes data to the memory pages. Higher overheads for disk benchmarks are due to a pro-active, per-page hypercall made by the root VM to copy a page before any DMA operation in anticipation of a CoW fault. This hypercall is made even if the page may already have been copied due to an earlier CoW fault or even if the page will not generate a CoW fault due to a DMA write request (which reads the memory page and writes to the device). This hypercall is serialized with the HyperShot proxy, thereby effectively reducing the parallelism in the system.

We measured HyperShot's CoW memory storage requirements for different workloads during the snapshot of a guest VM. These workloads have different runtime memory requirements and access patterns, and hence provide a good indicator of typical memory usage by HyperShot. We calculate percentage memory consumption as

$$\frac{\text{number of pages copied due to a CoW fault} \times 100\%}{\text{total number of memory pages protected}}$$

The denominator represents an upper bound on the number of pages that will need to be copied in the worst case. The total number of protected pages for a guest VM with 1 GB memory is 262,107.

The results shown in Table 1 are median values for 5 runs of each workload, with the overall as a median of all 30 runs. These results indicate that the memory requirement for HyperShot is $<1.5\%$ for all the workloads considered here, with memory write and disk read benchmarks on the higher side since they generate more CoW faults. This indicates that an alternative, more efficient, strategy for HyperShot would be to deposit only a fraction of the total target VM memory size from the root VM to successfully finish the snapshot, rather then its current strategy of depositing the amount equal to target VM's memory size. If more memory is needed during a guest VM's snapshot, the hypervisor can pause the guest VM (so it does not generate any further faults) and request more memory to be deposited from the root VM. Note that it may not work for all cases, e.g. if the fault is generated due to an access from the root VM, or if the root

Table 1. Memory overhead due to CoW fault handling for different workloads in a guest VM

Operations	*Number of CoW Fault*	*% Memory* Consumption
Idle VM	1622	0.62
CPU workload	3588	1.37
Memory read	2793	1.07
Memory write	3688	1.41
Disk write	2955	1.13
Disk read	3032	1.16
Overall	3407	**1.30**

Table 2. Time to take SHA-1 hash of a single page of size 4KB and the composite hash on the VM of size 1024 MB

Operations	*Time*
Single memory page hash	149.0 μs
Composite hash	186.5 ms

VM itself is being snapshotted. In this case, HyperShot will have to abort and restart the snapshot process.

In our next set of experiments, we quantify the performance of the snapshot generation and verification. First, we micro-benchmarked the time to perform hashing operations. As described earlier, HyperShot performs two different hashes: (a) hashing individual memory pages at the time of memory reads by the HyperShot proxy, and (b) a composite hash at the end of the snapshot process. The results shown in Table 2 indicate that hashing operations are fast, and their impact on overall snapshot generation is small.

Next, we measured the time to finish various stages of the snapshot process from the HyperShot proxy for a guest VM and for an idle root VM. The guest VM was either idle or was executing one of the 5 benchmarks from the PassMark suite described earlier. We measured CoW setup time (initialization time), VM memory copy time, cleanup time, and TPM quote time. Due to brevity, we have skipped the details for each workload type. Not surprisingly, we found the overall time to be dependent more on the number of memory pages being snapshotted and less on the workload itself. *For guest VMs of memory size 1 GB, the overall time to finish the snapshot was 39s, of which 37s were spent in making the memory copy from the hypervisor one page at a time and writing it to the snapshot file, 1s in obtaining the TPM quote, and 1s in initialization and cleanup.* Overall time for the root VM's snapshot was 391s, of which 218s are spent in memory copy, 3s in initialization, 169s in cleanup, and 1s in TPM quote. The increased cost for the root VM is due to both large memory size and a larger number of CoW faults generated by the root VM during its normal execution. Further performance improvements can be made by sharing memory page information between the HyperShot proxy and the hypervisor so that it can avoid hypercalls for pages that are deposited for CoW processing and are not part of snapshot, or have been copied already in DMA setup path.

For the challenger, generating the hash for a snapshot of 1GB memory took 22s, while the TPM quote validation in software and matching the hash to the one reported in the quote took negligible time. We assume that the snapshot file and the TPM quote would be transferred out-of-band to the challenger, and the cost of this step would depend on the network bandwidth between challenger and the cloud. As mentioned in Section 4.1, the challenger could be the end-user of the cloud or any third party who has been appointed on the behalf of the customer to verify the integrity of the snapshot.

Although further performance optimizations are possible, results from the prototype implementation show that HyperShot's design is viable, and it imposes moderate overhead on the snapshotted VMs and on the overall platform.

5.3 Deployment Strategy

We envision that cloud providers would add the interface for trusted snapshot requests to their web-based VM management front-end. This front-end communicates with the management plane of the cloud infrastructure that controls hundreds of thousands of machines, each running multiple VMs hosting customers code, data, and services. We assume that each physical machine has a TPM chip, which is available today on most commodity hardware platforms. Cloud providers set up each physical machine as it is provisioned to the data center with software and credentials in order to participate as part of the hosting infrastructure. As part of this provisioning, the system software on the machine would generate the AIK using the TPM and would obtain an AIK certificate from a trusted certification authority (CA). This CA is either part of the trusted infrastructure of the cloud provider or an external party, and the certificate ensures that AIK_{priv} is protected by a physical TPM. This certificate is then shared with the HyperShot service front-end. On a request from a customer to generate a trusted snapshot, the trusted infrastructure maps the globally unique identifier (guid) of the VM mentioned in the request to a physical host running the customer's VM. After generating the snapshot, the final quote, the AIK certificate associated with the physical machine, and the snapshot are returned to the customer. The customer can use the certificate to validate the AIK which is then used to verify the integrity of the snapshot. Since the contents of a snapshot are integrity protected, the cloud provider is free to store the snapshot as it deems fit—on a local disk, a network share, or using a cloud storage service such as S3 or EBS.

5.4 Discussion

HyperShot is vulnerable to a scrubbing attack where a malicious administrator or compromised root VM can scrub attack traces from the customer VM before the snapshot process starts. A possible solution to this problem is to keep the snapshot request hidden from the malware until the CoW initialization is over. This requires an out-of-band direct communication channel between the service front-end and the hypervisor on the physical machine. It may be possible to establish such a channel over the secondary management communication infrastructure using special purpose processors, such as Intel AMT. The viability of this type of solution has been demonstrated by recent integrity measurement work [1]. An alternative solution makes the snapshot generation an asynchronous process, with the hypervisor initiating the CoW protection at a random point in time. HyperShot only records a VMs' memory and registers; it does not yet snapshot disk contents. We plan to extend it with an existing virtual disk snapshot solution, such as Parallax [23] or the disk snapshotter available with Hyper-V.

6 Related Work

Trust issues in cloud computing are an active area of research investigations. Recent work has showed how to use untrusted cloud infrastructures to store and relay information [10, 19]. Trusted computing and virtualization can further enable more general purpose services in a public cloud environment. Santos et al. [34] presented a TPM-based architecture to protect the confidentiality and integrity of data in the cloud. Krautheim [18] proposed a new management and security model using TPMs called private virtual infrastructure (PVI) for cloud computing. TrustVisor [20] provided safety of computation using a TPM based design. Sailer et al. [33] designed and implemented a trusted computing based integrity measurement architecture for desktops. Schiffman et al. [35] proposed a centralized verification service that produces TPM-based attestation of customers' cloud instances for the integrity and transparency purposes. In a similar manner, HyperShot uses trusted computing technologies such as TPM and secure boot to enable a trusted snapshot capability in a cloud infrastructure, where only minimal trust is placed in infrastructure and the hypervisor.

Past research has proposed various mechanisms to reduce the size of trusted computing bases in virtualized environments. Murray et al. [26] proposed a disaggregation based approach for Xen that split dom0 into multiple small privileged VMs, with emphasis on securing interfaces among them. Nova [37] proposed a micro-kernel inspired design for a secure virtualization architecture. It minimized the amount of code in the privileged hypervisor and moved more functionality into service VMs, thereby reducing the core platform TCB. Terra [11] offered security to customer VMs by developing a small trusted virtual machine monitor and provided an isolated closed-box environment to execute sensitive applications. All of these approaches can be readily leveraged by HyperShot by moving the snapshot functionality to its own *service VM*. However, in order to provide similar trust guarantees as our hypervisor-based approach, the TCB for this snapshot service VM must be carefully managed to limit it to just the hypervisor.

Many commercial virtualization platforms provide a VM snapshot facility based on a CoW mechanism [9, 24]. However, these solutions depend upon the correct operation of the root VM, and they rely on potentially malicious administrators to assist in the snapshot process [30]. In contrast to these systems, HyperShot does not rely on the privileged VM and administrators and protects the integrity of a VM's snapshot from unwanted modification from these entities. HyperShot even supports snapshotting of the privileged VM itself using the same mechanisms, a feature missing from current virtualization solutions. These properties makes HyperShot more suitable for a virtualization based public cloud infrastructure.

7 Conclusions

We investigated trust issues between cloud customers and providers. To allow customers to establish trust in the public cloud infrastructures, we designed and developed *HyperShot*, a system that securely snapshots a VM even in the presence of a malicious root VM or malicious administrators. HyperShot employs TPM-based *attestation* and

TXT-based *trusted launch of the hypervisor* to protect the integrity of the snapshot via a signature and trusted reporting of the platform state. HyperShot extends the same snapshot functionality and trust guarantees to the root VM itself. This enables cloud infrastructure providers and customers alike to manage the security and integrity of their VMs based on trusted snapshots. We integrated HyperShot with forensic and runtime integrity measurement utilities. Our performance evaluations showed that HyperShot incurs moderate overhead on the VMs' performance.

References

1. Azab, A.M., Ning, P., Wang, Z., Jiang, X., Zhang, X., Skalsky, N.C.: Hypersentry: Enabling stealthy in-context measurement of hypervisor integrity. In: ACM CCS, Chicago (October 2010)
2. Balding, C.: What everyone ought to know about cloud security, http://www.slideshare.net/craigbalding/what-everyone-ought-to-know-about-cloud-security (last accessed April 08, 2012)
3. Baliga, A., Ganapathy, V., Iftode, L.: Automatic inference and enforcement of kernel data structures invariants. In: Proc. of ACSAC, Anaheim, CA (December 2008)
4. Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I., Warfield, A.: Xen and the art of virtualization. In: ACM SOSP, NY (October 2003)
5. Carbone, M., Cui, W., Lu, L., Lee, W., Peinado, M., Jiang, X.: Mapping Kernel Objects to Enable Systematic Integrity Checking. In: ACM CCS, Chicago, IL (November 2009)
6. Christodorescu, M., Sailer, R., Schales, D., Sgandurra, D., Zamboni, D.: Cloud Security Is Not (Just) Virtualization Security. In: Proc. of CCSW, Chicago, IL (November 2009)
7. Cihula, J.: Trusted Boot: Verifying the Xen Launch, http://xen.org/files/xensummit_fall07/23_JosephCihula.pdf (last accessed April 08, 2012)
8. Clark, C., Fraser, K., Hand, S., Hansen, J.G., Jul, E., Limpach, C., Pratt, I., Warfield, A.: Live Migration of Virtual Machines. In: Proc. of USENIX NSDI, Boston, MA (May 2005)
9. Colp, P., Matthews, C., Aiello, B., Warfield, A.: VM Snapshots, http://www.xen.org/files/xensummit_oracle09/VMSnapshots.pdf (last accessed April 08, 2012)
10. Feldman, A.J., Zeller, W.P., Freedman, M.J., Felten, E.W.: SPORC: Group Collaboration using Untrusted Cloud Resources. In: Proc. of OSDI, Vancouver, Canada (October 2010)
11. Garfinkel, T., Pfaff, B., Chow, J., Rosenblum, M., Boneh, D.: Terra: A virtual machine-based platform for trusted computing. In: Proc. of ACM SOSP, NY (October 2003)
12. Garfinkel, T., Rosenblum, M.: A Virtual Machine Introspection Based Architecture for Intrusion Detection. In: Proc. of NDSS, San Diego, CA (February 2003)
13. Goldman, K.A., Perez, R., Sailer, R.: Linking remote attestation to secure tunnel endpoints. In: ACM STC, Alexandria, VA (October 2006)
14. Haeberlen, A.: A case for the accountable cloud. In: Proc. of LADIS, Big Sky, MT (October 2009)
15. Intel. Intel Trusted Execution Technology, http://www.intel.com/technology/security/ (last accessed April 08, 2012)

16. Intel. Intel Virtualization Technology: Hardware support for efficient processor virtualization, http://download.intel.com/technology/itj/2006/v10i3/v10-i3-art01.pdf (last accessed April 08, 2012)
17. King, S.T., Chen, P.M., Wang, Y.-M., Verbowski, C., Wang, H.J., Lorch, J.R.: SubVirt: Implementing Malware with Virtual Machines. In: IEEE Symposium on Security & Privacy, Oakland, CA (May 2006)
18. Krautheim, F.J.: Private Virtual Infrastructure for Cloud Computing. In: Proc. of HotCloud, San Diego, CA (June 2009)
19. Mahajan, P., Setty, S., Lee, S., Clement, A., Alvisi, L., Dahlin, M., Walfish, M.: Depot: Cloud Storage with Minimal Trust. In: Proc. of OSDI, Vancouver, Canada (October 2010)
20. McCune, J.M., Li, Y., Qu, N., Zhou, Z., Datta, A., Gligor, V., Perrig, A.: TrustVisor: Efficient TCB reduction and attestation. In: IEEE Symposium on Security & Privacy, CA (May 2010)
21. McCune, J.M., Parno, B., Perrig, A., Reiter, M.K., Isozaki, H.: Flicker: An Execution Infrastructure for TCB Minimization. In: Proc. of ACM EuroSys, Glasgow, UK (March 2008)
22. Merkle, R.C.: A Digital Signature Based on a Conventional Encryption Function. In: Pomerance, C. (ed.) CRYPTO 1987. LNCS, vol. 293, pp. 369–378. Springer, Heidelberg (1988)
23. Meyer, D.T., Aggarwal, G., Cully, B., Lefebvre, G., Feeley, M.J., Hutchinson, N.C., Warfield, A.: Parallax: Virtual Disks for Virtual Machines. In: Proc. of ACM Eurosys, Scotland (March 2008)
24. Microsoft. Hyper-V Architecture, http://msdn.microsoft.com/en-us/library/cc768520BTS.10.aspx (last accessed April 08, 2012)
25. Molnar, D., Schechter, S.: Self Hosting vs. Cloud Hosting: Accounting for the security impact of hosting in the cloud. In: Proc. of WEIS, Boston, MA (June 2010)
26. Murray, D.G., Milos, G., Hand, S.: Improving Xen security through disaggregation. In: Proc. of ACM VEE, Seattle, WA (March 2008)
27. Open TC. OpenTC PKI: AIK Certificate Creation Cycle, http://opentc.iaik.tugraz.at/index.php?item=pca/pca_aik_create (last accessed April 08, 2012)
28. Parno, B., McCune, J., Perrig, A.: Bootstrapping Trust in Commodity Computers. In: Proc. of IEEE Symposium on Security & Privacy, Oakland, CA (May 2010)
29. Passmark Software. PassMark Performance Test, http://www.passmark.com/products/pt.htm (last accessed April 08, 2012)
30. Potter, S., Bellovin, S.M., Nieh, J.: Two-person control administration: Preventing administation faults through duplication. In: Proc. of LISA, Baltimore, MD (November 2009)
31. Ristenpart, T., Tromer, E., Shacham, H., Savage, S.: Hey, you, get off of my cloud! Exploring information leakage in third-party compute clouds. In: ACM CCS, Chicago (November 2009)
32. Rutkowska, J.: Subverting Vista kernel for fun and profit. In: Black Hat USA (2006)
33. Sailer, R., Zhang, X., Jaeger, T., van Doorn, L.: Design and implementation of a TCG-based integrity measurement architecture. In: Usenix Security, San Diego, CA (August 2004)
34. Santos, N., Gummadi, K.P., Rodrigues, R.: Towards trusted cloud computing. In: HotCloud, San Diego, CA (June 2009)
35. Schiffman, J., Moyer, T., Vijayakumar, H., Jaeger, T., McDaniel, P.: Seeding clouds with trust anchors. In: Proc. of CCSW, Chicago, IL, (Novemer 2010)
36. Seshadri, A., Luk, M., Qu, N., Perrig, A.: SecVisor: A Tiny Hypervisor to Provide Lifetime Kernel Code Integrity for Commodity OSes. In: Proc. of ACM SOSP, WA (October 2007)
37. Steinberg, U., Kauer, B.: NOVA: A Microhypervisor-Based Secure Virtualization Architecture. In: Proc. of ACM Eurosys, Paris, France (April 2010)

38. Trusted Computing Group. TPM Specification version 1.2, Parts 1, 2, & 3, http://www.trustedcomputing.org (last accessed April 08, 2012)
39. VMware. Debugging Virtual Machines with the Checkpoint to Core Tool, http://www.vmware.com/pdf/snapshot2core_technote.pdf (last accessed April 08, 2012)
40. VMware. The Architecture of VMware ESXi, http://www.vmware.com/files/pdf/vmware_esxi_architecture_wp.pdf (last accessed April 08, 2012)
41. VMware. Virtualization Software, http://www.vmware.com (last accessed April 08, 2012)
42. Volatility. The Volatility framework: Volatile memory artifact extraction utility framework, https://www.volatilesystems.com/default/volatility (last accessed April 08, 2012)

Secure and Robust Monitoring of Virtual Machines through Guest-Assisted Introspection

Martim Carbone[1], Matthew Conover[2], Bruce Montague[2], and Wenke Lee[1]

[1] College of Computing, Georgia Institute of Technology, Atlanta, GA 30332, USA
[2] Symantec Research Labs, Mountain View, CA 94043, USA
{mcarbone,wenke}@cc.gatech.edu,
{matthew_conover,bruce_montague}@symantec.com

Abstract. Current monitoring solutions for virtual machines do not incorporate both security and robustness. Out-of-guest applications achieve security by using virtual machine introspection and not relying on in-guest components, but do not achieve robustness due to the semantic gap. In-guest applications achieve robustness by utilizing guest OS code for monitoring, but not security, since an attacker can tamper with this code and the application itself. In this paper we propose SYRINGE, a *secure* and *robust* infrastructure for monitoring virtual machines. SYRINGE protects the monitoring application by placing it in a separate virtual machine (as with the out-of-guest approach) but at the same time allowing it to invoke guest functions (as with the in-guest approach), using a technique known as function-call injection. SYRINGE verifies the secure execution of the invoked guest OS code by using another technique, localized shepherding. The combination of these two techniques allows SYRINGE to incorporate the best of out-of-guest monitoring with that of in-guest monitoring. We implemented a prototype of SYRINGE as a Linux application to monitor a guest running Windows XP and have evaluated its performance and security. We also implemented a monitoring application built on top of SYRINGE to demonstrate its usefulness. Our results show that for a calling period of 1 second, the performance overhead created in the guest by this application is 8%.

Keywords: Virtualization, Introspection, Semantic Gap, Security Monitoring.

1 Introduction

The increasing popularity of whole-system virtualization, fueled by the rapid growth of industry trends such as cloud computing, creates the need for *robust* and *secure* infrastructures for monitoring virtual machines (VMs). By *robustness* we mean the ability of the monitoring infrastructure to accommodate variations in a guest VM system's characteristics (e.g., syntax and semantics of data structures) across different software releases. By *security* we mean protection against attacks targeting the monitoring infrastructure. Satisfying both of these requirements has proven itself a significant challenge. One common type of VM monitoring, on which this work is focused, is the passive monitoring of the guest OS's internal state. There are two main approaches for implementing this type of monitoring: out-of-guest and in-guest.

D. Balzarotti, S. Stolfo, and M. Cova (Eds.): RAID 2012, LNCS 7462, pp. 22–41, 2012.

The out-of-guest approach achieves good security by placing the monitoring application in an isolated, **security VM** (SVM), from where it can securely monitor the **guest VM** (GVM) using a technique known as virtual machine introspection (VMI) [1–5]. It does not usually rely on internal guest components to perform its monitoring, as these components can be maliciously tampered with. VMI lacks robustness, however, due to the semantic gap inherent to low-level monitoring across different VMs. Any changes made to the syntax (i.e., internal disposition and location of fields) or semantics (i.e., the meaning of the data stored in each field) of monitored GVM data structures across different software releases can break introspection-based tools, which rely on pre-determined and, in many cases, reverse engineered knowledge. This is especially true for undocumented data structures, which are extremely common in closed-source operating systems and applications.

The in-guest approach achieves better robustness by placing the monitoring application inside the guest. So instead of directly parsing data structures in memory, in this approach the monitoring application calls functions provided by the guest OS API to get the information it needs (e.g., the list of active processes on the system) [6]. This method naturally accommodates changes made to data structure syntax and semantics across releases, as it uses the guest's own code, which is changed accordingly by the software vendor and has a public, documented API. This approach lacks the security of the out-of-guest approach, however, since the application and the guest OS can be easily tampered with by malware to report fake monitoring results, or be simply disabled. To fully protect an in-guest monitoring application is a very hard problem, and has only been shown for small agents operating under limiting constraints [7, 8] or without ensuring the applications' availability [9].

In this paper we propose SYRINGE, an infrastructure for monitoring VMs that combines the advantages of out-of-guest and in-guest approaches. SYRINGE satisfies both security and robustness requirements by placing the monitoring application in an isolated SVM, as done by the out-of-guest approach, but still using the GVM's own code for monitoring, as done by the in-guest approach. For this to work, (1) the SVM-resident monitoring application must be able to call GVM functions and (2) the security of the GVM's code execution must be verifiable. These problems are respectively addressed by two techniques: *function-call injection* and *localized shepherding*.

Function-call injection allows a monitoring application to be placed in the SVM and still be able to invoke functions in the GVM. By carefully interrupting the GVM's execution and manipulating the contents of its virtual CPU and memory through introspection, SYRINGE is able to *inject* a function call into the GVM. That is, when the virtual CPU is resumed, it executes the selected function as if it had just been called from inside the guest.

Localized shepherding is a novel technique for monitoring the execution of the invoked guest code against attacks. It is *localized*, because it only shepherds the thread of guest code executed as a result of an injected function call, and only until it returns, being then disabled. First, it verifies the code in memory against a pre-compiled whitelist to prevent code-patching attacks [10]. By using instrumentation, it also dynamically evaluates instructions that can be used by an attacker to divert the code's legitimate control flow. With this, SYRINGE is able to detect attacks such as hooking [10] and return-oriented programming [11, 12]. Finally, it enforces atomic code execution to prevent unauthorized tampering with temporary function state. When the invoked function returns, all the instrumentation is undone and the guest continues executing normally.

SYRINGE combines function-call injection and localized shepherding to create a robust VM monitoring infrastructure with strong security properties. It avoids the semantic gap inherent to introspection by using guest OS API code instead of directly parsing and reading data structures in memory. As such, changes in the syntax and semantics of guest data structures commonly performed by patches and new software releases do not affect SYRINGE, as long as the public exported API remains unaltered. The localized shepherding of guest code ensures that most attacks directed against it will be either prevented or detected, allowing the monitoring application to be notified.

We have implemented SYRINGE as a Linux library, using Windows XP as our monitored guest OS, VMware ESX Server as the hypervisor, and VMware's VMsafe API as our introspection engine [13]. A monitoring application based on SYRINGE, SYR-Mod, was implemented to illustrate SYRINGE's capabilities. Our performance evaluation revealed a maximum 8% guest performance overhead when the interval between successive SYRINGE monitoring operations is above or equal to 1 second, for a reasonably complex guest monitoring function. Our security evaluation tested the effect of common malware attacks on SYRINGE's monitoring. In all cases the attacks were prevented or detected.

In summary, in this paper we claim the following contributions:

- Localized shepherding, a novel virtualization-based technique for verifying the secure execution of guest code in a localized, on-demand fashion, using on-the-fly instrumentation;
- The SYRINGE VM monitoring infrastructure, which combines function-call injection and localized shepherding to achieve secure and robust VM monitoring. We implemented a fully-functional prototype of SYRINGE on a platform running VMware ESX Server and using VMware's VMsafe introspection engine;
- An evaluation and discussion of SYRINGE's performance and security against attacks targeting the monitoring application and the guest.

This paper is organized as follows. Section 2 describes in detail the design and implementation of SYRINGE. Section 3 describes our performance and security evaluations of SYRINGE. Section 4 describes our monitoring application and Section 5 discusses the limitations of our system. Finally, Section 6 describes related work and Section 7 concludes our work.

2 SYRINGE VM Monitoring Infrastructure

SYRINGE's design process started from a basic in-guest monitoring architecture, which, as discussed previously, already incorporates the desired robustness. We then focused on determining what additions and modifications should be done to it so as to make it secure. In more concrete terms, this meant securing the two high-level entities involved in in-guest monitoring: (1) the monitoring application and (2) the execution of guest OS functions invoked by the application.

Protecting the Monitoring Application. In this work we assume that the monitoring application is a user-space program. This assumption is based on the fact that most real-world monitoring applications such as AV scanners, intrusion detection systems and system diagnostic tools are implemented in user-space application. Fully protecting a user-space application (monitoring or otherwise) running inside an untrusted guest OS is a hard problem. As demonstrated by Chen et al., it is possible to use virtualization to protect the confidentiality and integrity of its code/data [9]. However, the control that the guest OS has over the application's resources (CPU time, memory, etc) means that it is extremely difficult to ensure the application's availability on an untrusted guest OS. In other words, it would be easy for an attacker who has compromised the guest OS to disable the application, or deny it essential computing resources controlled by the OS. For these reasons, in SYRINGE we opted to remove the monitoring application from the GVM, placing it in an isolated, trusted SVM. This move allowed us to secure the application, but disrupted its ability to invoke guest OS functions. We solved this problem with the *function-call injection* technique. Function-call injection enables the monitoring application to be moved out of the GVM, but still retain the ability to invoke guest functions by injecting function calls into the GVM. This technique works by interrupting the GVM's execution at a pre-determined point and manipulating the contents of its virtual CPU and memory using introspection, setting it to the desired target function with the desired parameters. In its current form, SYRINGE only supports the injection of function calls to kernel functions.

Protecting the Execution of the Invoked Guest OS Functions. We refer to the execution thread triggered inside the guest as a result of the function-call injection as the *monitoring thread*. To protect the execution of the monitoring thread we introduce a novel technique: *localized shepherding*. This technique basically performs on-demand monitoring of the control-flow integrity of the monitoring thread by using on-the-fly instrumentation, in accordance to a policy that we defined to address the most common attacks that rely on control-flow manipulation. Together with function-call injection, localized shepherding also ensures the *atomic execution* of the monitoring thread. This property is necessary to prevent malicious threads from tampering with the monitoring thread's local state when their executions are interleaved. Atomic execution is implemented by disabling interrupts at the start of the monitoring thread and shepherding interrupt-related instructions to prevent them from being re-enabled.

SYRINGE *was not* designed as a general security system. Its goal is not to defend the guest against attacks in general. SYRINGE focuses on the task of determining whether the data returned by the monitoring thread to the monitoring application results from a secure execution. If SYRINGE detects any form of tampering with the monitoring thread, such as a control-flow violation, it will not attempt to repair it. For safety, it will allow the monitoring thread to continue executing unshepherded, but will notify the monitoring application in the SVM that the results returned by the function should not be trusted. An attacker can exploit this fact to disrupt SYRINGE's monitoring, effectively causing a Denial of Service. The monitoring application, however, will know at this point that the system has been compromised, at which point the best course of action may be to restore the GVM to a previous snapshot or employ another type of remediation procedure.

The atomic execution property enforced by SYRINGE creates some functional limitations. First, SYRINGE cannot shepherd guest code that relies on asynchronous code execution, such as I/O or Deferred Procedure Calls (DPCs). This prevents certain types of exceptions, such as page faults, from being handled properly. We do a detailed discussion of these limitations in Section 5.

2.1 Assumptions

In this work, we assume an underlying x86 architecture running a hypervisor with two virtual machines: a monitored guest virtual machine (GVM); and a security virtual machine (SVM) in which SYRINGE will be deployed. Our whitelisting-based code integrity approach also assumes previous access to legitimate copies of the binaries composing the guest OS's kernel. On Windows, this includes the kernel executive (NTOS) and other kernel-level modules. It *does not* include 3rd-party modules. We believe this to be a reasonable assumption, given that this set of binaries is manageable in size and relatively homogeneous for each particular OS version. A database of such binaries can be easily created and automatically updated, for instance, when patches are issued by the OS vendor. We also assume knowledge of the public Windows kernel API, which includes function prototype and parameter type definitions.

Knowledge of the base address in guest memory for each loaded whitelisted binary is also assumed. This information can be obtained through a variety of methods and heuristics that are orthogonal to this work [14], and are thus not detailed here. We further assume in our threat model that the GVM can be fully compromised by an attacker, including its kernel. The system hardware, hypervisor, and SVM constitute our trusted computing base.

2.2 Function-Call Injection

Function-call injection (FCI) secures the monitoring application by placing it in an isolated SVM, while still keeping its ability to invoke GVM functions. This is the first piece of our solution to the problem of creating a secure and robust VM monitoring infrastructure. FCI essentially provides the ability for code running in one VM to call a function in another VM and retrieve its results. FCI uses simple VM introspection techniques. It can be viewed as a type of inter-VM Remote Procedure Call (RPC), but without the need for an RPC server running on the destination.

SYRINGE currently assumes that the GVM only has one virtual CPU (VCPU) in order to ensure atomicity for the monitoring thread. Multiple VCPU support would require the virtualization infrastructure to be able to suspend individual VCPUs during the guest's execution. This is not the case, however, with ESX/VMsafe. Although this assumption may be limiting for certain types of VMs, we believe it to be a consequence of a platform limitation, rather than a fundamental flaw in our approach.

The first step in FCI is to interrupt the execution of the guest so that a function call can be injected. Pre-selected *injection contexts* designate the execution contexts under which the guest must be interrupted so that a function-call injection may occur. An injection context is a tuple (P_S, A_I), where P_S represents a *surrogate process* and A_I is an *injection address*. FCI can only happen when process P_S is currently active in a

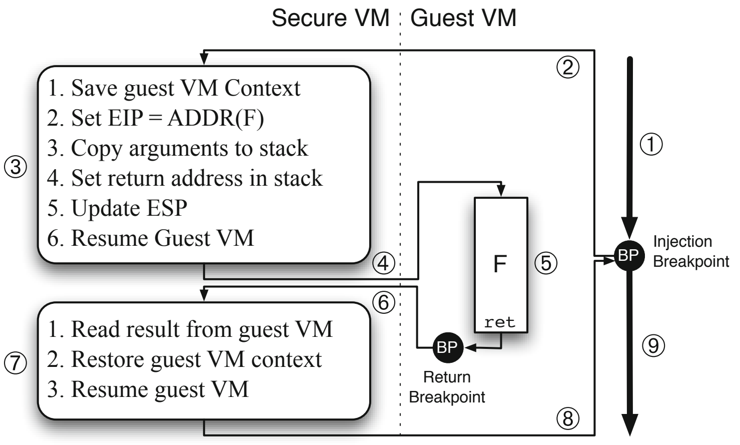


Fig. 1. Injecting a call to guest function F. (1) The GVM executes normally until it reaches an injection context; (2) a breakpoint placed at the injection address transfers execution to the SVM and suspends the GVM; (3) The SVM saves the guest VCPU context and sets its EIP to point to F's start address and copies F's arguments to the stack, also updating its ESP; (4) the guest VCPU is resumed and function F starts execution, as if it had just been called by guest code; (5) F is executed; (6) control is returned to the SVM through another breakpoint placed at F's return address; (7) the guest VCPU's context is set by the SVM to the saved context (8 and 9) when resumed and it continues running from the point where it was originally interrupted.

guest virtual CPU (VCPU) *and* the instruction at A_I is about to be executed by that same VCPU. A surrogate process is identified by the physical address of its page directory table (stored in the CR3 register). Each surrogate process can have its own injection addresses, or they can be shared between multiple surrogates. Injection addresses can be selected in the guest's kernel-space, for injecting calls to kernel functions, or in user-space for injecting calls to user-level API functions. Multiple distinct injection contexts can be used. To minimize injection delay, it is important to choose injection contexts that are reached frequently enough in the GVM's normal execution. They must also not be easily circumvented by a malicious entity in control of the guest OS. Details concerning our choice of injection context are given in Section 2.4.

SYRINGE interrupts the GVM by using VMsafe page-table level breakpoints placed at the injection addresses. We call these *injection breakpoints*. This type of breakpoint cannot be detected or tampered with by the guest OS because it is implemented at the hypervisor level, and is therefore transparent to the guest. Whenever the instruction corresponding to the injection address is executed, a trap is triggered, the GVM is suspended, and control transferred to the hypervisor, and then to SYRINGE in the SVM. SYRINGE then checks if the current CR3 value of the guest's VCPU corresponds to that of a surrogate process associated with the injection address where the execution was interrupted. In case it does, SYRINGE determines that an injection context has been reached. Injection contexts are only made active (i.e., the hypervisor-level breakpoints are activated) when SYRINGE has requests queued for function-call injections, otherwise the system runs normally without any performance penalty. In its current form, SYRINGE allows only one monitoring thread to be running in the GVM. In other words, function-call injections cannot overlap each other.

The operation of FCI is shown in Figure 1. Let us assume a function call $F(A_0, ..., A_n)$, i.e., a call to the guest kernel function F with arguments A_i. Let us also assume a `stdcall` or `cdecl` calling convention, so that arguments are placed on the stack, in reverse order. When the injection breakpoint is triggered, SYRINGE first saves the guest VCPU's context so that it can be restored later and then sets the VCPU's EIP register to F's starting address. F's offset in its corresponding binary can be extracted from the binary's export table. Knowledge of the binary's base address in memory is listed as part of our assumptions and is obtained when SYRINGE is initialized.

Next, the stack needs to be set with arguments A_i and a return address. This is done by using memory introspection to map the guest memory region corresponding to the value of the VCPU's ESP register and making the necessary changes. Arguments are handled according to their evaluation semantics. Call-by-value arguments are copied directly onto the stack. Call-by-reference arguments require a more careful treatment. The data buffer referenced by the argument must be copied to the guest and the reference itself must be placed on the stack as an argument. SYRINGE provides two ways of doing this. The simplest way is to place the data structure at the bottom of the current stack frame and push a reference to it. Another possibility is to allocate a special memory buffer inside the guest (for example, by injecting another function call to a memory allocation function) and use it to store the referenced data structure. This is useful in the case where the structure is too large to be placed on the stack. The return address is set to a special memory location inside the guest containing another VMsafe execution breakpoint—the *return breakpoint*—placed by SYRINGE. This can be any memory location whose page does not contain valid code, so as to avoid unnecessary VM switches caused by execution of code.

Once the stack is set, the value of ESP is updated to accommodate the arguments and return address. To ensure atomicity, the guest's VCPU state is modified so that regular guest interrupts, hardware breakpoints, instruction-tracing exceptions and performance-monitoring interrupts (PMIs) are disabled when the monitoring thread starts executing. This is done by clearing the IF (Interrupt Flag) bit in the guest VCPU's EFLAGS register, bits 1 and 8 in `IA32_DEBUGCTL`; bits 0, 1, 32–34 in `IA32_PERF_GLOBAL_CTR`; and bits 0–9, 13 in `DR7`.

Finally, at this point, the guest VCPU is resumed. F then begins to execute as if it had been called with arguments A_i from inside the surrogate process, at the injection address. This is our monitoring thread. At this point, the localized shepherding component (described in Section 2.3) takes over and shepherds the monitoring thread. When the final `RET` instruction is reached, the return breakpoint is triggered, suspending the VCPU and passing control back to SYRINGE in the SVM. At this point, the result of F's execution is read from the `eax` register and returned to the monitoring application. If any memory buffers have been passed by reference on the stack or heap to receive results from the function, it is the monitoring application's responsibility to retrieve their contents. Finally, SYRINGE restores the original VCPU context that was saved when

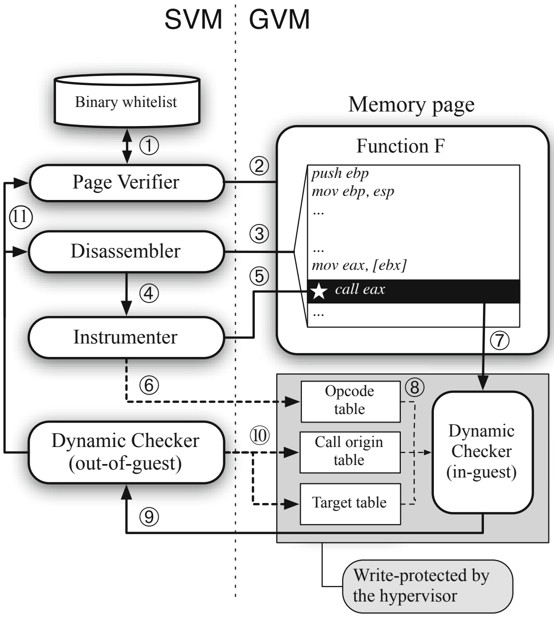


Fig. 2. Localized shepherding of function F. (1) Page Verifier pre-builds a whitelist of the OS kernel binaries. (2) Upon injection, Page Verifier verifies the code regions of the target page against the whitelist, (3) Disassembler recursively disassembles the target function, recording the locations of critical instructions and (4) passing them to the Instrumenter. (5) Instrumenter patches all critical instructions with `INT3` breakpoints and (6) updates the in-guest opcode table. (7) When triggered, a critical instruction breakpoint transfers control to SYRINGE's in-guest Dynamic Checker. (8) It consults the in-guest tables to determine whether it can evaluate the instruction by itself. (9) If not, it passes control to the out-of-guest dynamic checker, which (10) updates the in-guest call origin and target tables and, if necessary, (11) re-invokes the Disassembler to analyze new code and the Page Verifier, if the control-flow has transitioned into a new page. This process is conducted for all subsequent function calls.

the guest OS was first interrupted and resumes the GVM, which continues its original execution thread from the point where it was interrupted.

2.3 Localized Shepherding

Localized shepherding is the second piece of our solution to the problem of creating a secure and robust VM monitoring infrastructure. Localized shepherding monitors the control-flow integrity and ensures the atomic execution of the monitoring thread. Figure 2 illustrates this process and the role played by each component.

Control-flow integrity is monitored by: (1) checking that all guest code executed by the monitoring thread matches the pre-compiled whitelist database of OS API binaries and (2) dynamically evaluating indirect control-flow transferring instructions in accordance to a pre-specified policy. Action (1) guarantees the integrity of direct branches while action (2) monitors the integrity of indirect branches. Thus, together, they cover all control-flow transfers. Action (1) further ensures that non-control-flow related instructions are not modified by an attacker. If a control-flow integrity violation is detected, SYRINGE allows the execution of the monitoring thread to proceed but sends an alert to the monitoring application in the SVM. This alert indicates that malicious tampering has been detected during the execution and therefore the results returned by the monitoring thread cannot be trusted.

Code integrity checking (action (1)) is performed by the Page Verifier component through binary whitelisting. As stated in our assumptions, we assume previous access to legitimate copies of the binaries composing the OS API (both user-space libraries and kernel modules). These binaries are analyzed in an offline manner by the Page Verifier. Based on the metadata and content of each binary's PE sections, the Page Verifier constructs a database containing the location, size, and SHA1 hash corresponding to each code (executable) section in each binary. This information is used at runtime to check the integrity of the code being executed in the guest. Immediately before a function call to F is injected, SYRINGE activates the Page Verifier to check the integrity of all the code present in the page where F's starting address is located. If the page contains a mixture of code and data, each code region in the page is checked individually. A SHA1 hash is calculated for each code region and is compared against its corresponding whitelisted hash. Any discrepancies indicate that the code has been modified. This allows SYRINGE to detect *code patching* attacks, where an attacker maliciously modifies the guest code, and notify the monitoring application. This process is repeated whenever the control-flow of the monitoring thread transitions into a new page. After being checked and before execution is allowed to begin, code pages are marked as write-protected again by using VMsafe. This marking avoids the need for future checks and prevents time-of-check-time-of-use (TOCTOU) race conditions.

Indirect branches (action (2)) are monitored by the Disassembler, Instrumenter, and Dynamic Checker components. These components employ a combination of dynamic recursive disassembly, code instrumentation and reference monitoring. The Disassembler performs a recursive disassembly of the function, stopping at indirect control transfer instructions and direct function calls. During this disassembly, it records the location of all instructions whose execution needs to be trapped and evaluated at runtime to ensure control-flow integrity. We refer to these instructions as *critical instructions* and they are shown in the top part of Table 1. When the Disassembler is done analyzing the function, the Instrumenter instruments all critical instructions so that they can be evaluated by SYRINGE before being executed. This instrumentation consists of an `INT3` instruction that overwrites the first byte of the critical instruction. The overwritten byte is recorded by SYRINGE in a write-protected in-guest *opcode table*. The entry #3 of the guest Interrupt Descriptor Table (IDT) is set to point to the in-guest component of the Dynamic Checker. The guest's IDT is write-protected by SYRINGE. The Disassembler is invoked only for those cases where the target in question has not been analyzed previously; otherwise, cached results are used by the Instrumenter for performance.

The Dynamic Checker is responsible for evaluating critical instructions according to our control-flow integrity policy. This policy is shown in the top part of Table 1.

Table 1. SYRINGE's shepherding policy and handler types for critical instructions. The top part shows those instructions related to control-flow integrity monitoring. The bottom part shows those related to atomic execution enforcement.

Instruction	Description	Policy/Action	Handler
Control-flow integrity			
`CALL r/m32`	Indirect function calling	Target must be in trusted code regions. Update in-guest call-origin table	In-guest and Out-of-guest
`JMP r/m32`	Indirect jumping	Target must be in current module	In-guest and Out-of-guest
`RET`	Function returning	Target must be present in pseudo-shadow stack	In-guest
Atomic execution			
`STI/CLI`	Interrupt enabling and disabling	Skip instruction	In-guest
`POPF`	EFLAGS popping	Emulate, clearing the IF and TF bits in the EFLAGS	In-guest
`WRMSR`	Writing to MSR_IA32 DEBUGCTL or IA32 PERF_ GLOBAL_CTR	Emulate, clearing bits 1 and 8 in the IA32_DEBUGCTL_MSR and bits 0, 1, 33–34 in IA32 PERF_GLOBAL_CTR	In-guest
`MOV DR7, *`	Writing to DR7 debug register	Emulate, clearing bits 0–9 and 13 in the DR7 register	In-guest

It has an in-guest component, which implements handlers for those critical instruction invocations that do not need to be handled in the SVM. In-guest handling of critical instructions greatly favors performance in comparison to out-of-guest handling, as it does not require VM switches. The in-guest handlers are injected into the guest by SYRINGE and are write-protected with support from the hypervisor. This protection is effective because in-guest handlers do not require any persistent state to be maintained and are present only when a monitoring thread is being executed. Thus, code write-protection suffices to ensure their good behavior. The in-guest component is invoked by all critical instructions. It determines the type of instruction by consulting the opcode table and whether the instruction can be handled in-guest or not. If not, it generates a trap so that the Dynamic Checker's out-of-guest component can handle it.

Direct `CALL` instructions do not need to be dynamically evaluated, but are instrumented nevertheless so that their targets can be properly scanned and instrumented before execution is allowed to continue. This instrumentation is only needed until the instruction's first execution, however, and is then removed. They are handled out-of-guest. The target of indirect `CALL`s must be evaluated dynamically at every execution. The in-guest handler first determines if the computed target of the indirect `CALL` has been analyzed before, by consulting an in-guest *target table*, maintained by SYRINGE.

This table is write-protected inside the guest. If so, then the target is legitimate and execution is allowed to proceed. If not, the in-guest handler generates a trap and passes control to the out-of-guest handler. The later then applies the following policy: the target must be located inside the authorized memory ranges containing the whitelisted system code, as determined previously by the Page Verifier. If this policy is satisfied, the target is added to the target table. The handling of all CALL instructions (both direct and indirect) also includes adding the address of the CALL to an in-guest *call origin table*. This table is also write-protected and is used for the evaluation of RET instructions, explained later. Indirect JMP instructions are handled exactly as described for indirect CALLs with the one following policy difference: their targets must be located inside the current module. The policies used for indirect CALLs and JMPs can detect a large portion of attacks that rely on *hooking* [10] function and code pointers to hijack control-flow.

All RET instructions are handled in-guest, except for the last one. The handler evaluates the RET by comparing its target against all the addresses contained in the call origin table. The RET is considered legitimate if a match is found. This model differs from a shadow stack in that, for a particular RET, the address of its originating CALL is not necessarily at the top of the stack. As a result, our model allows a RET to return to the origin point of any CALLs that were executed previously by the monitoring thread. A complete shadow stack implementation would require the RET in-guest handler to have write access to the call-origin table, and thus open way to attacks. Thus, we decided against it. Despite not being ideal, the number of allowed return targets is significantly constrained so we believe that this policy is powerful enough to detect most return address manipulation attacks such as *return-oriented programming* [11, 12].

Localized shepherding must also ensure that the monitoring thread is executed atomically. As described in Section 2.2, FCI clears the IF flag in the VCPU's EFLAGS register, thereby ensuring that the monitoring thread will start executing with interrupts disabled. Localized shepherding must ensure that they remain disabled throughout its entire execution. The Instrumenter patches another set of critical instructions that can affect atomic execution, and are shown in the bottom part of Table 1. Instructions CLI and STI are commonly used in OSes to execute critical code sections atomically by temporarily disabling interrupts. SYRINGE's policy is to simply skip these instructions. Thus, they are simply overwritten with a NOP by the Instrumenter for the duration of the monitoring thread's execution. All other critical instructions are patched with int3, and are handled in-guest by the Dynamic Checker. The handler for POPF pops the stack into the guest's EFLAGS and clears the IF and TF flags. SYRINGE ensures that hardware debugging and instruction tracing facilities remain disabled by handling instructions WRMSR, when the destination is MSRs IA32_DEBUGCTL or IA32_PERF_GLOBAL_CTR; and MOV, when the destination is the CPU debugging control register DR7. Bits 1 and 8 are cleared in IA32_DEBUGCTL; bits 0, 1, 32–34 in IA32_PERF_GLOBAL_CTR; and bits 0–9, 13 in DR7.

With regard to multiprocessing, our assumption that the GVM has just one VCPU guarantees that simultaneous code execution in other CPUs is not an issue.

When the monitoring thread finishes executing (i.e., executes the final RET instruction) and SYRINGE reassumes control, the guest is restored to its original state.

At this point, all patched critical instructions are un-patched and the IDT is restored to its original content. Likewise, if external interrupts were enabled when the GVM was interrupted by the FCI component, they will again be enabled when the guest is resumed. This localized, on-demand variant of shepherding satisfies our secure monitoring properties while at the same time does not affect the guest's normal performance when monitoring operations are not being conducted.

Software exceptions present a challenge for localized shepherding. SYRINGE is capable of shepherding exceptions that happen during the execution of the monitoring thread, as long as these exceptions are handled synchronously. This shepherding is done by installing a read-breakpoint in the memory page containing the IDT, so that any attempts to access a descriptor (as should happen during an exception) are trapped and transfer control to SYRINGE. From this point on, the process is the same as the one described above for regular function call invocations.

Exceptions requiring asynchronous activity, such as I/O, cannot be shepherded by SYRINGE. This is a limitation shared with other works that leverage guest code, such as Virtuoso [15]. Non-maskable interrupts (NMIs) are not handled either, since they usually indicate a fatal hardware error that would require the GVM to be rebooted or restored to a previous snapshot.

2.4 Implementation

SYRINGE was implemented as a Linux library using approximately 3,500 lines of C and Python code. The function-call injection component makes up one third of the code, while the localized shepherding code is the rest. VMware's ESX Server 4.1 was used as the hypervisor and VMware's VMsafe was used as our introspection infrastructure [13]. VMsafe natively provides the introspection and breakpoint functionality used by SYRINGE. Despite page-level breakpoints not being provided by other open-source introspection infrastructures and hypervisors (such as XenAccess [2]), we would like to emphasize that the mechanics of this technique are simple and well understood and could therefore be incorporated into them.

In our prototype, we chose the OS's system call dispatcher as the injection address, and we allow all processes running in the system to act as surrogates. We selected as our return breakpoint address location the start of the `.data` section of the kernel executive module (NTOS).

3 Evaluation

We conducted a performance and security evaluation of SYRINGE. Our host machine was an Intel Core i7 870 2.93GHz with 4 CPU cores, 8GB of RAM, running VMware ESX Server 4.1. The GVM was configured with 1 VCPU, 1GB of RAM, running Windows XP SP2. The SVM was configured with 1 VCPU, 1 GB of RAM, running Linux CentOS 5.5.

3.1 Security

We now analyze and evaluate SYRINGE's security properties. Again, SYRINGE's goal is not to to act as a general attack prevention system. Its goal is to be able to tell, based on the localized shepherding of the monitoring thread, wether the results returned by the invoked guest function can be trusted or not and notify the monitoring application. So it is possible for an attacker to cause a monitoring Denial of Service by repeatedly attacking the monitoring thread, but not without SYRINGE knowing about it. At this point, remediation rather than monitoring becomes the main concern.

SYRINGE applies a mixture of prevention and detection techniques against attacks directed at itself and the monitoring thread.

Attacks against SYRINGE's Components. Attacks against the monitoring application and SYRINGE's out-of-guest components are prevented by the isolation between the GVM and SVM. In-guest components are protected as follows. For FCI, the injection and return breakpoints cannot be tampered with or disabled, since they operate at the hypervisor level. Our choice of injection context (the system call dispatcher), combined with the continuous monitoring of the MSR_SYSENTER_EIP register, ensure that it cannot be easily circumvented.

For localized shepherding, several components are involved. The `INT3` instrumentation used to trap on critical instructions is protected by the write-protection of guest code, which prevents the guest from modifying pages containing code being shepherded. This `INT3` relies of the guest's IDT to pass control to the dynamic checker's in-guest component. The IDT is write-protected from inside the hypervisor, preventing any modifications from inside the guest. The dynamic checker's in-guest component consists of a code segment and three tables: the call origin table, the opcode table and the target table. All three tables are write-protected, and can only updated by SYRINGE's out-of-guest components in the SVM. Since the code does not rely on any data maintained by the guest OS (only the three tables) and does not itself maintain any persistent data across dynamic checker invocations, write-protecting its code is enough to protect it against attacks. It is conceivable that an attacker could attempt to modify guest page table mappings so that invocations to the in-guest dynamic checker could fail. While this attack is possible, the memory pages containing the dynamic checker and the three tables are locked in memory and its corresponding mappings are constantly monitored by SYRINGE. Any changes to these mappings, being unexpected, are interpreted as an attack and the monitoring application is notified.

Attacks against the Monitoring Thread. SYRINGE's localized shepherding guards the monitoring thread against attacks using a combination of prevention and detection techniques.

Attacks attempting to patch the guest code cannot succeed given that, before being executed, all code is checked by the Page Verifier against the binary whitelist database and then write-protected. This step occurs atomically to eliminate the possibility of a time-of-check-time-of-use race condition, where the code could be modified after being checked and before being write-protected.

Table 2. Security evaluation results. SYRINGE was able to detect all the attacks and notify the monitoring application.

Attack	Target	Result
Code Patching	`nt!ZwQueryInformationProcess` code	**Detected** by *Page Verifier*
Hooking	`KeServiceDescriptorTable[]` pointer	**Detected** by *Dynamic Checker*
Return-into-libc	Return address on monitoring thread's stack	**Detected** by *Dynamic Checker*

Indirect control-flow instructions are patched by the Instrumenter and are evaluated dynamically. Attacks directed at these instructions have their power severely restricted by SYRINGE's control-flow integrity policy. Function pointers cannot point anywhere outside whitelisted code, indirect jumps cannot point anywhere outside their own module and function returns must target the instruction following a CALL executed previously in the monitoring thread. This policy greatly restricts the effectiveness of function pointer hooking, which in most cases rely on injected code; and jump-oriented [16, 17] and return-oriented programming [11, 12], which need access to a large code base to extract a good variety of gadgets. More fine-grained policies can be integrated into SYRINGE by using techniques such as alias analysis.

To empirically validate our claims above, we simulated one code patching, one hooking and one return-into-libc attack. Results are show in Table 2. A function call to ZwQueryInformationProcess was injected and then shepherded. The first attack patched function ZwQueryInformation Process's code, and was detected by SYRINGE's Page Verifier, which is responsible for checking the integrity of code sections before they are executed. Hooking was performed on the system service descriptor table (KeServiceDescriptorTable), on the entry corresponding to the NtQuery InformationProcess system call. Since the hooked address is used by an indirect function call instruction inside the system call dispatcher, it was trapped and evaluated by the Dynamic Checker, which detected the attack, since the address pointed to injected code. Return-into-libc was also detected by the Dynamic Checker, when it could not find the function's return destination in the call origin table, indicating that it had been modified.

The atomicity property enforced by SYRINGE makes it difficult for attacks to tamper with the monitoring thread's state in the stack or heap. This state is only valid during the time when the monitoring thread is executing, and we can be sure that no other potentially malicious thread will be running during that time, and cannot tamper with it. The only way to do so would be to exploit a software vulnerability, such as a stack or heap overflow, in the shepherded code, so that the monitoring thread itself does the tampering. This could be difficult however, given that the arguments passed to the top-level function in the monitoring thread are controlled by SYRINGE.

3.2 Performance

In this section, we investigate the performance of function-call injection and localized shepherding. The reported results are wall-clock times, derived from the host CPU's

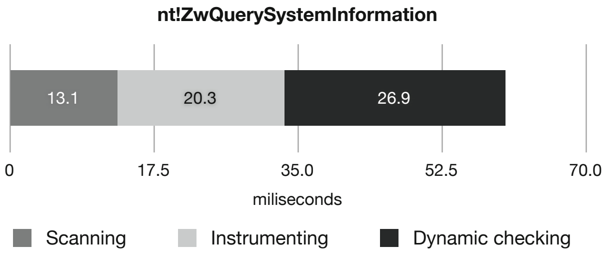


Fig. 3. Shepherding execution time breakdown for the Windows executive's `ZwQuerySystemInformation` function, when used for a common monitoring task: obtain the list of active modules in the guest. In the scanning phase represented above, 3163 bytes of code were disassembled by the Disassembler and 12 4KB code pages were verified and write-protected by the Page Verifier. In the instrumenting phase, 53 critical instructions and 23 direct calls were patched/unpatched by the Instrumenter. Finally, in the dynamic checking phase, 17 critical instruction executions and 9 direct calls were handled by the out-of-guest Dynamic Checker and 316 critical instructions executions were handled by the in-guest Dynamic Checker.

timestamp counter. In all experiments, five samples were taken for each measurement and the median was used.

Function-call injection was evaluated by injecting a function call to a Windows kernel function and measuring the time between when the injection starts and the target guest function starts running (steps 2–4 in Figure 1). For this experiment, no parameters were passed to the function. The entire operation, consisting of CPU and memory introspection operations, followed by a VM switch, consumed an average of 0.7ms, with a very low variance. We also measured the triggering delay for our selected injection context, the OS system call dispatcher. This delay indicates the amount of time that a monitoring application has to wait between its request for a function call to be injected and the moment of injection. Results varied widely, ranging from a minimum of 18ms to a maximum of 51ms, with a mean of 33ms. We consider this number to be acceptable for most monitoring applications.

Localized shepherding was evaluated by injecting a function call to a guest OS function and measuring the execution time consumed by each shepherding component. Performance measurements corresponding to the run with the median execution time and other shepherding statistics are shown in Figure 3. For this experiment, we selected a function from the Windows kernel executive commonly used for monitoring: `ZwQuerySystemInformation`.

The scanning and instrumenting phases were dominated by inter-VM page copying operations that VMsafe uses for memory introspection. Performance could be improved by using sharing-based introspection such as used by XenAccess. The dynamic checking phase used about 50% of the total execution time, as shown in Figure 3, totaling 28.5ms. This time is almost entirely consumed by context switches between the SVM and the GVM for critical instructions that need to be handled out-of-guest, and first-time execution of direct calls, which are also handled out-of-guest. Considering that

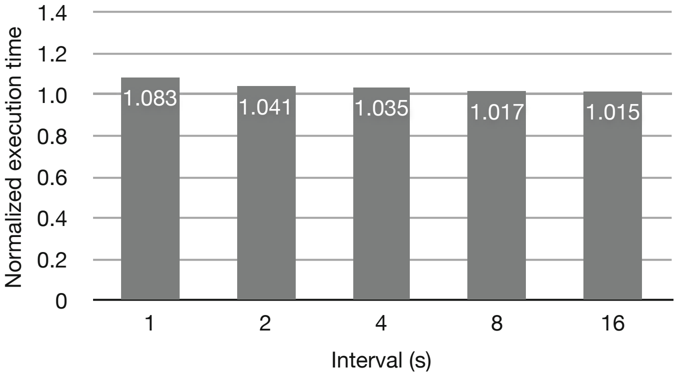


Fig. 4. Normalized execution time for the decompression of the Linux kernel source code tree in the GVM. The interval between successive calls to ZwQuerySystemInformation is varied.

these 28.5ms correspond to just 17 critical instructions and 9 direct calls being handled out-of-guest (out of 325), averaging 0.91ms per instruction, the importance of in-guest handling cannot be overemphasized. If we were to handle all critical instructions out-of-guest, the execution overhead created would make SYRINGE impractical for use in any virtualized environment. Of the 17 instructions handled out-of-guest, 1 was an indirect `CALL` executed by the system call dispatcher and 16 were indirect jumps triggered by two different instructions. These 17 executions were handled out-of-guest because the in-guest Dynamic Checker, by consulting the in-guest target table, determined that their targets were being reached for the first time and so needed to be analyzed and instrumented by SYRINGE. These were included in the target table to indicate that all future indirect `CALL`s and `JMP`s instructions targeted at those addresses could be handled in-guest. This allowed the following 88 executions of these instructions to be analyzed in-guest. Direct and indirect `CALL` instructions, in their first execution, also have the in-guest call origin table updated, so that `RET` instructions can be evaluated in-guest. In our example, 211 `RET` instructions were executed and handled in-guest. The execution of the actual guest code consumes an insignificant amount of time when compared to shepherding, and is not shown in Figure 3.

4 Example Application

After measuring the performance of its individual components and its security properties, we evaluated SYRINGE in the context of a rudimentary monitoring application. This application, named SYRMod, uses SYRINGE to periodically obtain a list of the user and kernel modules loaded in the current process' address space. The calling interval can be defined by the application's user. This module list is obtained by injecting a call to and shepherding guest OS kernel function `ZwQuerySystemInformation`. This is a generic wrapper function that can be used to obtain information about a Windows system.

After each invocation, when ZwQuerySystemInformation returns, SYRMod is notified by SYRINGE. SYRMod then uses regular introspection to retrieve the results from the guest OS's memory, parsing and printing the list of modules to the SVM's standard output. This output includes, for each loaded module, the filesystem path of its corresponding binary, base address in memory and size. The correctness and completeness of the list was verified. No security alerts were raised during the shepherding phase, indicating that no integrity violations were detected with the code's execution, and that the results can be trusted.

We evaluated SYRMod's performance impact on the guest. We varied the time interval between successive callings of ZwQuerySystemInformation and measured the time taken inside the guest to decompress the 2.6.33.7 Linux kernel source tree, a 64MB tar.bz2 file. The performance penalty comes from the suspension of all guest activity when the monitoring thread is running. Results show a maximum overhead of 8% for a calling period of 1 second (Figure 4). Setting this period obviously depends on the nature of the monitoring application and the type of information being retrieved. We believe, however, that for many monitoring applications a 1 second period is considered low enough so that our results indicate an acceptable performance penalty.

5 Limitations and Future Work

The techniques used by SYRINGE have limitations. Perhaps the most noticeable one is SYRINGE's intrinsic inability to shepherd the complete execution of certain operations. Specifically, operations involving any type of asynchronous code execution, such as reading a file from disk, cannot be shepherded by SYRINGE. This limitation is inherent to the shepherding technique. We do not see it as a deal-breaker, though, since I/O and asynchronous primitives (e.g., deferred procedure calls) are not commonly used by information querying functions, such as ZwQuerySystemInformation.

The handling of page fault exceptions is also affected by this problem. In this case, our current solution relies on first performing a non-shepherded call injection to the target guest function so that all necessary pages can be swapped in, and then following it with a shepherded call injection. Attacks targeting the non-shepherded call, by modifying the code read from disk, would later be detected during the shepherded call.

The effect of guest OS internal synchronization mechanisms can be problematic for SYRINGE. Due to the disabling of interrupts, it is possible for the shepherded code to be blocked indefinitely by synchronization constructs, creating a deadlock situation. We only rarely came across such a situation in our tests, but a more careful investigation on how to prevent and detect such occurrences is needed. One possible solution would be to create a shepherding timeout mechanism that, if triggered, would cause SYRINGE to re-enable interrupts inside the guest and notify the application of the fact. The application can then choose to re-invoke the function at a later time.

Due to SYRINGE's reliance on guest OS's functions, it could be argued that its monitoring capabilities are not as powerful as those of regular memory introspection. The first is restricted to the results returned by a finite set of guest OS functions, while the second can in principle retrieve any information from the guest's memory. This argument assumes that SYRINGE aims to completely replace regular introspection, which is not true. We envision SYRINGE as an information extraction tool that, due to its resilience to the semantic gap, can be used to aid many uses of regular introspection.

Finally, the control-flow integrity policy enforced by SYRINGE is coarse. Still, it is sufficient to block kernel injected-code attacks, which is the most common type. A more precise policy could be constructed by using techniques such as points-to analysis to derive a smaller set of possible destinations for indirect control flow transfers and include that knowledge in the dynamic checker's policy [18].

6 Related Work

Secure monitoring of virtual machines has received much attention in the past 10 years from academia and industry. The seminal work by Garfinkel et al. first introduced the concept of virtual machine introspection (VMI), whereby the state of a GVM is passively analyzed by a monitor placed in a separate VM [1]. Multiple VMI-based solutions have since been proposed to address specific problems such as tracking the execution of guest processes [19], identifying covertly executing binaries [20], verifying semantic integrity constraints [4], detecting persistent control-flow integrity violations [5], and detecting past vulnerability exploitations through record and replay [21]. These solutions are vulnerable to the semantic gap problem.

The protection of in-guest monitors has also been explored. Lares [7] and SIM [8] protect a small agent inside the guest using hypervisor memory protection and additional address spaces. Nevertheless, the agent is subject to significant limitations that would not allow such schemes to be used with sophisticated monitoring tools, such as AV scanners. SYRINGE uses a hybrid in-guest/out-of-guest approach. SADE dynamically injects a small agent into the guest that can call internal guest functions, but does not protect the agent or the execution of guest code [6]. Overshadow uses virtualization to protect an in-guest application's confidentiality and integrity in the presence of a compromised OS, but does not protect the execution of guest OS code or the application's availability [9]. Srinivasan et al. mitigates the semantic gap when actively monitoring guest processes by moving them to the security VM while still maintaining its interactions with the guest OS [22].

The Virtuoso project shares SYRINGE's basic insight of leveraging the guest's code to minimize the semantic gap [15]. Virtuoso relies on pre-extracted execution traces of guest monitoring functions to automatically generate introspection programs that can be executed in the SVM. These traces must be extracted before any monitoring can be performed, and must be re-extracted whenever the guest OS is updated. Virtuoso also suffers from the fundamental incompleteness of dynamic analysis, which can create significant runtime hazards for the generated introspection programs. SYRINGE shepherds the guest's own internal execution, thus avoiding the hazards of execution trace replaying and the need for a recurrent learning phase.

A brief discussion of the technique underlying function-call injection was first presented by Joshi et al. [21], and a more basic variant was later proposed by SADE [6]. Program shepherding was first proposed by Kiriansky et al. to protect systems against application vulnerabilities [23]. It dynamically monitors the execution of control transfer instructions in the program to ensure that it does not deviate from a certain control-flow integrity policy. SYRINGE only requires indirect control transfers to be checked, and ensures atomic execution. Also, our shepherding is not done system or application-wide, but is localized to the monitoring thread and is activated/deactivated on-demand, minimizing the performance impact.

7 Conclusion

We proposed SYRINGE, a secure and robust infrastructure for monitoring virtual machines. SYRINGE removes the monitoring application from the guest and uses function-call injection to leverage the guest's own functions, thus avoiding the semantic gap problem. Security is achieved by verifying the execution of the guest code using localized shepherding. We implemented SYRINGE using the VMsafe introspection engine to monitor a guest OS running Windows XP. We evaluated its performance and security, showing that all simulated attacks were detected. Finally, we built and demonstrated a prototype application that uses SYRINGE to periodically obtain the list of loaded guest modules. Its evaluation showed that for a calling period of 1 second, the overhead imposed by SYRINGE on the system is 8%.

Acknowledgments. The authors would like to thank the anonymous reviewers for their comments. This work is supported in part by the National Science Foundation grant 0831300, the Department of Homeland Security contract FA8750-08-2-0141, the Office of Naval Research grants N000140710907 and 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, the Department of Homeland Security, or the Office of Naval Research. Part of this work was done while Martim Carbone was interning at Symantec.

References

1. Garfinkel, T., Rosenblum, M.: A virtual machine introspection based architecture for intrusion detection. In: Proceedings of the 2003 Network and Distributed System Symposium (2003)
2. Payne, B.D., Carbone, M., Lee, W.: Secure and flexible monitoring of virtual machines. In: Proceedings of the 23rd Annual Computer Security Applications Conference (ACSAC 2007) (2007)
3. Jiang, X., Wang, X., Xu, D.: Stealthy malware detection through vmm-based "out-of-the-box" semantic view reconstruction. In: Proceedings of the 14th ACM Conference on Computer and Communications Security (2007)
4. Petroni, N.L., Fraser, T., Walters, A., Arbaugh, W.A.: An architecture for specification-based detection of semantic integrity violations in kernel dynamic data. In: Proceedings of the 15th USENIX Security Symposium (2006)
5. Petroni, N.L., Hicks, M.: Automated detection of persistent kernel control-flow attacks. In: Proceedings of the 14th ACM Conference on Computer and Communications Security (2007)
6. Chiueh, T., Conover, M., Lu, M., Montague, B.: Stealthy deployment and execution of in-guest kernel agents. In: Blackhat Technical Security Conference (2009)

7. Payne, B.D., Carbone, M., Sharif, M., Lee, W.: Lares: An architecture for secure active monitoring using virtualization. In: Proceedings of the IEEE Symposium on Security and Privacy (2008)
8. Sharif, M., Lee, W., Cui, W., Lanzi, A.: Secure In-VM Monitoring Using Hardware Virtualization. In: Proceedings of the 16th ACM Conference on Computer and Communications Security (2009)
9. Chen, X., Garfinkel, T., Lewis, E.C., Subrahmanyam, P., Waldspurger, C.A., Boneh, D., Dwoskin, J., Ports, D.R.K.: Overshadow: A virtualization-based approach to retrofitting protection in commodity operating systems. In: Proceedings of Thirteenth International Conference on Architectural Support for Programming Languages and Operating Systems (2008)
10. Hoglund, G., Butler, J.: Rootkits: Subverting the Windows Kernel. Addison-Wesley Professional (2005)
11. Hund, R., Holz, T., Freiling, F.: Return-Oriented Rootkits: Bypassing Code Integrity Protection Mechanisms. In: Proceedings of the 18th USENIX Security Symposium (2009)
12. 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 (2007)
13. VMware Inc.: VMware VMsafe partner program, (December 2010), `http://www.vmware.com/technical-resources/security/vmsafe.html`
14. bugcheck, skape: Finding Ntoskrnl.exe Base Address. Uninformed 3 (2006)
15. Dolan-Gavitt, B., Leek, T., Zhivich, M., Giffin, J., Lee, W.: Virtuoso: Narrowing the semantic gap in virtual machine introspection. In: Proceedings of the IEEE Symposium on Security and Privacy (2011)
16. Bletsch, T., Jiang, X., Freeh, V.W., Liang, Z.: Jump-oriented programming: A new class of code-reuse attack. In: Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security (ASIACCS) (2011)
17. Checkoway, S., Davi, L., Dmitrienko, A., Sadeghi, A.-R., Shacham, H., Winandy, M.: Return-oriented programming without returns. In: Proceedings of the 17th ACM Conference on Computer and Communications Security (2010)
18. Carbone, M., Cui, W., Lu, L., Lee, W., Peinado, M., Jiang, X.: Mapping Kernel Objects to Enable Systematic Integrity Checking. In: Proceedings of the 16th ACM Conference on Computer and Communications Security (2009)
19. Jones, S.T., Arpaci-Dusseau, A.C., Arpaci-Dusseau, R.H.: Antfarm: Tracking processes in a virtual machine environment. In: Proceedings of the 2006 USENIX Annual Technical Conference, USENIX 2006 (2006)
20. Litty, L., Lagar-Cavilla, H.A., Lie, D.: Hypervisor support for identifying covertly executing binaries. In: Proceedings of the 17th USENIX Security Symposium (2008)
21. Joshi, A., King, S.T., Dunlap, G.W., Chen, P.M.: Detecting past and present intrusions through vulnerability-specific predicates. In: Proceedings of the 20th ACM Symposium on Operating Systems Principles (2005)
22. Srinivasan, D., Wang, Z., Jiang, X., Xu, D.: Process out-grafting: An efficient “out-of-vm” approach for fine-grained process execution monitoring. In: Proceedings of the 18th ACM Conference on Computer and Communications Security (2011)
23. Kiriansky, V., Bruening, D., Amarasinghe, S.: Secure Execution Via Program Shepherding. In: Proceedings of the 11th USENIX Security Symposium (2002)

Assessing the Trustworthiness of Drivers

Shengzhi Zhang and Peng Liu

The Penn State University, University Park, PA, USA
suz116@psu.edu, pliu@ist.psu.edu

Abstract. Drivers, especially third party drivers, could contain malicious code (e.g., logic bombs) or carefully designed-in vulnerabilities. Generally, it is extremely difficult for static analysis to identify these code and vulnerabilities. Without knowing the exact triggers that cause the execution/exploitation of these code/vulnerabilities, dynamic taint analysis cannot help either. In this paper, we propose a novel cross-brand comparison approach to assess the drivers in a honeypot or testing environment. Through hardware virtualization, we design and deploy diverse-drivers based replicas to compare the runtime behaviour of the drivers developed by different vendors. Whenever the malicious code is executed or vulnerability is exploited, our analysis can capture the evidence of malicious driver behaviour through comparison and difference telling. Evaluation shows that it can faithfully reveal various kernel integrity/confidentiality manipulation and resource starvation attacks launched by compromised drivers, thus to assess the trustworthiness of the evaluated drivers.

Keywords: Driver code safety, diversity, hardware virtualization.

1 Introduction

Drivers, especially third party drivers, could contain malicious code (e.g., logic bombs) and/or carefully designed-in vulnerabilities. Once got executed/exploited, such compromised drivers render the attackers the opportunity of leveraging drivers' privilege to manipulate system integrity and data confidentiality. Even worse, some attackers have successfully stolen certification from benign third-party and easily obtained trust from the most cautious system engineers. For instance, *mrxcls.sys*, a driver digitally signed with a compromised Realtek certificate, may be viewed as trusted and loaded into industrial OS by system engineers. Once loaded, it injects malware *Stuxnet* into the victim OS, which in turn causes catastrophe in Siemens supervisory control and data acquisition industrial systems [2].

Fully assessing third party drivers before running them in most commodity server systems is challenging. First, static analysis of such drivers is not always possible due to the unavailability of their source code. Furthermore, carefully designed-in vulnerability or malicious code triggered by some specific logic are extremely difficult to be pinpointed during static analysis. Second, dynamic taint

D. Balzarotti, S. Stolfo, and M. Cova (Eds.): RAID 2012, LNCS 7462, pp. 42–63, 2012.

analysis (e.g., [37] and [11]) of driver code is generally infeasible, due to the unknown taint seed during assessment. Without an accurate reference model, tainting the entire code space of drivers can only reveal drivers' behaviour, instead of distinguishing legitimate actions from malicious ones. Last, besides promiscuous attacks such as kernel integrity manipulation, some passive attacks, e.g., listening post, launched by compromised drivers are more difficult to be captured.

Previous research proposes to protect kernel integrity from drivers by confining the drivers' execution context, e.g., Nooks [30], Gateway [28], HUKO [35], Device Driver Reuse and Isolation [24], and Mondrix [34]. Although these systems can effectively monitor drivers' interaction with kernel functions or data, deploying such isolation approach to assess drivers would cause a large number of false positives or false negatives. For instance, both Gateway [28] and HUKO [35] rely on explicitly white-listed legitimate entry points for control transfer from drivers to OS kernel. However, such explicit and complete reference model is quite difficult to be established in practice. We observe that legitimate drivers indeed invoke kernel functions not defined in legitimate entry points occasionally, which results in false positives. Moreover, frequently invoking kernel APIs defined in legitimate entry points can also lead to Denial of Service attack due to resource starvation, which causes false negatives.

In this paper, we present a novel driver evaluation approach, Heter-device, to comprehensively assess drivers against an implicit and complete model before putting any trust on them. Heter-device relies on virtual platforms to emulate heterogeneous device (Heter-device) pairs (e.g., Intel 82540EM NIC and Realtek RTL8139) for guest operating system replicas. Each replica loads heterogeneous drivers corresponding to the devices it runs on. Heter-device approach stands on the assumption that heterogeneous drivers should not have the same exploitable vulnerability due to their separated developing processes. So they provide an implicit and complete reference model for each other when trustworthiness assessment is conducted via fine-grained auditing. Hence, by deploying Heter-device as a high-interaction honeypot, we can closely compare the divergence of two replicas when the vulnerable driver is being compromised and leveraged.

The two replicas with heterogeneous drivers are synchronized at the exported function entry points, which are declared by OS kernel and implemented by each driver. We start a fine-grained auditing of driver's execution whenever kernel calls the corresponding driver functions. During driver's execution, every jump, call or return to kernel or other kernel modules' address space are logged for verification. The logs from heterogeneous drivers are parsed and compared to check any suspicious control flow redirection, e.g., one driver jumps to a kernel segment written by itself, while the other does not exhibit such behaviour. Moreover, any modification to key kernel data by drivers is recorded and verified against the heterogeneous drivers to check if it is a legitimate modification or a malicious manipulation.

We also deal with passive attacks launched from compromised drivers, e.g., network card driver intercepts incoming/outgoing packets and redirects them to remote entities. Thus, the network outgoing packets of the two replicas are audited and compared to find mismatch. Additional amount of traffic on one replica against the other suffices an alarm of confidentiality compromise. Finally, abuse of kernel APIs, such as spin lock or kernel memory allocation requests, may cause CPU or memory starvation. Hence, any call to these resource request APIs from drivers is also verified against heterogeneous drivers. By placing the synchronization and monitoring "sensors" in Heter-device, our honeypot can faithfully reveal multiple attack vectors of compromised drivers, including kernel integrity manipulation, resource starvation, and confidentiality tampering.

We target a honeypot or testing environment; accordingly, we implement Heter-device framework based on open source QEMU [1] project for the following reasons. First, QEMU facilitates our Heter-device architecture by providing heterogeneous device emulation options for several types of devices, e.g., sound card (sound blaster 16 or Gravis ultrasound GF1), Ethernet network card (Intel 82540EM NIC or Realtek RTL8139), video card (Cirrus Logic GD5446 Video card or Standard VGA card with Bochs VBE extensions), and etc. Furthermore, it enables our fine-grained auditing of driver's execution through binary translation blocks. Specifically, since each jump of register *eip* generates a new translation block in QEMU, we can simply monitor *eip* at the beginning of each translation block to capture the driver's execution context, rather than auditing every instruction. Last, though the overhead of QEMU is significant, providing good performance is not so critical in either honeypot or testing environment.

Our evaluation shows that Heter-device is effective in revealing multiple attack vectors of compromised drivers, e.g., kernel APIs abuse, malicious code injection, key kernel data tampering, resource starvation, and sensitive information leakage. Accordingly, typical real world use of Heter-device can be as follows: the system engineers assess drivers using Heter-device first, and then choose the trustworthy drivers[1] to run their server systems. Compared to native QEMU execution, the performance overhead incurred by auditing control flow transition and synchronization can be optimized to range from 20 % to 90 %, depending on the amount of kernel data to be audited. Heter-device driver assessment only requires drivers' binary code to run in network-oriented testing environment (i.e., honeypot), and dose not involve any modification to driver source code, compilers, or targeted operating systems.

The rest of this paper is organized as follows. The next section overviews Heter-device threat model. Section 3 presents the design details of Heter-device approach, focusing on Heter-device architecture, address-alias correlation, runtime synchronization and multi-aspect auditing and verification. Section 4 summarizes the implementation issues of Heter-device. In Section 5, we evaluate Heter-device by case studies and measure its performance overhead. In Section 6, we discuss the limitation and future work of Heter-device. Finally, we present related work in Section 7 and conclude in Section 8.

[1] System engineers can either buy the corresponding real hardware devices or configure virtual platforms to emulate those devices.

2 Threat Model

In this paper, we assume that the device drivers are untrusted, either with vulnerabilities that can be exploited locally or remotely, or inherently malicious. Furthermore, we only focus on the exploitations that are carefully designed and crafted by attackers. Otherwise, crashing the target system will definitely draw system engineers' attention to trace such consequence back to the root cause. Last, as the base of Heter-device, we assume that heterogeneous drivers should have different vulnerabilities or different malicious code, in terms of where and what the vulnerabilities or malicious code are. Hence, at least one driver can serve as a criterion to verify and alarm the other compromised driver's execution.

It is generally believed to be challenging to verify the behaviour of untrusted drivers in an efficient and robust way due to at least the following reasons. First, drivers in most commodity OS have exactly the same privilege as kernel and run in the same address space as kernel. Thus, any kernel module performing auditing or monitoring tasks may be manipulated by the compromised driver. Second, the attackers may leverage the compromised driver to tamper arbitrary OS components (e.g., function pointers, file metadata, system call table, etc.), to accomplish their intrusion goals. Hence, it is also quite difficult, if not impossible, to pinpoint the comprehensive auditing points covering all possible damages/harms that could be caused by compromised drivers. Last, even if it is possible to censor drivers' execution efficiently and comprehensively, lacking a complete reference model makes the verification of drivers' behaviour challenging. Significant false positive or false negative is expected.

In this paper, we propose a novel approach, Heter-device, using driver-diversity-based replica as a complete and implicit reference model, to assess drivers in the following attack vectors:

Control Flow Manipulation. The control flow transition from driver to kernel is tampered by compromised drivers, e.g., jumping to a specific address in the middle of kernel functions, making suspicious kernel function calls to modify critical registers, and etc.

Key Kernel Data/Code Manipulation. Compromised drivers tamper with kernel code, static global variables, or key dynamic data specified by kernel developers or system engineers, e.g., system call table, interrupt descriptor table, double linked list pointers in process control block, and etc.

Confidentiality Manipulation. Compromised drivers intercept bypassing information or access sensitive files, and send them out through network to remote unknown entities. For instance, compromised NIC driver intercepts all the incoming/outgoing packets and redirects them to attackers' machine.

Resource Starvation. Compromised drivers abuse critical resources and incur denial of service, e.g., dominating CPU by locking interrupts or exhausting memory by endless allocation request.

Since we assume OS kernel is fully trusted, we don't verify the control flow transition from OS kernel to driver code, nor audit the driver's data accessed by OS kernel. Furthermore, the function parameters and stack data passed between OS kernel and drivers are not verified currently, which could be leveraged by compromised drivers to tamper kernel integrity in certain ways. For instance, when calling a certain kernel API, attackers can launch a return-oriented attack to jump to other kernel functions through carefully crafted parameters. Such attack can indeed evade the auditing of Heter-device, but we believe that currently it requires significant manual efforts of attackers[2]. Finally, since Heter-device relies on underneath virtual platform to emulate heterogeneous devices for guest OS replicas, we assume the virtual machine monitor is in the trusted computing base. Exploiting bugs in virtual machine monitor, such as [3], and then controlling the guest OS are not in the scope of this paper.

3 Heter-Device Design

In this section, we first describe the novel virtualized device diversity approach to efficiently produce driver-diversity-based OS replicas, then present Heter-device approach to evaluate drivers in multi-aspects.

3.1 Heter-Device Architecture

Figure 1 shows our Heter-device architecture with the **front stage replica** and the **back stage replica** running as Guest OS atop the same Host OS. The device diversity is produced by virtual platform to emulate heterogeneous devices for the two VM replicas. The virtual platforms can run unmodified commodity OS by giving the Guest OS the illusion that it runs on top of "real" hardware. Thus, the guest OS will load corresponding drivers for the hardware devices that it regards as "real". In this way, the virtualized device diversity approach gains the same security benefits as costly hardware diversity in a much cheaper manner. For instance, software diversity approach enables two replicas to run on two separated emulated platforms, one with Intel 82540EM NIC (Network Interface Card), sound blaster 16 (sound card), Universal Host Controller Interface (USB controller) and etc., the other with Realtek RTL8139 NIC, Gravis ultrasound GF1(sound card), Intel Open Host Controller Interface (USB controller) and etc.

Our virtualized device diversity idea is inspired by both the hardware-based diversity approach and the sweeping deployment of virtual platforms (e.g., VMware, Xen, KVM, QEMU and etc.) in the production server environment. In this paper, we call the diverse devices with different models but performing the same functionality, e.g., Intel 82540EM NIC and Realtek RTL8139 NIC, as a pair of heterogeneous devices. As a result of pairs of heterogeneous devices emulated by virtual platform, the guest OS kernel of each replica will load heterogeneous

[2] The most recent work [21] fully automates the instruction sequence construction that can be used by an attacker for malicious computations. However, the side-effect of the construction time (2009 ms) and the runtime overhead (135 times slower) will cause significant divergence on the logs of the two replicas, which will be caught by Heter-device as CPU resource abuse.

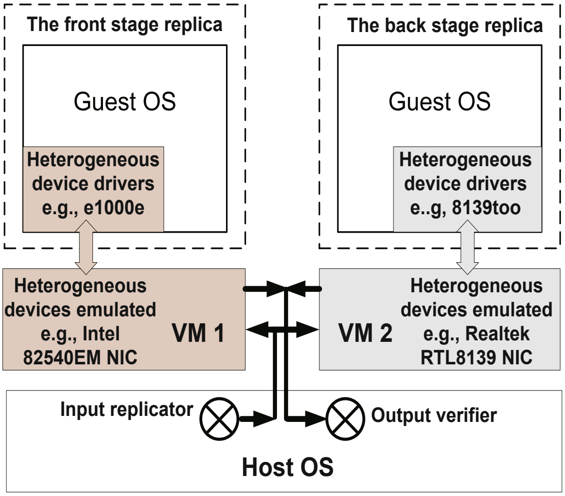

The front stage replica
Guest OS
Heterogeneous device drivers e.g., e1000e
The back stage replica
Guest OS
Heterogeneous device drivers e..g, 8139too
Heterogeneous devices emulated e.g., Intel 82540EM NIC
VM 1
VM 2
Heterogeneous devices emulated e.g., Realtek RTL8139 NIC
Input replicator
Output verifier
Host OS

Fig. 1. Heter-device Software-based Diversity Architecture for Driver Assessment

drivers correspondingly, e.g., e1000 or 8139too kernel modules[3]. Except heterogeneous drivers, the guest operating systems on the front stage VM replica and the back stage VM replica are exactly identical in terms of kernel version, installed applications and services, other loaded modules, start-up scripts, and etc.

The external input should be redirected to both the two replicas. We implement the input replication at the Host OS, totally transparent to the Guest OS replicas. Basically, every external input from network, keyboard, and mouse triggers both the device emulation modules of the two virtual machines. Since the input data from virtual disk is initialized by guest OS replicas, it does not need to be replicated. The output from the two guest OS replicas is intercepted and recorded by virtual machines. For instance, the network output traffic from each replica is audited and verified to capture any confidentiality tampering through network. During evaluation, we ensure that only the output from the front stage replica is sent out, while the output from the back stage replica is discarded, to guarantee the correctness of communication context.

3.2 Heter-Device Approach

There exist several challenges to assess drivers based on Heter-device architecture, so we abstractly present our system design to tackle these challenges in the following.

Address-Alias Correlation. Through pre-configuration, both the front stage and back stage OS replicas can load root symbols (defined in *System.map* in

[3] We focus our discussion on Linux operating system in this paper, but Heter-device is easily transported to other operating systems through reasonable efforts.

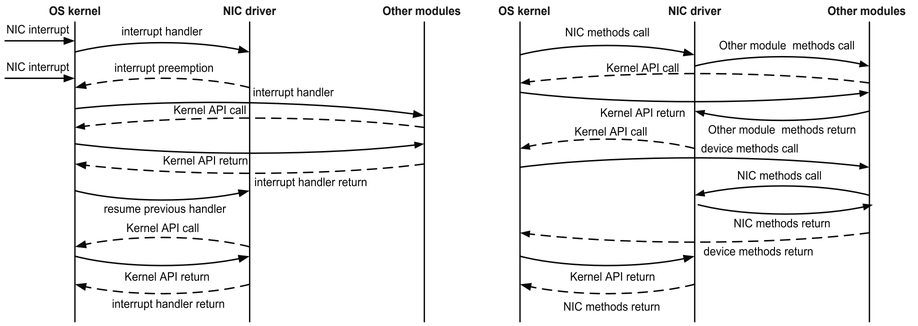
Linux) into the same memory address. However, other dynamic kernel data may be loaded into different addresses, even if the data represents exactly the same semantics. For instance, with the same kernel version and configuration, the two OS replicas store the process descriptors of their root processes in the same address (pointed by the root symbol *init_task*). By traversing the double linked list of all the processes on the two replicas, we observe exactly the same process list, including kernel threads. However, the memory addresses storing all the other process descriptors, except the root process descriptor, do not match. Such address-alias of the same kernel data on the two replicas is prevalent and poses challenge to our auditing of kernel function calls and key kernel data accessed by evaluated drivers.

To tackle this challenge, we propose to correlate the address-alias kernel semantics of the front stage and back stage replicas. First, we need to reconstruct kernel semantics from raw physical memory of each replica respectively. Due to the challenge of reconstructing dynamic kernel data, such issue catches researchers' continuous attention recently, e.g., [22], [7], [15], [25], [14] and etc.

Heter-device efficiently integrates both the out-of-VM and in-VM approaches to comprehensively reconstruct kernel semantics. All kernel exported function pointers can be referred to from root symbol definition (*System.map* file), which is identical for both the replicas through default configuration. Some key kernel data can also be referred to in a similar way, with additional effort of recursive identification of kernel data structures. Regarding dynamic data of both kernel and drivers, we insert a fully trusted kernel module into both the front and back stage replicas, which notifies underneath virtual platforms about the allocation/reclaim of kernel memory and loading/unloading of kernel modules. With the reconstructed semantics, address-alias correlation recursively maps the addresses of the same kernel semantics, either function pointers or kernel data structures. Hence, it makes possible the efficient auditing and verification of heterogeneous drivers' execution.

Runtime Synchronization. Running the front stage and back stage replicas at large may incur "out-of-band" comparison of heterogeneous drivers' execution on the two replicas. Though we delivered the replicated external input to the two replicas at the virtual machine monitor level simultaneously, the corresponding interrupt to CPU on each replica may not be "simultaneous". Thus the actual processing of the interrupt on the two replicas may still be "out-of-band". Researchers have proposed interrupt-redelivery approach for deterministic replay, e.g., [16], [36], and [38], which could be leveraged by Heter-device to apply the exact-replay-style synchronization. However, due to the heterogeneous driver diversity introduced by Heter-device, synchronizing such diverse replicas poses quite realistic challenges, such as different instruction execution sequences.

We observed that although the implementation of heterogeneous drivers is different, they offer the same function interfaces to OS kernel. Such layered design of most operating systems implies that OS kernel only needs to know how to invoke the device driver's methods, rather than to understand the detailed implementation of driver's methods. Figure 2 shows the interaction among NIC



(a) Interaction between OS Kernel and NIC Driver through Interrupt or Interrupt Preemption

(b) Interaction between OS Kernel and NIC Driver through NIC Driver Methods

Fig. 2. NIC Driver Interaction with OS Kernel and Other Kernel Modules

driver, OS kernel and other kernel modules. Besides function call returns, the control flow transition to NIC driver code must be through NIC interrupt handler or NIC driver function calls. For instance, NIC driver (Linux version) has totally 18 methods, with 8 fundamental (e.g., *open*, *stop*, *hard_start_xmit*, and etc.) and 10 optional (e.g., *poll*, *set_mac_address*, *change_mtu*, and etc.), indicating the operations that can be performed on this network card.

OS kernel declares the corresponding driver function pointers and initialize them during the loading of driver modules. As described in Figure 2, these driver functions are the only entry points for the control flow to transit from OS kernel or other kernel modules to this driver. Since OS kernel is fully trusted, it will not redirect control flow to arbitrary driver addresses except driver function call returns. More importantly, these entry points are identical for heterogeneous drivers despite different implementation details of the driver functions. Address-alias of the entry points on the two replicas can be resolved by correlating the addresses of them together. Thus, the two replicas can be synchronized by the entry points of the same device driver functions.

Hence, the auditing and verification of drivers' execution can be triggered by sensors monitoring the entry points. Specifically, when OS kernel's execution encounters an entry point, i.e., OS kernel calls a driver function, the context of current execution is recorded on the two replicas separately. Then, all the following instruction sequences of the two replicas are audited respectively, until the return to the previously logged context[4]. In particular, the entry point of the interrupt handler function deserves special attention, since nested interrupts (new interrupt comes during the processing of previous interrupt) may happen sometimes. Hence, each entry to driver's interrupt handler function is sequenced, and strictly matched to the corresponding return. In this way, the driver's execution can be identified apart from OS kernel's execution.

[4] Driver may also call kernel APIs during its execution. So the return to OS kernel address space does not suffice the end of driver's execution. Instead, only the return to the caller's execution context indicates the end.

Kernel Integrity Mediation. Compromised drivers can leverage the ultimate privilege to manipulate kernel integrity, e.g., hijacking control flow or tampering kernel data. Below, we present the approach of auditing such kernel integrity that could be tampered by compromised drivers based on Heter-device architecture.

Control Flow. For benign OS kernel and drivers, the control flow transitions among them can be well regulated by confining exported functions. For instance, Figure 2 shows that the control flow transition from OS kernel to drivers can be made by calling functions exported by drivers. Besides call return or hardware interrupt, the transition from benign drivers to OS kernel is through functions exported by OS kernel itself or other kernel modules which may call exported kernel APIs on behalf of the calling driver. Since we trust OS kernel, OS kernel only calls functions exported by drivers to transit the control flow. However, compromised drivers may directly jump to any address inside kernel or other modules to continue execution. Moreover, they can inject malicious code into OS kernel memory using DMA, and subvert function pointers on stack to the injected code, to hijack the control flow transition.

Existing researches, e.g., Gateway [28] and HUKO [35], prevent such control flow integrity manipulation by isolating address spaces of OS kernel and drivers. Legitimate entry points for execution transition from drivers to OS kernel are explicitly listed, e.g., system root symbols in Linux *System.map*). However, we observe that legitimate drivers may invoke some kernel APIs not defined in their legitimate entry point set. Rather, invoking kernel APIs frequently in their legitimate entry point set can also lead to denial of service attack through resource starvation. Furthermore, compromised drivers may inject malicious code into their own stack or heap to launch attack without violating control flow transition policies.

The runtime synchronization facilitates the fine-grained auditing of drivers' execution, from the call to driver's function to the corresponding return to caller. During the auditing of driver's execution, every jump or call out of driver's code address space is logged. These calls of kernel APIs or other kernel modules should be verified against the two replicas. Since the implementation of the heterogeneous drivers is different, strict verification of the sequence of their OS kernel API calls would always fail. However, to provide the same functionality, during assessment we observed a set of specific kernel APIs is frequently called by heterogeneous drivers. For instance, the kernel API calls made by NIC converge at irq locking/unlocking and memory allocation/deallocation.

We expect system engineers to manually analyze and verify the logged kernel API calls made by heterogeneous drivers within each specific driver function. Based on our experience, most system engineers (even those not quite familiar with OS kernel) have a sense of which set of kernel APIs are relevant in a specific driver function based on our cross-checking reference model. In addition, it is also relatively easy for them to capture some outlier kernel API calls through the pairwise comparison for further verification. For example, one volunteer system engineer at the first glance, pointed out that *rtl8139_open* makes a *kernel_thread* call, while *e1000_open* does not. Though such behaviour is finally verified as benign, we believe that such significant variance deserves further verification.

Besides the outlier kernel APIs, the kernel APIs that are called with an extremely high frequency deserve further verification as well. For instance, endlessly calling resource request kernel APIs will cause resource starvation as discussed in Section 3.2. Repeatedly calling *prepare_to_wait* kernel API will put all the runnable processes into sleep. Such malicious behaviour can be easily captured through the comparison of the amount of each kernel API calls within a specific driver function against heterogeneous drivers. Furthermore, benign drivers typically will only write data, rather than code, into their own stack/heap, or DMA-mapped kernel memory. Thus, any jump/call to the address within driver's stack/heap, or DMA-mapped kernel memory is strictly verified to check if such behaviour is general for the heterogeneous drivers to provide desired functionality. If not, further verification should be given to the driver that exhibited such suspicious behaviour.

Data Integrity. Drivers have the privilege to modify any kernel control-relevant or control-non-relevant data. Such modification by drivers should be strictly verified, rather than forbidden, since some of the modification might be legitimate to provide some desired functionality. For instance, previous Linux kernel does not export *set_current_state* API for drivers to change the state of a certain process. Instead, drivers have to directly set the state of current running process by *current->state = TASK_INTERRUPTIBLE*. However, compromised drivers may take advantage of those exported kernel APIs to hijack control flow, e.g., manipulating kernel control-relevant data, such as system call table, IDT, etc. A particular process can also be hidden by tampering kernel control-non-relevant data, such as pointers of double linked list processes.

Hence, we propose to verify the modification to key kernel data by heterogeneous drivers to capture any malicious manipulation. Beginning from the call to driver functions, copy-on-write is triggered on the memory storing the key kernel data on each replica, until the corresponding return to caller. Hence, the modification to key kernel data can be accounted to the corresponding drivers. However, driver's execution may be disrupted by a preempted interrupt, which is handled by OS kernel and corresponding interrupt handler. The modifications during the preempted interrupt handling should be accounted to the driver that implements the interrupt handler function. These modification logs of each driver function are verified against heterogeneous drivers for any malicious manipulation.

We observe that most kernel data integrity manipulation is accompanied with control flow hijacking, which can be identified as discussed in control flow manipulation. Regarding pure data integrity manipulation, kernel developers or security engineers can provide a list of critical kernel data based on empirical experiences or referring to kernel critical data profiling [32]. Generally, when the amount of key kernel data to be verified becomes large, the runtime overhead and the false positive of Heter-device verification will become significant. Hence, we propose to select a subset of kernel integrity critical data as the verification candidate, e.g., system call table, IDT, critical function pointers in process descriptor, double linked list pointers, and etc.

Confidentiality and Resource Consumption. Passive attacks, such as confidentiality tampering or resource abuse, are challenging to be identified or detected. Unfortunately, compromised drivers can leverage their ultimate privilege to maliciously intercept any data flow or repeatedly request any critical system resource, thus tampering confidentiality or crashing the system. Below, we discuss the methodology of relying on Heter-device architecture to capture such passive attacks during driver assessment, and present the framework we integrated into our approach.

Confidentiality. Compromised drivers can intercept bypassing data, call transmission (*hard_start_xmit* in Linux) function in network card driver, and send out to remote machines. Through control flow auditing and verification of heterogeneous drivers, the additional call to network card driver functions can be captured and identified as suspicious as discussed in Section 3.2. However, if network card driver itself is compromised, such data interception can be done totally within its own execution context, without any call to functions in other kernel modules. Although data flow auditing and verification of heterogeneous drivers can be added to capture the data interception, it would incur significant runtime overhead.

In this paper, we only focus on confidentiality leakage through network, that is, the intercepted data is transmitted to remote machines through network interface card. We assume that servers' running environment has strict physical access restrictions. Thus, it is out of our scope that the compromised drivers intercept the data and write it to local disk, which is then fetched through local access. Based on our assumption, we monitor and verify the network output of the front stage and back stage replicas for any confidentiality leakage. When deploying Heter-device architecture, OS kernel and service applications on the two replicas are identical, and the incoming packets to the emulated network cards are exactly replicated. So the output from the two replicas should be kept in rhythm unless anomaly happens. Hence, the output from the two replicas are matched with the combination of receiver's IP and packet sequence number. The additional traffic for information leakage from the compromised replica can be captured and alarmed.

Resource Consumption. Compromised drivers can launch various resource abuse attacks, and even cause denial of service due to resource starvation. Acquiring/releasing interrupt lock, allocating/freeing memory and etc., are benign operations for most drivers to provide desired functionality. However, such legitimate operations may be leveraged by compromised drivers to launch CPU or memory starvation attacks. Certainly, it is infeasible to restrict these kernel APIs from drivers, because benign drivers may not work or malfunction. Heter-device captures such resource abuse attack by strictly auditing and verifying the resource request kernel APIs issued by heterogeneous drivers. Although different implementation of heterogeneous drivers may cause variance in system resource consumption, we believe significant variance must indicate suspicious driver, at least inefficient implementation of the driver. System engineers can easily set up a threshold of such variance to alarm resource abuse. Based on our experience, such variance threshold can be set from 5 % to 15 % based on various context for reasonable false negative and false positive.

4 Implementation

In this section, we present the implementation of Heter-device framework. We begin with Heter-device architecture deployment based on QEMU open source project, followed by address-alias correlation for the heterogeneous drivers based replicas. At last, we describe the implementation of the fine-grained mediation of heterogeneous drivers' execution.

4.1 Heter-Device Deployment

Software Diversity Architecture. Instead of deploying the replicas on costly real heterogeneous devices platforms, we implement our Heter-device architecture using QEMU, with one virtual machine as the front stage replica, and the other one as the back stage replica. We configure the virtual machine (QEMU) to emulate heterogeneous devices for the two replicas, i.e., one with Realtek RTL8139 NIC and Gravis ultrasound GF1, the other with Intel 82540EM NIC and sound blaster 16. Although other heterogeneous devices options are also available, e.g., USB, video card and etc., we believe heterogeneous network cards and sound cards are sufficient to demonstrate the proof-of-concept of Heter-device.

The disk image file is replicated for the two replicas to ensure the same guest operating system (with kernel version 2.6.15), service applications, configurations, startup scripts, and etc. Moreover, the two replicas are configured with the same amount of memory and networking model. Hence, the only difference between the two replicas is heterogeneous drivers, which interact with underneath heterogeneous devices. During the assessment of heterogeneous drivers, Heter-device serves as "honeypot" to trigger either the inherently malicious drivers or remote exploitation to drivers' vulnerabilities.

Input Replication and Output Verification. To implement the external input replication to the two replicas, we insert a small piece of replication code into the host operating system kernel. Whenever there is any external input, i.e., keyboard, mouse, network packet, to the front stage replica, the inserted code on host OS kernel replicates the input and notifies both the two replicas for incoming events. Since the virtual machine we use (QEMU) behaves as a user process on host OS, the notification can be done either by signal or bit masking based on the context. In contrast, the network output from each replica is logged by the emulated network card of each virtual machine. On host OS, we implement a verification process examining the logs from the two replicas. In particular, it extracts the destination IP, sequence number information from each packet and matches the corresponding packets from the two replicas. A threshold of the amount of unmatched packets can be pre-determined, to alarm any confidentiality leakage.

Synchronization of Replicas. We rely on both virtual machine and guest OS support to synchronize the front stage and back stage replicas at the granularity of driver function calls. We assume that host OS runs on multi-core hardware platform, thus each virtual machine is configured to run on a dedicated core for maximum CPU capacity. We craft a trusted kernel module to monitor the loading of heterogeneous drivers on each replica. For instance, it audits the procedure of initializing functions declared by OS kernel, e.g., the interrupt handler function and other functions registered by the driver. The memory addresses of these functions are sent to underneath virtual machine through a secure channel. Only the functions implemented by both heterogeneous drivers are selected as synchronization points for the two replicas. During runtime, the value of register *eip* is monitored on both replicas to capture the synchronization points, as discussed in Section 4.3.

4.2 Address-Alias Correlation

We focus our implementation on Linux OS and start from a set of root symbols in *System.map* file. Since operating systems on the front stage and back stage replicas are with the same kernel version and configuration, the symbols and their addresses in *System.map* are exactly the same. Then we apply a *CIL* module [27] on the source code of guest OS kernel to automatically extract type definitions of kernel data structures. Finally, beginning from each correlated root symbol, we recursively reconstruct memory semantics based on type definitions of kernel data structures, and correlate the same data structure with address-alias.

In particular, in order to correlate *task_struct* of each process, we start from the data structure *CPUState* defined by QEMU to emulate the processor for virtual machine. All the CPU registers can be referred to through the instance of *CPUState*: *env*. From the register *tr*, we locate the kernel stack of the currently running process. At the bottom of kernel stack resides the *thread_info* structure, which includes a pointer to the *task_struct* of the corresponding process. By traversing the double linked list processes through the *task* pointers on the two replicas, we can obtain all the process descriptors and correlate their addresses together.

4.3 Fine-Grained Driver Execution Mediation

QEMU is a binary translation based virtual machine, which facilitates fine-grained auditing of guest OS execution. Instead of instruction-by-instruction translation, QEMU implements translation block to improve performance. Specifically, QEMU generates host code from a piece of guest code without control flow redirection or static CPU state modification. Thus, for each translation block, guest OS executes without QEMU intervention unless interrupt occurs. At the end of each translation block, QEMU takes over the control and prepares for the next translation block.

The translation block mechanism provides a perfect mediation approach for drivers' execution. We can audit the program counter at the beginning of each translation block, which represents a control flow redirection, including *return*, *jump*, or *call*, etc. In this way, the entry into or the leave from driver's code section can be recorded efficiently without monitoring every executed instruction. However, QEMU also implements translation block chaining for performance boost. In particular, every time when a translation block returns, QEMU tries to chain it to previous block, thus saving the overhead of context switch to QEMU emulation manager. The translation block chaining indeed poses challenges for our control flow redirection mediation, since QEMU emulation manager may miss some redirections, i.e., some transitions between OS kernel and drivers.

In order to tackle this challenge, we trace down through the translation block chain whenever QEMU emulation manager begins a new translation block. QEMU defines *TranslationBlock* data structure for each translation block, where we can locate the program counters of this block and the next one along the chain. Hence, we can traverse the chain till the end to audit and record the program counter at the each control flow redirection. However, key kernel data cannot be recorded in this way since detailed execution context has not been established yet during the pre-traversing of translation block chain. In order to preserve the performance and key kernel data integrity, we mark the memory regions storing those key kernel data as non-writeable. Any attempt to write to the memory will be trapped to QEMU manager, and validated against the heterogeneous drivers. Currently, we assume that key kernel data always involves some static code, data, critical function pointers, and etc.

5 Evaluation

In this section, we present experimental results on Heter-device framework in three aspects. First, we present the comparison results on OS kernel APIs called by different functions of heterogeneous drivers. Second, we show the effectiveness of Heter-device in capturing compromised drivers by two case studies. Last, we evaluate the performance overhead incurred by Heter-device approach. The host OS is Ubuntu 10.10 with kernel version 2.6.35, and both of the two guest operating systems are installed with Fedora 5 (kernel version 2.6.15). We choose qemu-0.12.5 as the virtual machine monitor emulating two virtual platforms: one with Realtek RTL8139 NIC and Gravis ultrasound GF1, the other with Intel 82540EM NIC and sound blaster 16.

5.1 Profiling Heterogeneous Drivers

First, we load Heterogeneous NIC drivers *e1000* and *8139too* on the two replicas running Intel 82540EM NIC and Realtek RTL8139 NIC respectively. Our trusted kernel module monitors *alloc_netdev* function to trace the newly allocated *net_device* structure for the network card. Then the function pointers in *net_device*, such as *open*, *stop*, *hard_start_xmit*, etc., are audited during the initialization of NIC drivers to obtain the addresses of these driver functions. The functions implemented by both heterogeneous drivers are correlated as the synchronization entries of the two replicas. We start to audit the control flow transition between OS kernel and NIC driver since the booting of the two replicas.

Then we trigger a set of user commands (e.g., ssh, sftp, ping, and etc.) and applications (e.g., Firefox, Filezilla, and etc.), which involve network card operations, to invoke the interaction between OS kernel and NIC driver. Simultaneously, the kernel API calls issued by each synchronized function of heterogeneous drivers are profiled.

Table 1 shows our profiling results of heterogeneous drivers *e1000* and *8139too*. Although implemented by different teams, the same functions of heterogeneous drivers typically invoke a similar set of kernel APIs. In particular, we find that *8139too* calls *kernel_thread* kernel API in *open* function. Thus, we monitor the forked kernel thread and observe the following kernel APIs invoked during the thread's lifetime: *daemonize*, *allow_signal*, *interruptible_sleep_on_timeout*, *refrigerator*, *flush_signal*, *rtnl_lock_interruptible*, *rtnl_unlock*, and *complete_and_exit*. Table 1 also indicates that previous works will generate lots of false positive when referring to exported functions in *System.map* as trusted entries from drivers to OS kernel[5].

5.2 Case Study 1: Kernel Integrity Manipulation

We refer to the implementation of *adore-ng* kernel rootkit, and integrate its malicious code into the function *snd_gf1_stop_voice* of *gus* (for Gravis ultrasound GF1) driver. When users try to turn off the audio, the injected code gets executed to replace the functions of *readdir*, *lookup*, and *get_info* with its own implementation to hide files, processes and ports. The newly generated driver *gus* is recompiled and loaded into OS kernel. In contrast, driver *sb16* (for sound blaster 16) remains unchanged.

During the assessment of drivers *gus* and *sb16*, we simulate user's command to turn off the audio, which is replicated to both replicas. The modification of those static kernel data (function pointers) by driver *gus* is observed and alarmed, while driver *sb16* does not exhibit such behaviour. Then we clear this alarm, let the two replicas run forward, and issue process and file listing commands. We observe that the control flow transition from OS kernel to driver *gus* code section through unrecognised entry. Afterwards, driver *gus* calls kernel APIs, i.e., *readdir*, *lookup*, and *get_info*, from its execution context. In contrast, driver *sb16* on the other replica is not involved in the process and file listing procedures.

5.3 Case Study 2: Resource Abuse and Confidentiality Tampering

With the kernel privilege of compromised driver, attackers can launch resource starvation attack to reduce the productivity of the victim systems, or tamper confidentiality by intercepting bypassing data. We simulate resource abuse by inserting malicious code into the source code of RTL8139 NIC driver. In particular, after *spin_lock* is called in function *rtl8139_interrupt*, repeated call of

[5] Similar profiling has been performed on heterogeneous sound card drivers *gus* (for Gravis ultrasound GF1) and *sb16* (for sound blaster 16). The profiling results are excluded due to page restriction.

Table 1. Kernel APIs called by different functions in *e1000* and *8139too*. For each synchronization function, the upper box contains the invoked kernel APIs defined in *System.map* file of guest OS, while the lower box includes the indirected invoked kernel APIs that are called by drivers through the following procedure. Drivers call some other *extern* kernel functions (not defined in *System.map* file) by including some *.h* files, and these functions in turn invoke the indirected kernel APIs identified by us.

Synch. Entry	Kernel APIs by e1000	Kernel APIs by 8139too
*open	request_irq, mod_timer, kmalloc, pci_clear_mwi, vmalloc_node	netif_carrier_on, netif_carrier_off, request_irq, spin_unlock_irqrestore, spin_lock_irqsave, kernel_thread
	dma_alloc_coherent, __alloc_skb, __aloc_pages	dma_alloc_coherent
*stop	free_irq, netif_carrier_off, mmset, vfree, kfree	free_irq, wait_for_completion, kill_proc, spin_lock_irqsave, spin_unlock_irqrestore
	netpoll_trap, dma_free_coherent, lock_timer_base, list_del, __kfree_skb, local_irq_save, local_irq_restore	netpoll_trap, dma_free_coherent
*interrupt_handler	spin_lock, spin_unlock, eth_type_trans, netif_rx	spin_lock, spin_unlock
	__alloc_skb, netpoll_trap, kree_skb local_irq_save, local_irq_restore	netpoll_trap, local_irq_save, local_irq_restore
*tx_timeout	schedule_work	spin_lock, spin_lock_irqsave, spin_unlock, spin_unlock_irqrestore
	spin_lock, spin_lock_irqsave, __wake_up	
*do_ioctl	request_irq, spin_unlock_irqrestore free_irq, spin_lock_irqsave, netif_carrier_off, mod_timer	spin_lock_irq, spin_unlock_irq
	lock_timer_base, list_del, __kfree_skb, local_irq_save, local_irq_restore netpoll_trap	capable
*hard_start_xmit	spin_trylock, spin_unlock_irqrestore, pskb_pull_tail, pskb_expand_head	spin_lock_irq, spin_unlock_irq
	local_irq_save, netpoll_trap, local_irq_restore	__kfree_skb, netpoll_trap
*poll	spin_lock, spin_unlock, disable_irq enable_irq, netif_carrier_ok	spin_lock, spin_unlock, netif_receive_skb
	local_irq_save, __kfree_skb, local_irq_restore	local_irq_disable, __alloc_skb local_irq_enable, list_del, local_irq_save, local_irq_restore
*set_multicast_list		spin_lock_irqsave, spin_unlock_irqrestore
*get_stats	spin_lock_irqsave, spin_unlock_irqrestore	spin_lock_irqsave, spin_unlock_irqrestore

Table 2. Runtime Performance of Different Benchmarks

Benchmark	Key Kernel Data	Whole Kernel
LMbench	*1.2021*	*1.2444*
Apache Benchmark	*1.4420*	*2.8273*
Interbench	*1.3356*	*1.4160*
Kernel Decompression	*1.1663*	*1.2262*

alloc_skb is issued until kernel memory is overwhelmed. Then the driver is recompiled and loaded into OS kernel as *8139too* module. We repeat a subset of the user commands and applications in Section 5.1. During the assessment, after the synchronization of *interrupt_handler* function entry, *e1000_intr* quickly returns. However, *rtl8139_interrupt* continues running with lots of *alloc_skb* calls recorded. Our verification alarms such anomaly immediately with a pre-defined difference threshold (200 in our experiment) reached.

Furthermore, we also simulate confidentiality tampering attack by injecting malicious code into the packet transmission function *e1000_xmit_frame* of e1000 NIC driver. The newly compiled *e1000* module will intercept all the outgoing packets and redirect them to a remote machine. During the assessment, we replicate *Apache http* servers on both the two replicas, and simulate continuous client requests to them on another machine. The verification on the front-tier proxy matches the output packets from the two replicas. An alarm is signalled when the amount of unmatched packets from the replica with *e1000* module reaches the pre-defined threshold (20 in our experiment) in two minutes.

5.4 Performance Evaluation

The runtime overhead of Heter-device highly depends on the amount of key kernel data that needs to be verified. Table 2 shows the performance (the ratio of Heter-device execution and QEMU native execution) of Heter-device architecture based on several benchmarks. By key kernel data protection, we only verify static key kernel data, including system call table, IDT, root symbols in *System.map* files. In contrast, by whole kernel protection, the entire kernel address space is verified by Heter-device during driver assessment. During each round of evaluation, both the heterogeneous NIC and sound card drivers are verified for fine-grained control flow transition.

We use *LMbench* to evaluate the pipe bandwidth, and also evaluate the time consumed to decompress Linux kernel *3.0* as shown in Table 2. Since both of them involve little interaction with either NIC or sound card, the pipe bandwidth and CPU capacity are mostly retained. We run *Apache Benchmark* to evaluate the network performance, and *Interbench* to evaluate the audio performance. Table 2 demonstrates that network throughput drops more significantly than audio performance. We think the main reason is that NIC drivers interact more frequently with OS kernel during packet transmission than sound card drivers do during audio playing.

6 Discussion and Future Work

In this section, we discuss limitations and future work of Heter-device. First, although Heter-device architecture is totally compatible with most existing fault tolerant systems, deploying Heter-device approach as a real-time compromised driver detection system requires performance boost. Due to performance overhead (largely incurred by QEMU), Heter-device is currently deployed as honeypot to assess drivers before they are put into use in server systems. Hence, it should be our future work to facilitate hardware support, such as Intel VT or EPT techniques into our Heter-device architecture to feasibly detect compromised drivers in responsive server environment.

Second, currently QEMU supports limited number of emulated devices. Most existing device emulation modules in virtual machines such as virtualbox, Xen and KVM are all based on QEMU. Hence, assessing other drivers, e.g., keyboard, mouse and etc., is impossible right now on Heter-device architecture. Aware of our design, attackers can craft malicious drivers only for those devices that have not been emulated by QEMU. We suggest that system engineers consider the devices that can be emulated by QEMU right now, thus facilitating the driver assessment of Heter-device architecture. As the renaissance of virtualization, we hope that such device emulation options will bloom in the near future, making Heter-device more general and practical.

Third, there exist some counter-attacks to Heter-device architecture. As a honeypot or testing approach, Heter-device cannot be claimed to be able to capture any malicious code or vulnerability of drivers. There is always the possibility that the malicious code is not detected because the environment or the workload did not trigger it. Furthermore, transparently translating instructions [13] and faithfully emulating hardware devices are challenging tasks. For instance, QEMU can be detected by various methods as discussed in [20]. Aware of our design, attackers can craft malicious code that first examines whether it runs on emulated platforms. If so, the malicious code will "keep silent" to avoid being detected or profiled. Otherwise, it will compromise the victim system. Hence, the compromised drivers with "split-personality" can generally evade the auditing and verification of Heter-device.

Last, existing Heter-device approach involves manual intervention during the driver assessment. For instance, key kernel data to be recorded and verified should be provided by system engineers in advance, though we also offer a candidate list. Moreover, the verification procedure (i.e., control flow transition verification) requires system engineers to investigate the variance to reduce false positive. Furthermore, such manual inspection can also help to determine which driver is compromised, since the two replicas serve as reference model for each other rather than always treating one as golden standard. Our future work is set to comprehensively profile the driver's behaviour, thus improving the automation by providing more general key kernel data verification list and enforcing more detailed verification policies.

7 Related Work

In this section, we will first briefly review the state-of-art diversity techniques, and then discuss prior approaches protecting kernel integrity or reliability from driver faults or bugs.

Diversity Approach. Software diversity approach for intrusion detection has been studied in several works, such as COTS [31], Behavioural Distance [19], Diversified Process Replica [9], Detection of Split Personalities [8] and DRASP [39]. COTS Diversity [31] and DRASP[39] applies N-version programming into web servers to verify their interactions with the environment for any anomaly, e.g., HTTP responses from those web servers. Generally, [8] and [39] are ideal to detect anomaly targeting web servers, especially for most promiscuous attacks. But for other attacks, such as denial of service attack, or resource abuse, comparing network packets cannot work, since such attacks does not involve additional network packet transmission. On the other hand, comparing network packets is not always possible. For instance, if the payload is encrypted through IPsec, comparing the payload is meaningless.

Behavioural Distance [19] and Detection of Split Personalities [8] aim to detect intrusion or anomaly by comparing the system call sequences made by diverse applications. Diversified Process Replica [9] proposes non-overlapping processes address spaces to defeat memory error exploits. Although all the above approaches are effective in detecting compromised applications, the response or system call comparison schemes cannot be applied to diverse drivers as in Heter-device.

The seminal work N-variant [12] proposes address space partition and instruction set tag diversities to detect divergences caused by intrusions. Although such approaches are quite effective in detecting code injection related attacks, other types of exploitation, such as direct kernel object manipulation, kernel APIs abuse and confidentiality tampering can evade N-variant's auditing. Heter-device proposes heterogeneous device based diversity design, and specially focuses on the auditing and verification of drivers to cover multiple attack vectors that could be leveraged by compromised drivers.

Isolation Based Protection. Isolation-based approach continues drawing researchers' attention to protect OS kernel from buggy drivers for years. Nooks ([30] and [29]) pioneers the driver isolation approach to protect OS reliability from driver failures. Mondrix [34] integrates hardware support to isolate kernel modules by memory protection domains. Virtual machine technique has also been applied to isolate OS from buggy drivers. For instance, [24] and [17] isolate drivers by running a subset of untrusted drivers in a separated OS/VM domain, thus achieving both driver reuse and isolation.

Moreover, efficient address space isolation approaches have been proposed to protect kernel integrity [35] or monitor kernel APIs issued by untrusted kernel extensions [28]. Neither HUKO [35] and Gateway [28] is not comprehensive to assess drivers, because it doesnt cover kernel data integrity, confidentiality manipulation and resource starvation attacks as Heter-device. Instead of address space isolation, Heter-device audits drivers' execution at a finer granularity:

instruction sequences. Furthermore, besides kernel control flow or data integrity protection which are the primary focus of previous approaches, Heter-device also proposes feasible approach to capture confidentiality tampering and resource abuse attacks launched by malicious drivers.

User Mode Driver Based Protection. User mode device driver framework has been proposed recently to de-grant driver code's privilege, thus ensuring the kernel's integrity ([4], [5], [23]). However, they either suffer from significant performance degradation ([6], [26]), or require complete rewriting of driver code and modifications to OS kernel ([5], [23]). Concerning the performance issue, Microdrivers [18] present a novel approach to split driver code into both user mode and kernel mode execution, with only performance-sensitive code remaining in the kernel. Based on Microdriver, RPC monitor [10] protects kernel from vulnerable driver by mediating all data transfers from untrusted user-mode execution to kernel-mode execution to preserve kernel integrity. In addition, the reference validation mechanism can be integrated into Microkernels (e.g., Nexus [33]) to effectively audit driver's behaviour against safety specification. In general, Heter-device requires no rewriting of existing drivers, and is applicable to most commodity operating systems.

8 Conclusion

In this paper, we present a novel diversity based honeypot, Heter-device, to assess the trustworthiness of drivers from multiple aspects, including kernel integrity manipulation, resource starvation and confidentiality tampering. Heter-device relies on virtual platforms to emulate heterogeneous devices for guest operating systems, and correspondingly produce driver-diverse replicas. The diverse replicas are deployed as honeypot to audit and verify the heterogeneous drivers' execution by placing synchronization and monitoring "sensors". We also propose automatic address-alias correlation, a subset of kernel data for default integrity protection, and a set of policies to defeat resource abuse and confidentiality tampering. The case studies show that Heter-device can capture various kernel integrity manipulation, resource starvation, and confidentiality tampering launched from compromised drivers, thus delivering the trustworthy drivers after assessment.

Acknowledgment. This work was supported by AFOSR FA9550-07-1-0527 (MURI), ARO W911NF-09-1-0525 (MURI), NSF CNS-0905131, NSF CNS-0916469, and ARO W911NF1210055.

References

1. QEMU, open source processor emulator, http://wiki.qemu.org/Main_Page
2. Stuxnet, http://en.wikipedia.org/wiki/Stuxnet

3. Vulnerability Summary for CVE-2008-1943: Buffer overflow in the backend of XenSource Xen Para Virtualized Frame Buffer, http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2008-1943
4. Windriver cross platform device driver development, Technical report, Jungo Corporation (2002), http://www.jungo.com/windriver.html
5. Architecture of the user-mode driver framework, Version 1.0, Microsoft (2007)
6. Francois, A.: Give a process to your drivers. EurOpen (1991)
7. Arati, B., Vinod, G., Liviu, I.: Automatic Inference and Enforcement of Kernel Data Structure Invariants. In: 24th ACSAC (2008)
8. Davide, B., Marco, C., Christoph, K., Christopher, K., Engin, K., Giovanni, V.: Efficient Detection of Split Personalities in Malware. NDSS (2010)
9. Danilo, B., Lorenzo, C., Andrea, L.: Diversified Process Replicae for Defeating Memory Error Exploits. In: IEEE International Performance, Computing, and Communications Conference (2007)
10. Shakeel, B., Vinod, G., Michael, M.S., Chih-Cheng, C.: Protecting Commodity Operating System Kernels from Vulnerable Device Drivers. In: ACSAC (2009)
11. Jim, C., Ben, P., Tal, G., Kevin, C., Mendel, R.: Understanding data lifetime via whole system simulation. In: USENIX Security Symposium (2004)
12. Benjamin, C., David, E., Adrian, F., Jonathan, R., Wei, H., Jack, D., John, K., Anh, N., Jason, H.: N-variant systems: A secretless framework for security through diversity. In: USENIX Security Symposium (2006)
13. Artem, D., Paul, R., Monirul, S., Wenke, L.: Ether: malware analysis via hardware virtualization extensions. In: 15th ACM CCS (2008)
14. Brendan, D., Tim, L., Michael, Z., Jonathon, G., Wenke, L.: Virtuoso: Narrowing the Semantic Gap in Virtual Machine Introspection. In: IEEE Security and Privacy Symposium (2011)
15. Brendan, D., Abhinav, S., Patrick, T., Jonathon, G.: Robust signatures for kernel data structures. In: 16th ACM CCS (2009)
16. George, W.D., Samuel, T.K., Sukru, C., Murtaza, A.B., Peter, M.C.: Revirt: enabling intrusion analysis through virtual-machine logging and replay. In: OSDI (2002)
17. Ulfar, E., Tom, R., Ted, W.: Virtual Environments for Unreliable Extensions. Technical Report MSR-TR-2005-82, Microsoft Research (2005)
18. Vinod, G., Matthew, J.R., Arini, B., Michael, M.S., Somesh, J.: The design and implementation of microdrivers. In: 13th ASPLOS (2008)
19. Gao, D., Reiter, M.K., Song, D.: Behavioral Distance for Intrusion Detection. In: Valdes, A., Zamboni, D. (eds.) RAID 2005. LNCS, vol. 3858, pp. 63–81. Springer, Heidelberg (2006)
20. Tal, G., Keith, A., Andrew, W., Jason, F.: Compatibility is not transparency: VMM detection myths and realities. In: 11th USENIX HotOS (2007)
21. Ralf, H., Thorsten, H., Felix, C.F.: Return oriented rootkits: Bypassing kernel code integrity protection mechanisms. In: USENIX Security Symposium (2009)
22. Xuxian, J., Xinyuan, W., Dongyan, X.: Stealthy Malware Detection Through VMM-Based 'Out-of-the-Box' Semantic View Reconstruction. In: 14th ACM CCS (2007)
23. Ben, L., Peter, C., Nicholas, F., Stefan, G., Charles, G., Luke, M., Daniel, P., Yueting, S., Kevin, E., Gernot, H.: User-level device drivers: Achieved performance. Journal of Computer Science and Technology 5, 654–664 (2005)
24. Joshua, L., Volkmar, U., Jan, S., Stefan, G.: Unmodified device driver reuse and improved system dependability via virtual machines. In: 6th OSDI (2004)

25. Zhiqiang, L., Junghwan, R., Xiangyu, Z., Dongyan, X., Xuxian, J.: SigGraph: Brute Force Scanning of Kernel Data Structure Instances Using Graph-based Signatures. In: 18th NDSS (2011)
26. Kevin, T., Van, M.: The Fluke device driver framework. Master's thesis, University of Utah (1999)
27. George, C.N., Scott, M., Shree, P.R., Westley, W.: CIL: Intermediate language and tools for analysis and transformation of C programs. In: International Conference on Compiler Construction (2002)
28. Abhinav, S., Jonathon, G.: Efficient Monitoring of Untrusted Kernel-Mode Executio. In: 18th NDSS (2011)
29. Michael, M.S., Muthukaruppan, A., Brian, N.B., Henry, M.L.: Recovering Device Drivers. In: 6th OSDI (2004)
30. Michael, M.S., Brian, N.B., Henry, M.L.: Improving the reliability of commodity operating systems. In: 19th SOSP (2003)
31. Totel, E., Majorczyk, F., Mé, L.: COTS Diversity Based Intrusion Detection and Application to Web Servers. In: Valdes, A., Zamboni, D. (eds.) RAID 2005. LNCS, vol. 3858, pp. 43–62. Springer, Heidelberg (2006)
32. Wang, Z., Jiang, X., Cui, W., Wang, X.: Countering Persistent Kernel Rootkits through Systematic Hook Discovery. In: Lippmann, R., Kirda, E., Trachtenberg, A. (eds.) RAID 2008. LNCS, vol. 5230, pp. 21–38. Springer, Heidelberg (2008)
33. Dan, W., Patrick, R., Kevin, W., Emin Gn, S., Fred, B.S.: Device Driver Safety Through a Reference Validation Mechanism. In: 8th OSDI (2008)
34. Emmett, W., Krste, A.: Memory isolation for Linux using Mondriaan memory protection. In: 12th SOSP (2005)
35. Xi, X., Donghai, T., Peng, L.: Practical Protection of Kernel Integrity for Commodity OS from Untrusted Extensions. In: 18th NDSS (2011)
36. Min, X., Vyacheslav, M., Jeffrey, S., Ganesh, V., Boris, W.: Retrace: Collecting execution trace with virtual machine deterministic replay. In: 3rd MoBS (2007)
37. Heng, Y., Dawn, S., Manuel, E., Christopher, K., Engin, K.: Panorama: capturing system-wide information flow for malware detection and analysis. In: 14th ACM CCS (2007)
38. Shengzhi, Z., Xiaoqi, J., Peng, L., Jiwu, J.: Cross-Layer Comprehensive Intrusion Harm Analysis for Production Workload Server Systems. In: 26th ACSAC (2010)
39. Shengzhi, Z., Peng, L.: Letting Applications Operate through Attacks Launched from Compromised Drivers. In: AsiaCCS (2012)

Industrial Espionage and Targeted Attacks: Understanding the Characteristics of an Escalating Threat

Olivier Thonnard[1], Leyla Bilge[1],
Gavin O'Gorman[2], Seán Kiernan[2], and Martin Lee[3]

[1] Symantec Research Labs, Sophia Antipolis, France
{Olivier_Thonnard,Leylya_Yumer}@symantec.com
[2] Symantec Security Response, Ballycoolin Business Park, Dublin, Ireland
{Gavin_OGorman,Sean_Kiernan}@symantec.com
[3] Symantec.cloud, Gloucester, UK
Martin_Lee@symantec.com

Abstract. Recent high-profile attacks against governments and large industry demonstrate that malware can be used for effective industrial espionage. Most previous incident reports have focused on describing the anatomy of specific incidents and data breaches. In this paper, we provide an in-depth analysis of a large corpus of targeted attacks identified by Symantec during the year 2011. Using advanced TRIAGE data analytics, we are able to attribute series of targeted attacks to attack campaigns quite likely performed by the same individuals. By analyzing the characteristics and dynamics of those campaigns, we provide new insights into the modus operandi of attackers involved in those campaigns. Finally, we evaluate the prevalence and sophistication level of those targeted attacks by analyzing the malicious attachments used as droppers. While a majority of the observed attacks rely mostly on social engineering, have a low level of malware sophistication and use little obfuscation, our malware analysis also shows that at least eight attack campaigns started about two weeks before the disclosure date of the exploited vulnerabilities, and therefore were probably using zero-day attacks at that time.

1 Introduction

In 2010, Stuxnet [8] and Hydraq [16] demonstrated dangers the security community had long anticipated – that malware could be used for cyber-terrorism, real-world destruction and *industrial espionage*. Several other long term attacks against the petroleum industry, non-governmental organizations and the chemical industry were also documented in 2011 [3]. Such targeted attacks can be extremely difficult to defend against and those high-profile attacks are presumably just the tip of the iceberg, with many more hiding beneath the surface.

While targeted attacks are still rare occurrences today compared to classical, profit-oriented malware attacks, successful targeted attacks can be extremely damaging. One of the recent high profile targeted attacks against RSA has reportedly cost the breached organisation $66 million in direct costs alone [10,22]. Preventing such attacks from

D. Balzarotti, S. Stolfo, and M. Cova (Eds.): RAID 2012, LNCS 7462, pp. 64–85, 2012.

breaching organisations and causing subsequent harm depends on a detailed understanding of the threat and how attackers operate [18,2,17].

To understand the nature of targeted attacks, Symantec collected data on over 26,000 attacks that were identified as targeted during 2011. These attacks were based on emails which contained a malicious payload. Using advanced data analytics based on multi-criteria clustering and data fusion, we were able to identify distinct targeted attack *campaigns* as well as define characteristics and dynamics of these campaigns. Our research clearly demonstrates that a targeted attack is rarely a "single attack", but instead attackers are often quite determined and patient. A targeted attack is rarely an extremely stealthy, tedious and manual attack limited to a very small number of targets. A certain level of automation seems to be used by attackers and thus the notion of "campaigns" exist, yet of a very different amplitude than other malicious, non-targeted activities performed on a much larger-scale. We found also that these targeted attack campaigns can either focus on a single (type of) organization or they can target several organizations but with a common goal in mind. We refer to the latter ones as MOTA, for *Massive Organizationally Targeted Attacks*, and demonstrate their existence by means of some real world data we have analyzed.

A common belief with targeted attacks is that only large corporations, governments and Defense industries, and more particularly senior executives and subject matter experts, are being targeted by such attacks. Our research has shown that, at least for our set of targeted attacks collected in 2011, this was true only for 50% of the attacks. Moreover, while the ultimate goal of attackers is more than often to capture the knowledge and intellectual property (IP) that senior-level employees have access to, they do not have to attack them directly to steal the information they want.

The contributions of this paper are twofold. First, we focus on studying the characteristics of a comprehensive set of targeted attacks sent via email and collected in the wild by Symantec during the year 2011. More particularly, we show how those attacks are being organized into long-running campaigns that are likely run by the same individuals and we provide further insights into their *modus operandi*.

Secondly, we evaluate the *prevalence* and *sophistication* level of those targeted attacks by analyzing more in-depth the malicious attachments used as *droppers*. While a majority of the observed attacks rely mainly on social engineering, have a low level of malware sophistication and use little obfuscation, our analysis also shows that, in at least eight campaigns, attackers launched their attacks about two weeks before the disclosure date of the targeted vulnerabilities, and therefore were using zero-day attacks at that time.

The structure of this paper is organized as follows. In Section 2, we start by defining a targeted attack, describe its profile and common traits, and explain how we identified the set of targeted attacks used for this analysis. Section 3 describes in more details our experimental dataset and the attack features extracted from the emails. Then, in Section 4 we describe how we identified attack campaigns and provide insights into the way these campaigns are being orchestrated by attackers. Finally, in Section 5 we evaluate the prevalence and sophistication level of the malware used as dropper in the targeted attacks involved in those campaigns. Section 6 concludes the paper.

2 Targeted Attacks: Definition and Common Traits

2.1 Profile of a Targeted Attack

The vast majority of non-targeted malware attacks do not exhibit evidence of selection of recipients of the attack. In these cases, it appears as if the attacker wishes to compromise a number of systems without regard to the identity of the systems. Presumably the attacker believes that some of the systems may contain information that can be sold on, or that the compromised systems may be monetised by other means.

In targeted attacks there is evidence that the attacker has specifically selected the recipients of the attack. It may be that the attacker suspects that the attacked individuals have access to high value information which the attacker wishes to compromise, or the compromised systems can be used to launch attacks against other high value systems or individuals. Another distinguishing feature of targeted attacks is that the malware is distinct from that used in non-targeted attacks, and usually exhibits a higher degree of sophistication.

The data provided by *Symantec.cloud*[1] for this analysis only relates to targeted attacks where the malware is contained as an attachment to an email. It would be naive to expect that this is the only attack vector used by attackers. There may be other types of malicious activities, such as hacking attacks, that are conducted against the individuals and organisations who receive email targeted attacks as part of the same campaign. Targeted attacks themselves can take many forms. "Spear phishing" is a subset of targeted attacks where malicious emails are sent to targeted individuals to trick them into disclosing personal information or credentials to an attacker. Well designed attacks sent to a handful of individuals that include information relevant to the professional or personal interests of the victim may be particularly difficult for targets to identify as malicious [6]. However, such attacks are beyond the scope of our dataset used for this analysis.

The term "Advanced Persistent Threat" is often used in association with targeted attacks. This term is problematic since it is often used inconsistently within the security industry [2]. The National Institute of Standards and Technology, in part, defines the advanced persistent threat as "an adversary that possesses sophisticated levels of expertise and significant resources which allow it to create opportunities to achieve its objectives by using multiple attack vectors" [12]. On the other hand, others use the term to refer to "any attack that gets past your existing defences, goes undetected and continues to cause damage" [11].

Different researchers may choose the particular type of attack, the degree of sophistication and level of targeting use to define their own criteria for being a *targeted attack*. Hence, the definition of a targeted attack might vary among researchers according to the level of sophistication employed in different phases of the attack and the criteria specified for selecting the victims. Our decision is to limit ourselves to targeted attacks that meet a specific set of criteria defined as:

[1] Symantec.cloud – http://www.symanteccloud.com/

- low copy number attacks that infect victims with malicious email attachments,
- showing some clear *evidence of a selection* of the subject and the targets, *e.g.*, emails that have an appealing subject or use a spoofed sender address in relation to the activity or the position of the targeted recipients,
- and embedding a relatively sophisticated malware compared to that used in high copy number attacks.

As described in Section 2.4, we use this set of criteria to detect instances of targeted attacks through a semi-automated process which is validated manually by Symantec threat analysts.

2.2 A Typical Modus Operandi

We observed that targeted attacks occur in several stages and we can usually distinguish the following phases: *incursion*, *discovery*, *capture* and *data exfiltration*.

These stages are best described using an example of a real world compromise of a defence contractor's network which took place in July 2011. A forensic investigation of the attack was performed, which allowed for the creation of a timeline showing how the attackers operated[2].

Incursion

Incursion is the stage in which an attacker attempts to penetrate a network. Attackers can use a number of approaches to achieve this. A common technique, probably due to the low level of effort required, is to send an email containing a malicious attachment to a victim. Alternative approaches are for the attacker to locate Internet facing services the victim is hosting on their network and attempt to penetrate those using some form of exploit. In this instance, emails were sent to the victims.

Typically, the emails contain PDF attachments that exploit vulnerabilities to drop a malicious backdoor trojan. When a victim receives the email and opens the attached document, their computer is compromised. The malicious document drops a backdoor trojan which connects to a remote command and control server and waits for commands from the attacker. Figure 1 is a 'map' of three PDF documents that were involved in the attack. The three documents, although containing different malicious samples, are otherwise identical. The PDFs are being created by a PDF 'exploit kit', which takes an empty PDF document and loads it with a malicious executable.

Discovery

At this point, the attacker can begin evaluating the network, identifying exactly what has been compromised and begin spreading through the network. In the example used, the contractor was compromised by an email which contained `wsn.exe`, as shown in Figure 1. The email which contained this particular executable was not located. A visualisation of that timeline is shown in Figure 2. Computer A is compromised at 09:43 on July 14th by `wsn.exe`. The attacker consolidates the compromise by downloading additional hacking tools on July 25th. Discovery begins on July 27th with the file `n.bat`. This is a very simple batch script which scans the local network for open shares and logs this list to a file. The attacker in then in a position to move onto the next stage.

[2] Some file names have been changed to protect the identity of the victim.

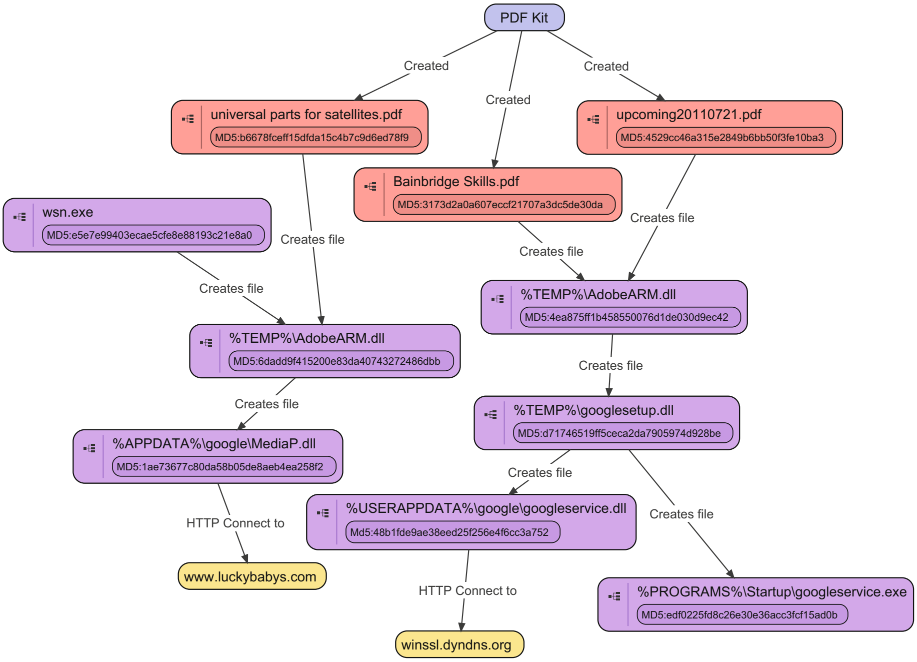

PDF Kit
Created
Created
Created
universal parts for satellites.pdf
MD5:b6678fceff15dfda15c4b7c9d6ed78f9
upcoming20110721.pdf
MD5:4529cc46a315e2849b6bb50f3fe10ba3
Bainbridge Skills.pdf
MD5:3173d2a0a607eccf21707a3dc5de30da
wsn.exe
MD5:e5e7e99403ecae5cfe8e88193c21e8a0
Creates file
Creates file
Creates file
Creates file
%TEMP%\AdobeARM.dll
MD5:4ea875ff1b458550076d1de030d9ec42
%TEMP%\AdobeARM.dll
MD5:6dadd9f415200e83da40743272486dbb
Creates file
Creates file
%TEMP%\googlesetup.dll
MD5:d71746519ff5ceca2da7905974d928be
%APPDATA%\google\MediaP.dll
MD5:1ae73677c80da58b05de8aeb4ea258f2
Creates file
Creates file
%USERAPPDATA%\google\googleservice.dll
Md5:48b1fde9ae38eed25f256e4f6cc3a752
HTTP Connect to
www.luckybabys.com
HTTP Connect to
winssl.dyndns.org
%PROGRAMS%\Startup\googleservice.exe
MD5:edf0225fd8c26e30e36acc3fcf15ad0b

Fig. 1. Example of emails sent during the incursion phase

Capture

Having obtained a list of additional computers on the victim network, the attacker begins to spread. On July 28th, the file `72492843` was created on Computer B. This file is a backdoor trojan which gives the attacker a limited control of Computer B.

When the attacker determines that Computer B was a computer of interest, on August 10th, the attacker downloads `msf.dat`, a more advanced backdoor which contains a password stealer and keylogger. The stolen data is logged to the local hard drive, ready for exfiltration.

Exfiltration

The attacker downloads additional, more comprehensive backdoor tools, on August 12th, for Computer A and August 22nd for Computer B. These tools, referred to as Remote Access Tools, or RATs, give the attacker complete control over the compromised computer. These tools let the attacker easily upload stolen data, including documents, passwords and logged key presses. The attacker can also perform more discovery from newly compromised computers. The cycle of discovery, capture and exfiltration is thus repeated until the attacker has thoroughly compromised the network and achieved his or her goals. With regard to stolen data, in some cases (*e.g.* the *Luckycat* attacks [15]) we could identify command and control servers which had a list of stolen data. In that particular instance, the attackers appear to have carefully picked both source code and research documents related to military systems.

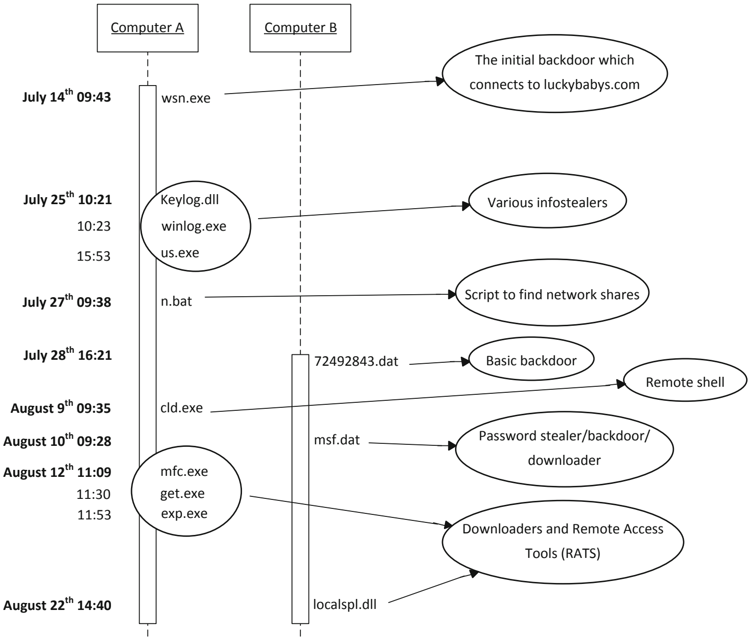


Fig. 2. Timeline of attack

2.3 Signs of a Growing Menace

In 2005, targeted attacks were observed by *Symantec.cloud* at the rate of one attack per week, rising to one or two per day during 2006, to approximately 60 per day during 2010, and approximately 100 per day towards the end of 2011 [14]. From April 2008 to April 2012, about 96,000 targeted attack emails have been identified and recorded by Symantec, with just under 30,000 of these being identified during 2011 alone. Figure 3 illustrates the growing trend in targeted attack activity over time by showing the average number of attacks blocked by Symantec on a daily basis in 2011.

These figures must be interpreted in the context of the 500,000 malware and phish emails that are detected each day by *Symantec.cloud*. Targeted attacks remain rare when compared with non-targeted malware. However, it is this rarity that makes detection all the more difficult. Without large volumes of email to sample, it is not possible to collect a corpus of such attacks containing enough samples to identify similarities between the attacks.

Two targeted attacks, publicised over the last year, describe the capabilities of attackers. Those are *The Nitro Attacks* [3] and *The Rise of Taidoor* [13]. The NITRO campaign targets chemical and petroleum companies. The attackers do not employ sophisticated exploits but instead rely on social engineering methods to trick victims into installing a backdoor. E-mails sent to the victims contain a compressed, password protected archive. The archive in turn contains a variant of Poison Ivy, a full featured RAT which gives

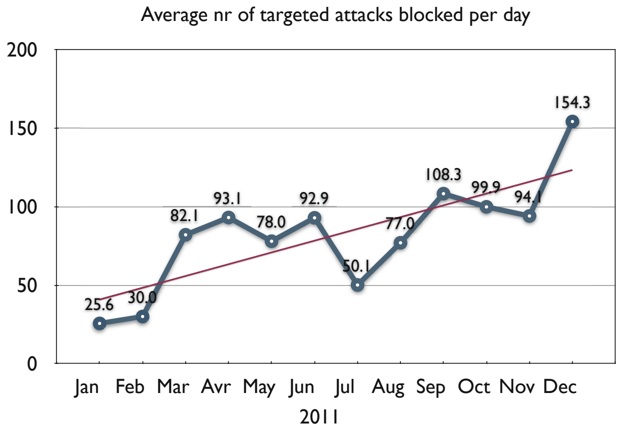


Fig. 3. Average number of targeted attacks

an attacker complete control of compromised computers. Twenty nine companies in the chemical industry were identified as being targeted and over 100 unique IPs were compromised. The infections spanned multiple countries but were primarily located in the United States of America. The ease with which the attackers compromised major corporate entities demonstrates how serious such targeted attacks are.

The TAIDOOR attacks employ instead more sophisticated malware to compromise victim computers. The victim industries vary, ranging from educational to think thanks. Over 160 emails and more than 300 variants of this threat have been identified by Symantec indicating the level of continuous effort required by the attackers.

2.4 Detecting Targeted Attacks

Targeted attacks can be identified by analysing emails containing malware which were intercepted by Symantec.cloud's email protection service. As explained in Section 2.1, we identify a targeted attack as an email attack that meets a specific set of criteria. High copy number malware containing emails, as evidenced by the malware or by the email to which it is attached, are discarded. The low copy number attacks are analysed by a semi-manual process to identify false positives, apparent prototypes for future high copy number attacks, as well as low sophistication malware; all of these are discarded. The remaining emails are manually screened to identify targeted malware containing emails that exhibit a high degree of sophistication. These emails are logged and added to the corpus of targeted attacks. The corpus of targeted attack emails almost certainly includes some emails that may not be considered as targeted attacks according to other criteria; equally there are almost certainly targeted attacks that should have been included that have been omitted. Nevertheless, this corpus represents a large number of sophisticated targeted attacks compiled according to a consistent set of criteria.

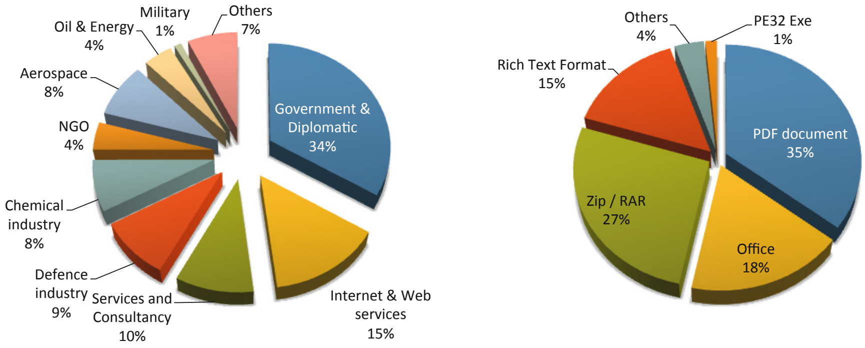


Fig. 4. *Left*: Top targeted Sectors. *Right*: Most frequent document types seen as email attachments.

3 Experimental Dataset

For the rest of our analysis, we have focused on a corpus of 26,000 email attacks identified by *Symantec.cloud* during 2011 as targeted based on the set of criteria defined previously in Section 2. A subset of 18,580 attack samples were then selected in an automated email parsing and filtering phase based on additional criteria, such as a well-formed and complete email header and body, a valid attachment, etc. Emails that could not be parsed or decoded successfully were simply ignored for this analysis.

Emails written in a foreign language were automatically translated to English, to facilitate the comparison of subject lines but also to understand the topics used by the attackers. Every attachment was further analyzed in a Symantec dynamic analysis platform (DAP) called RATS– the *Response Auto-Analysis Threat System*. RATS is a dynamic analysis platform, which generates and warehouses meta-data on malicious samples seen by Symantec. This meta-data is partially generated through Response-owned data collection agents, as well as by leveraging wider data sources available to Symantec (AV engines, CVE databases, etc). Dynamic data is collected through executing every sample in a sandbox environment and logging its activity on the file system and on the network, allowing us to detect any network connection attempt to external domains, which could possibly be linked to C&C activity. File parsers and other static analysis tools extract file-level characteristics from the sample, such as document type and the associated software application and version. Fig. 4 (*right*) shows for example the distribution of document types seen as malicious attachments in our email dataset. We can observe that malicious PDFs continue to be largely used in targeted attacks (over $1/3^{rd}$ of the attacks). However, malicious Zip and Rar archives start to be very frequent too (27% of the attacks). Malicious documents exploiting some vulnerability in Microsoft Office have been used as droppers in about 18% of targeted attacks. It is also worth noting that PE32 executable attached to emails are quite infrequent in targeted attacks (only 1% of attacks).

All email recipients were also categorized into *activity sectors*, based on the information we have on our customers as well as public information sources on the companies and their business. This allows us to get insights into the top targeted sectors, as shown in Fig. 4 (*left*). Not surprisingly, we observe that the most frequently targeted organizations are in the Government & Diplomatic sector. Note that the Military sector does not seem to be often targeted directly, but rather indirectly through employees of Defense and Aerospace industries or Governmental institutions.

Finally, we have extracted a number of email characteristics (or *features*), which we think are potentially relevant to correlate attacks, and ultimately identify series of attacks that are likely originating from the same individuals, forming what we identify as *attack campaigns* in Section 4. Some of those extracted features are:

Origin-Related Features: these are characteristics of the source of the attack, such as the *IP address* the email was originating from, the *country* and *ISP* associated to the origin IP, the *mailer* agent used to send the email (*e.g.*, Microsoft Outlook Express 6.00.*x*), and the email *From* address used by the sender.

Attack-Related Features: these are characteristics of the attack by itself. Examples include the *attachment MD5* and *filename*, the *date* of the attack, the *subject* line and *body* of the email, the *language* of the message, the *character set* used to encode the email, the *AV signature* given for the malicious attachment (if any), and the *document type* of the attachment.

Target-Related Features: these include characteristics of the targeted recipient, such as the destination fields (*To*, *Cc* or *Bcc* address fields of the email), the *activity sector* of the recipient's organization, etc.

Regarding attackers' IP addresses, most of them were located in large industrialized countries, with no particular pattern or abnormal bias for specific countries, and we have never seen *botnets* in use in the targeted attacks we could identify. With respect to *mailer* agents, most of them were apparently reflecting real mail clients with a substantial proportion of targeted attacks (over 45%) sent through popular Webmail providers. Only in 11% of the attacks, the emails had a fake, randomly chosen Mailer agent, perhaps as an attempt to fool certain filters.

4 Analysis of Targeted Attack Campaigns

4.1 Methodology

To identify series of targeted attacks (*i.e.*, attack *campaigns*) that are likely performed by the same individuals, we have used an advanced data analytics software framework named TRIAGE. Originally developed in the context of the WOMBAT project[3], TRIAGE is an attack attribution software that relies on data fusion algorithms and multi-criteria decision analysis (MCDA) [23,21,19]. This TRIAGE technology can automatically cluster virtually any type of attacks or security events based upon common elements (or *features*) that are likely due to the same *root cause*. As a result, TRIAGE can identify complex patterns within a data set, showing varying relationships among series of attacks or groups of disparate events. Previous analyses of various threat landscapes

[3] Worlwide Observatory of Malicious Behaviors and Threats. http://www.wombat-project.eu/

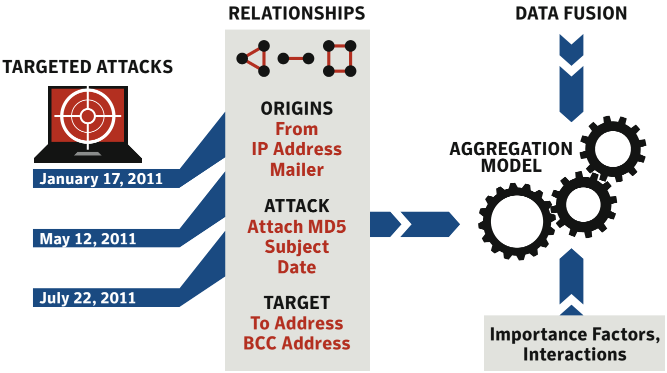


Fig. 5. Illustration of the TRIAGE methodology

[20,4,5] have demonstrated that TRIAGE can provide insights into the manner by which attack campaigns are being orchestrated by cyber criminals, and more importantly, can help an analyst understand the *modus operandi* of their presumed authors.

The TRIAGE approach, as used in this analysis, is represented in Figure 5. For each targeted attack, a number of attack features were selected from the set of characteristics extracted previously (see Section 3). In this specific analysis, we have selected the following features, which we believe could be relevant for linking attacks originating from the same team of attackers: the MD5 hash and the *fuzzy hash* (as given by *ssdeep* [9]) of the email attachment, the origin IP address, the *From* address, the email subject, the sending date, the targeted mailbox (*To* field), the mailer agent, and the AV signature.

In the second step, TRIAGE builds relationships among attack samples with respect to every selected attack feature by using appropriate similarity metrics [19]. In the final step, all feature similarities are fused using an *aggregation* model reflecting some high-level behavior defined by the analyst, *e.g.*, *at least* k highly similar attack features (out of n) are required to attribute different attack samples to the same campaign. In this analysis, we have also assigned different *weights* to attack features, by giving a higher importance to features like MD5, email subject, sender *From* and IP address; a lower importance to the AV signature and mailer agent, and a medium importance to the other features.

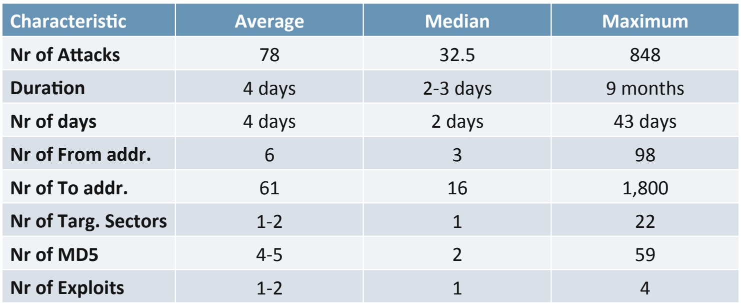
As outcome, TRIAGE identifies attack clusters that are called multi-dimensional clusters (or MDC's), as any pair of attacks within a cluster is linked by a number of common traits. As explained in [19], a decision threshold can be chosen such that undesired linkage between attacks are eliminated, *i.e.*, to drop any irrelevant connection that is due to a combination of small values or an insufficient number of correlated features.

4.2 Insights into Attack Campaigns

Our TRIAGE analysis tool has identified 130 clusters that are made of at least 10 attacks correlated by various combinations of features. We hypothesize that those attack clusters are likely reflecting different *campaigns* organized by the same individuals.

Indeed, within the same cluster, attacks are always linked by at least 3 different attack characteristics.

Figure 6 provides some global statistics calculated across all attack campaigns identified by TRIAGE. This Table shows that the average targeted attack campaign will comprise 78 attacks targeting 61 email addresses within a 4 days-period. Some attack campaigns were observed lasting up to 9 months and targeting as many as 1,800 mailboxes.



Characteristic	Average	Median	Maximum
Nr of Attacks	78	32.5	848
Duration	4 days	2-3 days	9 months
Nr of days	4 days	2 days	43 days
Nr of From addr.	6	3	98
Nr of To addr.	61	16	1,800
Nr of Targ. Sectors	1-2	1	22
Nr of MD5	4-5	2	59
Nr of Exploits	1-2	1	4

Fig. 6. Global statistics of targeted attack campaigns identified by TRIAGE

Based on the number of targeted recipients and sectors, we have thus classified attack campaigns into two main types:

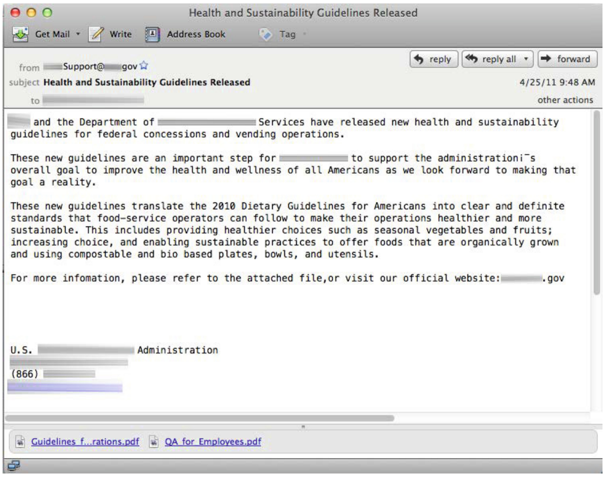
- *Type 1 – Highly targeted* campaigns: highly focused attack campaigns targeting only one or a very limited number of organizations within the same activity sector;
- *Type 2 – Multi-sector* campaigns: larger-scale campaigns that usually target a large number of organizations across multiple sectors. This type of attacks fit the profile of what we have dubbed *Massive Organizationally Targeted Attack* (MOTA).

4.3 Highly Targeted Campaigns: The Sykipot Attacks

$2/3^{rd}$ of the identified attack campaigns are targeting either a single or a very limited number of organizations active in the same sector. Over 50% of those highly focused campaigns target the *Government & Defense* sectors. However, other industries clearly are experiencing such highly targeted attacks. Our results show that *niche* sectors are usually more targeted by those very focused attacks. For example, industries active in sectors like Agriculture, Construction, Oil and Energy mainly see attacks that are very targeted at a small number of companies and individuals within them.

A good example of highly targeted campaign is SYKIPOT, a long series of attacks that has been running for at least the past couple of years[4]. These long-running series of attacks are using the Sykipot family of malware, with a majority of these attacks targeting the Defense industry or governmental organizations. The latest wave spiked on December 1, 2011 with a huge uptick of targeted entities being sent a PDF containing a zero-day exploit against Adobe Reader and Acrobat (CVE-2011-2462).

[4] Unconfirmed traces of SYKIPOT date back to as early as 2006 in Symantec threat data.



from [redacted]Support@[redacted]gov
subject **Health and Sustainability Guidelines Released** 4/25/11 9:48 AM
to [redacted]

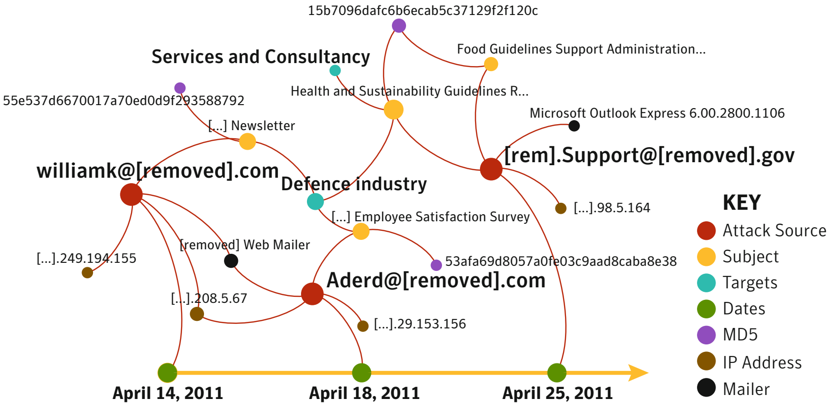
```
[redacted] and the Department of [redacted] Services have released new health and sustainability
guidelines for federal concessions and vending operations.

These new guidelines are an important step for [redacted] to support the administrationi¯s
overall goal to improve the health and wellness of all Americans as we look forward to making that
goal a reality.

These new guidelines translate the 2010 Dietary Guidelines for Americans into clear and definite
standards that food-service operators can follow to make their operations healthier and more
sustainable. This includes providing healthier choices such as seasonal vegetables and fruits;
increasing choice, and enabling sustainable practices to offer foods that are organically grown
and using compostable and bio based plates, bowls, and utensils.

For more infomation, please refer to the attached file,or visit our official website:[redacted].gov




U.S. [redacted] Administration
[redacted]
(866) [redacted]
[redacted]
```

Guidelines f...rations.pdf QA for Employees.pdf

Fig. 7. An example of Sykipot email attack

The modus operandi of SYKIPOT attackers is actually always the same: they send to carefully chosen targets an email with an appealing subject, sometimes using a spoofed email address in relation to the activity or the position of the recipient, and containing a malicious document, which usually exploits some vulnerability in Adobe or Microsoft Office software products. Figure 7 shows an example of such email. To make it looking more legitimate, the attacker used a sender address belonging to a large US administration that is directly related (at least partially) to the business of the targeted Defense industry.

Figure 8 visualizes a SYKIPOT attack wave identified by TRIAGE in April 2011. Three different attackers (red nodes) have sent about 52 emails to at least 30 mailboxes of employees working for two different Defense industries on three different dates. Many subject lines (yellow key) are shared among attackers and two of them used the same



15b7096dafc6b6ecab5c37129f2f120c
Services and Consultancy
Food Guidelines Support Administration...
55e537d6670017a70ed0d9f293588792
Health and Sustainability Guidelines R...
Microsoft Outlook Express 6.00.2800.1106
[...] Newsletter
[rem].Support@[removed].gov
williamk@[removed].com
Defence industry
[...].98.5.164
[...] Employee Satisfaction Survey
[removed] Web Mailer
[...].249.194.155
53afa69d8057a0fe03c9aad8caba8e38
Aderd@[removed].com
[...].208.5.67
[...].29.153.156
April 14, 2011
April 18, 2011
April 25, 2011
KEY
Attack Source
Subject
Targets
Dates
MD5
IP Address
Mailer

Fig. 8. Visualizing a SYKIPOT campaign and the relationships between different attacks

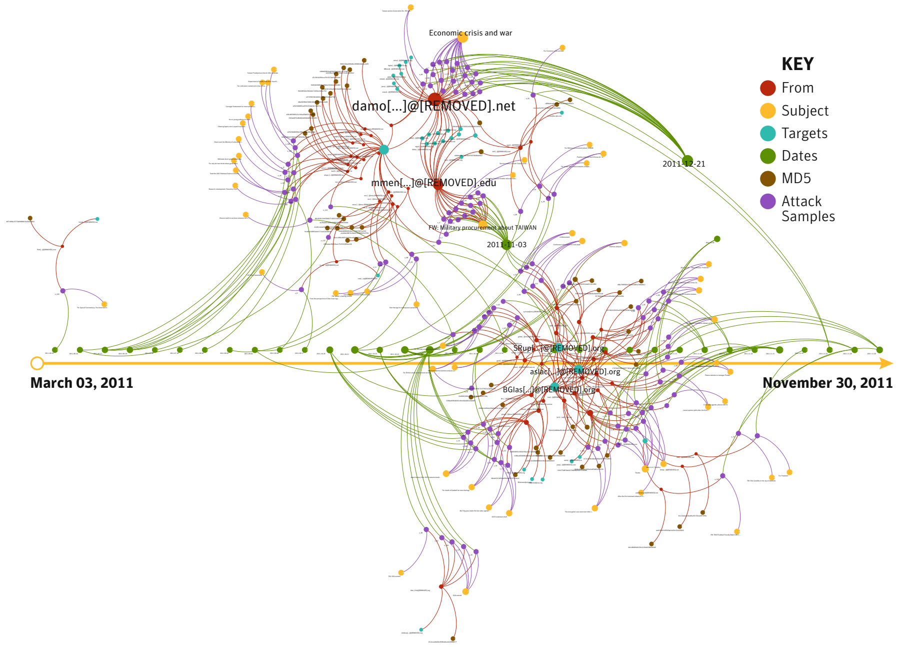


Fig. 9. Visualizing the complexity of relationships among TAIDOOR attacks

mailer agent from the very same IP address to launch the attacks. Note that three different MD5s were used in this SYKIPOT campaign (nodes in purple).

4.4 Massive Organizational Targeted Attacks (MOTA): Nitro and Taidoor

About $1/3^{rd}$ of attack campaigns are instead organized on a "large-scale" and fit the profile of a *Massive Organizationally Targeted Attack* (MOTA), *i.e.*, they target many people in multiple organizations, working in different sectors, over multiple days. Most of these large-scale campaigns are very well resourced, with up to 4 different exploits used during the same campaign. Some are even *multilingual* – meaning that the language used in the email attack is tuned to the targeted recipients (use of Chinese for `.cn` recipient domains, Japanese for `.jp`, Russian for `.ru`, etc).

The TAIDOOR attacks, which we covered in a previous threat report [13], illustrate also nicely this type of mass-scale attack campaign. These attacks can include a long series of attack waves, sometimes spread over a long period of time (several months, or even a few years in some cases). As illustrated in Figure 9, the relationships between attackers in those campaigns are usually much more complex, involving many interconnections at multiple levels (*e.g.*, common MD5s, same mailer or IP address, etc). This might indicate that several teams of attackers are collaborating or sharing some of their resources (like malicious code, virtual servers to launch attacks, or intelligence data on the targets). They usually target a very large number of recipients working for different organizations, which can be active in completely different sectors.

The NITRO attacks are another example of *mass-scale* attack campaign also identified by TRIAGE. The bulk of the NITRO attacks was launched in late July 2011 and continued into mid-September. Another unconfirmed NITRO campaign was also identified later in October 2011. As for many other targeted attacks, the purpose of the NITRO attacks appears to be industrial espionage, mainly targeting the chemical and petroleum industries, to collect intellectual property for competitive advantage. An example of email sent during those NITRO attack waves is given below. In this campaign, Symantec blocked over 500 attacks of this type, in which the attackers use a spoofed email address (presumably coming from an IT support desk) to entice users to install a fake Adobe software update packaged in a zip file, and which contains a zero-day exploit to compromise the users machines. While most targeted recipients were employees working for chemical industries, our research has showed that the NITRO attackers have also targeted senior executives working in the Defense industry and the Aerospace domain during the same series of attacks in October 2011.

Case Study - The NR4 Campaign

Let's focus on another case of MOTA campaign (Figure 10). NR4 is one mass-scale attack campaign out of 130 identified by TRIAGE (note that there is no significance to the name NR4). We do not know the ultimate goal of the attackers behind this campaign, but we do know that they were targeting diplomatic and government organizations.

In this NR4 campaign, 848 attacks were made on 16 different days, over a 3 months period. The attacks all originated from accounts on a popular free webmail service. All attacks came from one of three different sender aliases. Multiple email subject lines were used in the targeted attacks, all of potential interest to the recipients, with the majority being about current political issues. Almost all targeted recipients were put in BCC field of the email.

The first wave of attacks began 4/28/2011 from a single email alias. Four organizations were targeted in this first series of attacks. One of these organizations saw the CEO as well as media and sales people targeted. Over the course of the attack campaign the CEO was targeted 34 times.

On 5/13/2011 a new email account began sending email to targets. It was from this account that the majority of the attacks occurred. This aliases continued attacks on the four previous organizations but added dozens of additional organizations. One organization first targeted in this attack wave was targeted 450 times. A total of 23 people in the organization were targeted, with the main focus being on researchers within the organization.

The final attack wave started 6/30/2011 and ended 19 days later. While attacking a number of organizations already part of the campaign, it also targeted 5 new organizations.

By 7/19/2011 the NR4 targeted campaign came to an end. During the 3 months of this campaign hundreds of emails, in English and in Chinese (used against Chinese speaking targets) arrived in targeted users mailboxes. While the content of the email was constantly being changed, each email contained an attached PDF or RAR file with the same exploit that would infect users once the attachment was opened. Interestingly, we also found that the three attackers involved in this NR4 campaign have been using the same C&C servers for controlling compromised machines and exfiltrating data.

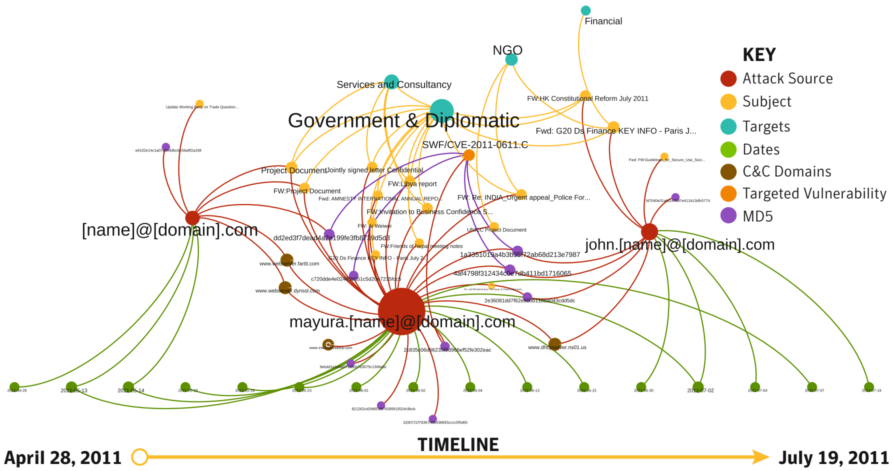

Financial
NGO
Services and Consultancy
Government & Diplomatic
FW:HK Constitutional Reform July 2011
Fwd: G20 Ds Finance KEY INFO - Paris J...
SWF/CVE-2011-0611.C
Update Working Party on Trade Question...
e6102e14c1a07e534dbc0e39a8f2a338
Project Document
Jointly signed letter Confidential
FW:Libya report
FW:Project Document
Fwd: AMNESTY INTERNATIONAL ANNUAL REPO...
FW: Re: INDIA_Urgent appeal_Police For...
FW:Invitation to Business Confidence S...
FW: Ai Weiwei
UNICC Project Document
Fwd: FW:Guidelines_for_Secure_Use_Soci...
fd7040e31afd147597e411b13dfc5774
[name]@[domain].com
dd2ed3f7dead4a2e199fe3fb8739d5d3
FW:Friends of Nepal meeting notes
G20 Ds Finance KEY INFO - Paris July 2...
1a3351019a4b3b35f72ab68d213e7987
john.[name]@[domain].com
www.webserver.fartit.com
c720dde4e02485b551c5d20e7235fcc5
4af4798f312434c0e7db411bd1716065
www.webserver.dynssl.com
2e36081dd7f62e800b1188a043cdd5dc
mayura.[name]@[domain].com
www.internet.freetcp.com
2c835c06d66235f9d966ef52fe302eac
www.dhcpserver.ns01.us
9ebdd1e1e0a071f3cb7e3075c1306abc
2011-04-28
2011-05-13
2011-05-14
2011-05-18
2011-05-19
2011-05-23
2011-06-01
2011-06-02
2011-06-04
2011-06-13
2011-06-15
2011-06-30
2011-07-02
2011-07-04
2011-07-07
2011-07-19
821262cd2fd6f2887638951f024c9bcb
2d30721f78367006438693cccc5f5d65
KEY
● Attack Source
● Subject
● Targets
● Dates
● C&C Domains
● Targeted Vulnerability
● MD5
TIMELINE
April 28, 2011
July 19, 2011

Fig. 10. Visualizing the NR4 targeted attack campaign (*mass-scale* type)

5 On the Prevalence and Sophistication Level of Targeted Attacks

In this Section, we aim at finding evidences that support the validity of following assumptions we make about the targeted attacks:

- Targeted attacks employ more sophisticated exploitation methods compared to other types of attacks, therefore, the attachment files in our experimental data set should not be seen in the wild long before the attack dates we observed.
- Targeted attacks target only a limited number of organizations or sectors, therefore, the attachment files or the dropped binaries after successful exploitations should not be present on a large number of computers.

In addition to the endeavour to find such evidences, we perform deeper investigation on the sophistication level of targeted attacks. To this aim, we enrich the existing information about the attachment files used in targeted attacks by correlating a number of components of Symantec's Worldwide Intelligence Network Environment (WINE).

5.1 Investigating Historical Information

To achieve the goals set here above, we took advantage of the information provided by Symantec's dynamic malware analysis platform, by VirusTotal[5] and by the WINE platform [7].

The Symantec's dynamic analysis platform (DAP) not only analyzes binary files but also other file types such as **.pdf, *.doc, *.xlsx*. To get a better understanding of the intentions of the attackers, one reasonable approach is to analyze the files that are downloaded after successful exploitations. Hence, we use Symantec's DAP to acquire

[5] VirusTotal – https://www.virustotal.com/

a list of downloaded or created files (*droppee*) once the attachment file (*dropper*) has been opened by the victims and has successfully compromised their computer. We have then searched for all droppees in WINE to get some historical information on those files. To validate our results, we also queried VirusTotals to be able to compare them with other Anti-Virus companies' findings.

WINE is a platform through which Symantec shares data with the research community[6]. WINE consists of a number of datasets such as a malware samples collection, binary files reputation data, A/V telemetry data, e-mail spam, URL reputation, IPS telemetry and yet some others. The WINE datasets on which we performed our analysis are the A/V telemetry and the binary reputation system. Since the beginning of 2009, the A/V telemetry records the detections of known threats for which Symantec generated a signature and was subsequently deployed in anti-virus products. A typical record in A/V telemetry data comprises several fields; however, in our analysis we only use the detection time, the associated threat label and the hash (MD5 and SHA2) of the malicious file.

The binary reputation data does not involve threat detections; instead, it includes all binary files that were not detected by Symantec's security products. This data allows us to look back in time and get more insights about what happened before signatures for malicious binaries were created. The binary reputation component records the download date and time of all binaries, the source they were downloaded from and some information about the binary file, such as the file name, the hash and the file size. Since mid-2008, the binary reputation data is collected only from Symantec customers who gave consent to share this invaluable information.

In summary, to investigate the prevalence and sophistication of the targeted attacks studied in this paper, we apply the following methodology for each campaign TRIAGE produces: *(i)* we prepare a list of MD5 hashes of the droppers employed in the attacks. The droppers afterwards are searched in Symantec's DAP to find the associated droppees. *(ii)* Once dropper-droppee association list is constructed, both types of files are searched in the A/V telemetry data to determine the lifetime of the targeted attacks launched by each campaign. *(iii)* Finally, we search the droppees in the binary reputation data. Note that droppers in our experimental set cannot be found there, since binary reputation data only stores binary files.

Correlating the results of the three WINE components[7], we extract following information about the dropper and droppees:

- The first and last time the file was detected to be malicious,
- The first and last time the file was downloaded by our customers before a detection signature was generated for the specific threat,
- The number of machines that downloaded or attempted to download the file,
- The associated threat name and vulnerability id if it exists.

[6] WINE provides external researchers access to security-related data feeds collected in the wild. See `http://www.symantec.com/WINE` for more information.

[7] The WINE data set used for this analysis is available to other researchers as "WINE 2012-002".

In the following section, we show how we leveraged this information to find interesting results about the prevalence and sophistication of the targeted attacks we analyzed in this paper.

5.2 Malware Analysis Results

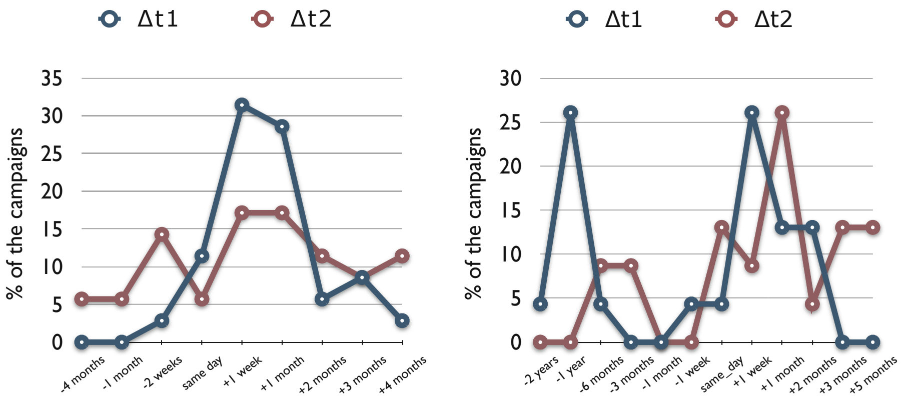
We performed a set of experiments to get more insights about the characteristics of malware used in targeted attacks analyzed in this paper.

The first experiment consists in querying 18,850 attachment files in Symantec's DAP and VirusTotal. Not surprisingly, only 941 (5.0%) of the droppers used in targeted attacks were identified. Therefore, we could not retrieve any information about 195 out of 345 campaigns TRIAGE produces from either Symantec or any other A/V product. Since a majority of the attachments were not found by any of the Anti-Virus scanners, we could conclude that targeted attacks are not very prevalent. If they were as prevalent as other types of large-scale attacks, it would have been harder to stay undercover. Another possible reasoning would be that targeted attacks are carried out through more sophisticated techniques, and therefore, they manage to evade most of the security walls and can stay hidden over longer periods of time.

In our second experiment, we parsed the analysis reports produced by Symantec's DAP to acquire the list of droppees that were downloaded or created after droppers successfully compromised the victims. The droppers after the exploitation stage created 1,660 distinct files. We then searched all droppers and their associated droppees in WINE and extracted the information listed above. In WINE, we found records for droppee/dropper detections and downloads for only 51 attack campaigns identified previously by TRIAGE. The explanation for not finding all dropper and droppees of the remaining campaigns can be that the victims of the targeted attacks in our experimental dataset were not Symantec customers. Another reason could be related to the fact that more sophisticated targeted attacks could use *zero-day* attacks, and therefore might escape signature-based detection methods.

To measure the prevalence of those 51 attacks campaigns found in WINE, we computed the number of computers reporting the presence of the droppees or the droppers. The average number of machines that were subject to one of the attacks sourced by one of the campaigns is only 5. Hence, this is also a strong evidence that seems to show that targeted attacks are not very prevalent.

Targeted attacks are usually active for a limited period of time varying from a few days to several months. To validate this claim, we compared the first and last time the droppers/droppees were recorded in WINE with the start and end time of the targeted attacks we have analyzed. In Figures 11(a) and 11(b), Δt_1 represents the difference between the start time of the attack and the first observation time of the related malware in WINE, whereas Δt_2 represents the difference between the end time of the targeted attack and the last observation time of the related malware in WINE. Figure 11(a) shows that the majority of the attachment files were not observed in the wild more than 2 weeks before the attack time specified by Symantec.cloud. On the other hand, droppees exhibit a different behavior than droppers. Figure 11(b) shows that, while there are some droppees that are never observed outside the attack window (*i.e.*, the period in which



(a)

(b)

Fig. 11. First and last detection time of the droppers/droppees in WINE compared to the first and last day of the targeted attacks

the attack is being observed in the wild), there are also some campaigns that reuse old malware to perform the subsequent stages of the targeted attack.

In the previous sections, we have shown that targeted attacks employ a wide range of vulnerabilities. While some exploit old vulnerabilities (*i.e.*, disclosed in 2009 or 2010), approximately 50% of them exploit vulnerabilities that were disclosed in 2011. Since we performed our analysis on data collected in 2011, it is possible that some of the targeted attacks were actually *zero-day* attacks. A *zero-day* is defined to be an attack that involves exploitation of a vulnerability that is still not publicly disclosed at the time of the attack [1]. To identify the campaigns that might have performed zero-day attacks, we compared their attack window with the disclosure date of the vulnerability they exploit. As a result, eight campaigns started on average 16 days before the disclosure date; therefore, the attacks involved in those campaigns were apparently exploiting unknown vulnerabilities at that time (*i.e.*, CVE-2011-0609, CVE-2011-0611 and CVE-2011-2462), and thus we can safely conclude that the attackers were using *zero-day* attacks.

To the best of our knowledge, the remaining of the campaigns did not perform zero-day attacks. However, attackers reacted very fast to deploy new exploits during their attack campaign. Indeed, most of them started exploiting zero-day vulnerabilities just a few days after the disclosure date.

The campaigns we have identified in this paper are associated with a number of different droppers during their lifetime. Attackers can create different droppers over time, either to evade malware detection systems by applying polymorphism, or to increase their effectiveness by adding new exploitation methods as new vulnerabilities are being disclosed (*e.g.*, `Taidoor` attacks). However, a majority of the campaigns do not update their droppees over time or do not apply polymorphism at that stage. The graph in Figure 12 shows that for almost 70% of the campaigns, a dropper downloads exactly the same droppees, *i.e.*, trojans or backdoors that are shared with other droppers used within the same campaign.

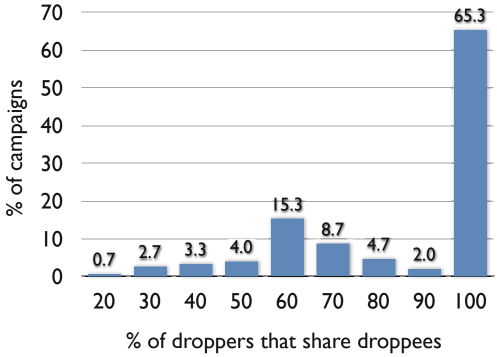
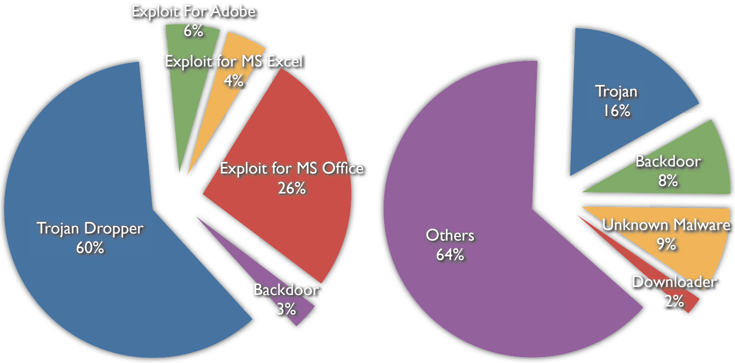


Fig. 12. The percentage of campaigns whose exploit files share the same malware



(a) Droppers (b) Droppees

Fig. 13. AV labels for the droppers and droppers

As explained in Section 2, in a typical targeted attack scenario, the attackers first attempt to compromise the victim to install a backdoor or a trojan such that in later stages of the attack, data exfiltration can be realized. To find out whether the targeted attacks we analyzed always follow the same lifecyle, as a final experiment we have identified the threat categories that droppers and droppees fall into. Figure 13(a) illustrates that the majority of the droppers fall into the Trojan or backdoor category. Therefore, it is fair to conclude that, after the exploitation phase, most malicious attachments downloaded another malicious binary that aims at installing a backdoor program. On the other hand, only one quarter of the droppees were trojans or backdoors. This shows that some of the campaigns reuse old malware and perform other malicious activities, in addition to exfiltrating sensitive information from the targeted organization.

6 Conclusion

Targeted attacks are still rare occurrences today compared to classical malware attacks, which are usually more profit-oriented and performed on a much larger-scale. However, this type of attacks can be extremely difficult to defend against and has the potential to seriously impact an organization. In the longer-term, targeted attacks and APTs represent a significant threat against the economic prosperity of many companies and against the digital assets of governmental organizations, as demonstrated by the recent high profile attacks that made the headlines in 2010-2011.

To understand the real nature of targeted attacks and how those attacks are being orchestrated by motivated and well resourced teams of attackers, we have conducted an in-depth analysis of 18,580 email attacks that were identified as targeted by *Symantec.cloud* during the year 2011. Using advanced TRIAGE data analytics, we were able to attribute series of targeted attacks to attack *campaigns* likely performed by the same individuals. By analyzing the characteristics and dynamics of those attack campaigns, we have provided new insights into the *modus operandi* of the attackers involved in various series of targeted attacks launched against industries and governmental organizations. Our research has clearly demonstrated that a targeted attack is rarely a single attack, but instead that attackers are often determined and patient, as they usually perform long-running campaigns which can sometimes go on for months while they target in turn different organizations and adapt their techniques.

We also showed that about 2/3rd of attack campaigns were *highly focused* and targeting only a limited number of organizations within the same activity Sector (such as the SYKIPOT attacks), whereas 1/3rd of the campaigns were fitting the profile of a *Massive Organizationally Targeted Attack* (MOTA) – *i.e.*, targeting a large number of organizations across multiple sectors (like in the NITRO and TAIDOOR attacks).

Finally, we have evaluated the prevalence and sophistication level of the targeted attacks in our dataset by analyzing the malicious attachments used as *droppers*. While a large deal of the attacks are apparently relying more on social engineering, have a low level of sophistication and use little obfuscation, our malware analysis also showed that, in at least eight attack campaigns, attackers were using *zero-day* exploits against unknown vulnerabilities which were disclosed two weeks after the date of the first series of attacks observed for those campaigns.

Acknowledgments. We would like to thank the Symantec.cloud and Security Response teams for providing us with the data set and for sharing thoughts on this analysis. Thanks also to Marc Dacier, Corrado Leita, Jakob Fritz for reviewing this paper. Special thanks to Tony Millington, for his continued persistence in identifying targeted attacks.

This research was partly supported by the European Commission's Seventh Framework Programme (FP7 2007-2013) under grant agreement nr. 257495 (VIS-SENSE). The opinions expressed in this paper are those of the authors and do not necessarily reflect the views of the European Commission.

References

1. Zero-day Attack, http://en.wikipedia.org/wiki/Zero-day_attack
2. Bejtlich, R.: Understanding the Advanced Persistent Threat. Searchsecurity Magazine (July 2010), http://searchsecurity.techtarget.com/magazineContent/Understanding-the-advanced-persistent-threat
3. Chien, E., O'Gorman, G.: The Nitro Attacks, Stealing Secrets from the Chemical Industry. Symantec Security Response, http://bit.ly/tDd3Jo
4. Cova, M., Leita, C., Thonnard, O., Keromytis, A.D., Dacier, M.: An Analysis of Rogue AV Campaigns. In: Jha, S., Sommer, R., Kreibich, C. (eds.) RAID 2010. LNCS, vol. 6307, pp. 442–463. Springer, Heidelberg (2010)
5. Dacier, M., Pham, V., Thonnard, O.: The WOMBAT Attack Attribution Method: Some Results. In: Prakash, A., Sen Gupta, I. (eds.) ICISS 2009. LNCS, vol. 5905, pp. 14–18. Springer, Heidelberg (2009)
6. Downs, J.S., Holbrook, M.B., Cranor, L.F.: Decision strategies and susceptibility to phishing. Institute for Software Research. Paper 20 (2006)
7. Dumitras, T., Shou, D.: Toward a Standard Benchmark for Computer Security Research: The Worldwide Intelligence Network Environment (WINE). In: EuroSys BADGERS Workshop (2011)
8. Falliere, N., Murchu, L.O., Chien, E.: W32.Stuxnet Dossier (February 2011), http://www.symantec.com/security_response/whitepapers.jsp
9. Kornblum, J.: Identifying almost identical files using context triggered piecewise hashing. Digital Investigation 3(suppl.), 91–97 (2006)
10. MacSweeney, G.: The Top 9 Most Costly Financial Services Data Breaches, http://www.wallstreetandtech.com/data-security/232800079
11. Pescatore, J.: Defining the Advanced Persistent Threat (2010), http://blogs.gartner.com/john_pescatore/2010/11/11/defining-the-advanced-persistent-threat/
12. Ross, R., Katzke, S., Johnson, A., Swanson, M., Stoneburner, M., Stoneburner, G.: Managing Risk from Information Systems: An Organizational Perspective. NIST Spec. Publ. 800-39 Appendix B
13. Doherty, S., Krysiuk, P.: Trojan.Taidoor: Targeting Think Tanks. Symantec Security Response, http://bit.ly/ymfAcw
14. Symantec. Symantec Intelligence Report (November 2011), http://bit.ly/slWzF5
15. Symantec Security Response. The Luckycat Hackers, White paper, http://www.symantec.com/security_response/whitepapers.jsp
16. Symantec Security Response. The Trojan.Hydraq Incident: Analysis of the Aurora 0-Day Exploit (January 2010), http://www.symantec.com/connect/blogs/trojanhydraq-incident-analysis-aurora-0-day-exploit
17. The Ponemon Institute. Growing Risk of Advanced Threats. Sponsored by Netwitness (June 2010), http://www.netwitness.com/resources/whitepapers
18. The Security for Business Innovation Council. When Advanced Persistent Threats Go Mainstream (August 2011), http://www.rsa.com/go/innovation/index.html
19. Thonnard, O.: A multi-criteria clustering approach to support attack attribution in cyberspace. PhD thesis, École Doctorale d'Informatique, Télécommunications et Électronique de Paris (March 2010)

20. Thonnard, O., Dacier, M.: A strategic analysis of spam botnets operations. In: Proceedings of the 8th Annual Collaboration, Electronic Messaging, Anti-Abuse and Spam Conference, CEAS 2011, pp. 162–171. ACM, New York (2011)
21. Thonnard, O., Mees, W., Dacier, M.: On a multicriteria clustering approach for attack attribution. SIGKDD Explor. Newsl. 12(1), 11–20 (2010)
22. Week, I.: RSA SecurID Breach Cost $66 Million, http://www.informationweek.com/news/security/attacks/231002833
23. WOMBAT. Deliverable D22 (D5.2) Root Causes Analysis: Experimental Report, http://wombat-project.eu/deliverables/

Memory Errors: The Past, the Present, and the Future*

Victor van der Veen[1], Nitish dutt-Sharma[1], Lorenzo Cavallaro[1,2], and Herbert Bos[1]

[1] The Network Institute, VU University Amsterdam
[2] Royal Holloway, University of London

Abstract. Memory error exploitations have been around for over 25 years and still rank among the top 3 most dangerous software errors. Why haven't we been able to stop them? Given the host of security measures on modern machines, are we less vulnerable than before, and can we expect to eradicate memory error problems in the near future? In this paper, we present a quarter century worth of memory errors: attacks, defenses, and statistics. A historical overview provides insights in past trends and developments, while an investigation of real-world vulnerabilities and exploits allows us to answer on the significance of memory errors in the foreseeable future.

1 Introduction

Memory errors in C and C++ programs are among the oldest classes of software vulnerabilities. To date, the research community has proposed and developed a number of different approaches to eradicate or mitigate memory errors and their exploitation. From safe languages, which remove the vulnerability entirely [53,72], and bounds checkers, which check for out-of-bounds accesses [3,54,82,111], to countermeasures that prevent certain memory locations to be overwritten [25,29], detect code injections at early stages [80], or prevent attackers from finding [11,98], using [8,56], or executing [32,70] injected code.

Despite more than two decades of independent, academic, and industry-related research, such flaws still undermine the security of our systems. Even if we consider only classic buffer overflows, this class of memory errors has been lodged in the top-3 of the CWE SANS top 25 most dangerous software errors for years [85]. Experience shows that attackers, motivated nowadays by profit rather than fun [97], have been effective at finding ways to circumvent protective measures [39,83]. Many attacks today start with a memory corruption that provides an initial foothold for further infection.

Even so, it is unclear how much of a threat these attacks remain if all our defenses are up. In two separate discussions among PC members in two of 2011's top-tier venues in security, one expert suggested that the problem is mostly

* This work was partially sponsored by the EU FP7 SysSec project and by an ERC Starting Grant project ("Rosetta").

D. Balzarotti, S. Stolfo, and M. Cova (Eds.): RAID 2012, LNCS 7462, pp. 86–106, 2012.

solved as “dozens of commercial solutions exist” and research should focus on other problems, while another questioned the usefulness of the research efforts, as they clearly “could not solve the problem”. So which is it? The question of whether or not memory errors remain a significant threat in need of renewed research efforts is important and the main motivation behind our work.

To answer it, we study the memory error arms-race and its evolution in detail. Our study strives to be both comprehensive and succinct to allow for a quick but precise look-up of specific vulnerabilities, exploitation techniques or countermeasures. It consolidates our knowledge about memory corruption to help the community focus on the most important problems. To understand whether memory errors remain a threat in the foreseeable future, we back up our investigation with an analysis of statistics and real-life evidence. While some papers already provide descriptions of memory error vulnerabilities and countermeasures [110], we provide the reader with a *comprehensive bird-eye view* and *analysis* on the matter. This paper strives to be *the* reference on memory errors.

To this end, we first present (Section 2) an overview of the most important studies on and organizational responses to memory errors: the first public discussion of buffer overflows in the 70s, the establishment of CERTs, Bugtraq, and the main techniques and countermeasures. Like Miller et al. [68], we use a compact timeline to drive our discussion, but categorize events in a more structured way, based on a branched timeline.

Second, we present a study of memory errors statistics, analyzing vulnerabilities and exploit occurrences over the past 15 years (Section 3). Interestingly, the data show important fluctuations in the number of *reported* memory error vulnerabilities. Specifically, vulnerability reports have been dropping since 2007, even though the number of exploits shows no such trend. A tentative conclusion is that memory errors are unlikely to lose much significance in the near future and that perhaps it is time adopt a different mindset, where a number of related research areas are explored, as suggested in Section 4. We conclude in Section 5.

2 A High Level View of Memory Error History

The core history of memory errors, their exploitations, and main defenses techniques can be summarized by the branched timeline of Figure 1.

Memory errors were first publicly discussed in 1972 by the Computer Security Technology Planning Study Panel [5]. However, it was only after more than a decade that this concept was further developed. On November the 2nd, 1988, the Internet Worm developed by Robert T. Morris abruptly brought down the Internet [86]. The worm exploited a buffer overflow vulnerability in `fingerd`.

In reaction to this catastrophic breach, the Computer Emergency Response Team Coordination Center (CERT/CC) was then formed [22]. CERT/CC’s main goal was to collect user reports about vulnerabilities and forward them to vendors, who would then take the appropriate action.

In response to the lack of useful information about security vulnerabilities, Scott Chasin started the Bugtraq mailing list in November 1993. At that time,

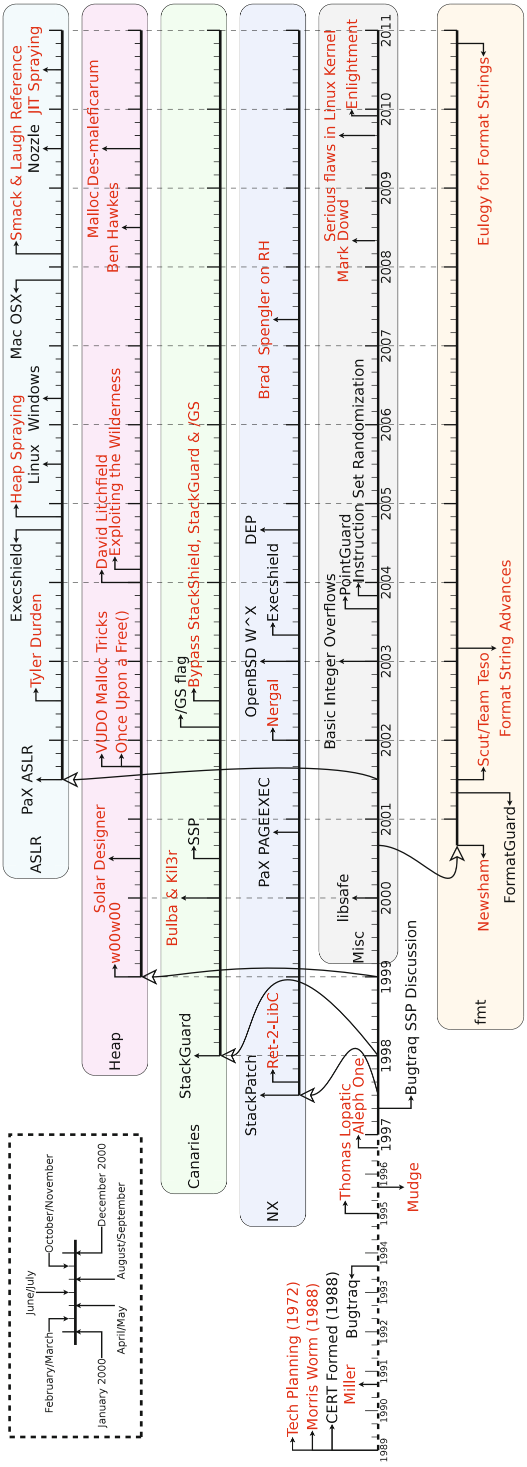


Fig. 1. General timeline

many considered the CERT/CC of limited use, as it could take years before vendors released essential patches. In contrast, Bugtraq offered an effective tool to *publicly* discuss on the subject, without relying on vendors' responsiveness [88].

In 1995, Thomas Lopatic boosted the interest in memory errors by describing a step-by-step exploitation of an error in the NCSA HTTP daemon [63]. Shortly after, Peiter Zatko (Mudge) released a private note on how to exploit the now classic memory error: stack-based buffer overflows [112]. So far, nobody really discussed memory error countermeasures, but after Mudge's notes and the well-known document by Elias Levy (Aleph One) on stack smashing [4], discussions on memory errors and protection mechanisms proliferated.

The introduction of the non-executable (NX) stack opened a new direction in the attack-defense arms-race as the first countermeasure to address specifically code injection attacks in stack-based buffer overflows. Alexander Peslyak (Solar Designer) released a first implementation of an NX-like system, StackPatch [34], in April 1997. We discuss NX in Section 2.1.

A few months later, in January 1998, Cowan et al. proposed placing specific patterns (canaries) between stack variables and a function's return address to detect corruptions of the latter [29]. Further details are discussed in Section 2.2.

After the first stack-based countermeasures, researchers started exploring other areas of the process address space—specifically the heap. In early 1999, Matt Conover and the w00w00 security team were the first to describe heap overflow exploitations [27], which we discuss in Section 2.3.

On September 20, 1999, Tymm Twillman introduced format string attacks by posting an exploit against ProFTPD on Bugtraq [101]. Format string exploits became popular in the next few years and we discuss them in Section 2.4.

The idea of adding randomness to prevent exploits from working (e.g., in StackGuard) was brought to a new level with the introduction of Address Space Layout Randomization (ASLR) by the PaX Team in July 2001. We discuss the various types of ASLR and its related attacks in Section 2.5.

Around the same time as the introduction of ASLR, another type of vulnerability, NULL pointer dereference, a form of dangling pointer, was disclosed in May 2001. Many assumed that such dangling pointers were unlikely to cause more harm than a simple denial of service attacks. In 2007 and 2008, however, Afek and Sharabani and Mark Dowd showed that these vulnerabilities could very well be used for arbitrary code injection as well [1,37]. Unfortunately, specific defenses against dangling pointers are still mostly research-driven efforts [2].

Due to space limitations, a number of historical details were omitted in this paper. The interested reader can refer to [102] for more information.

2.1 Non-executable Stack

Stack-based buffer overflows are probably the best-known memory error vulnerabilities [4]. They occur when a stack buffer overflows and overwrites adjacent memory regions. The most common way to exploit them is to write past the end of the buffer until the function's return address is reached. The corruption of this code pointer (making it point to the buffer itself, partially filled with attacker-injected code) allows the execution of arbitrary code when the function returns. A non-executable stack prevents such attacks by marking bytes of the stack as non-executable. Any attempt to execute the injected code triggers a program crash. The first non-executable stack countermeasure was proposed by Alexander Peslyak (Solar Designer) in June 1997 for the Linux kernel [34].

Just a few months after introducing the patch, Solar Designer himself described a novel attack that allows attackers to bypass a non-executable stack [33]. Rather than returning to code located on the stack, the exploit crafts a fake call stack mainly made of libraries' function addresses and arguments. Returning from the vulnerable function has the effect of diverting the execution to the library function. While any dynamically linked library can be the target of this diversion, the attack is dubbed *return-into-libc* because the return address is typically replaced with the address of proper C library functions (and arguments).

An enhancement of Solar Designer's non-executable stack was quickly proposed to withstand return-into-libc attacks [33]. However, shortly thereafter, Rafal Wojtczuk (Nergal) circumvented Solar Designer's refinement by taking advantage of specific ELF mechanisms (i.e., dynamic libraries, likely omnipresent functions, and dynamic libraries' function invocation via PLT) [73]. McDonald [66] built on such results and proposed return-into-libc as a first stage loader to run the injected code in a non-executable segment.

The PaX Team went far beyond a non-executable stack solution. With the PaX project released in the year 2000 [99], they offered a general protection against the execution of code injected in data segments. PaX prevents code execution on all data pages and adds additional measures to make return-into-libc much harder. Under PaX, data pages can be writable, but not executable, while code pages are marked executable but not writable. Most current processors have hardware support for the NX (non-executable) bit and if present, PaX will use it. In case the CPU lacks this feature, PaX can emulate it in software. In addition, PaX randomizes the mmap base so that both the process' stack and the first loaded library will be mapped at a random location, representing the first form of address space layout randomization (Section 2.5).

One of the first attacks against PaX ASLR was published by Nergal [73] in December, 2001. He introduced advanced return-into-libc attacks and exposed several weaknesses of the mmap base randomization. He showed that it is easy to obtain the addresses of libraries and stack from the Linux proc file system for a local exploit. Moreover, if the attacker can provide the payload from I/O pipes rather than the environment or arguments, then the program is exploitable.

OpenBSD version 3.3, released in May 2003, featured various memory error mitigation techniques. Among these, W^X proved to be effective against code-injection attacks. As a memory page can either be writable or executable, but never be both, injected code had no chances to get executed anymore.

In August 2005, Microsoft released the Service Pack 2 (SP2) of the Windows XP OS, featuring Data Execution Protection (DEP)—which prevents code execution of a program data memory [70].

By this time all major OSes were picking up on memory error mitigation techniques. NX stack was considered a strong protection against code-injection attacks and vendors soon backed up software implementations by hardware support for non-executable data. However, techniques like return-into-libc soon showed how NX can only partially mitigate memory errors from being exploited.

In 2005, Krahmer [58] pioneered short code snippet reuse instead of entire libc functions for exploit functionality—a direction that reached its zenith in return-oriented programming (ROP). Attackers chain code snippets together to create *gadgets* that perform predetermined but arbitrary computations [83]. The chaining works by placing short sequences of data on the stack that drive the flow of the program whenever a call/return-like instruction executes.

To date, no ROP-specific countermeasures have seen deployment in mainstream OSes. Conversely, low-overhead bounds checkers [3,111] and practical taint-tracking [16] may be viable solutions to defeat control-hijacking attacks.

2.2 Canary-Based Protections

Canaries represent a first line of defense to hamper classic buffer overflow attacks. The idea is to use hard-to-predict patterns to guard control-flow data. The first of such systems, StackGuard, was released on January 29, 1999 [29]. When entering a function, StackGuard places a hard-to-predict pattern—the canary—adjacent to the function's return address on the stack. Upon function termination, it compares the pattern against a copy. Any discrepancies would likely be caused by buffer overflow attacks on the return address and lead to program termination.

StackGuard assumed that corruption of the return address only happens through direct buffer overflows. Unfortunately, indirect writes may allow one to corrupt a function return address while guaranteeing the integrity of the canary. StackShield [96], released later in 1999, tried to address this issue by focusing on the return address itself, by copying it to a "secure" location. Upon function termination, the copy is checked against the actual return address. A mismatch would result in program termination.

StackShield clearly showed that in-band signaling should be avoided. Unfortunately, as we will see in the next sections, mixing up user data and program control information is not confined to the stack: heap overflows (e.g., dynamic memory allocator metadata corruption) and format bug vulnerabilities intermix (in-band) user and program control data in very similar ways.

Both StackGuard and StackShield, and their Windows counterparts, have been subject to a slew of evasions, showing how such defenses are of limited effect against skilled attackers [21,81]. On Windows, David Litchfield introduced a novel approach to bypass canary-based protections by corrupting specific exception handling callback pointers, i.e., structured exception handing (SEH), used during program cleanup, when a return address corruption is detected [61].

Matt Miller subsequently proposed a protection against SEH exploitation [71] that was adopted by Microsoft (Windows Server 2008 and Windows Vista SP1). It organizes exception handlers in a linked list with a special and well-known terminator that is checked for validity when exceptions are raised. As SEH corruptions generally make the terminator unreachable, they are often easy to detect. Unlike alternative solutions introduced by Microsoft [17], Miller's countermeasure is backward compatible with legacy applications. Besides, if used in conjunction with ASLR, it hampers the attackers' ability to successfully exploit SEH.

Despite their initial weaknesses, canary-based protection spun off more countermeasures. ProPolice, known also as Stack Smashing Protection (SSP), built on the initial concept of StackGuard but addressed its shortcomings [40]; stack variables are rearranged such that pointers corruptions due to buffer overflows are no longer possible. SSP was successfully implemented as a low-overhead patch for the GNU C compiler and was included in mainstream from version 4.1.

2.3 Protecting the Heap

While defense mechanisms against stack-based buffer overflow exploitations were deployed, heap-based memory errors were not taken into consideration yet.

The first heap-based buffer overflow can be traced to January 1998 [36], and the first paper published by the underground research community on heap-based vulnerabilities appeared a year later [27]. While more advanced heap-based exploitation techniques were yet to be disclosed, it nonetheless pointed out that memory errors were not confined to the stack.

The first description of more advanced heap-based memory error exploits was reported by Solar Designer in July 2000 [35]. The exploit shows that in-band control information (heap management metadata) is still the issue, a bad practice, and should *always* be avoided, unless robust integrity checking mechanisms are in place. Detailed public disclosures of heap-based exploitations appeared in [7,65]. Such papers dug into the intricacies of the System V and GNU C library implementations, providing the readers with all the information required to write reliable heap-based memory error exploits.

Windows OSes were not immune from heap exploitation either. BlackHat 2002 hosted a presentation by Halvar Flake on the subject [45], while more advanced Unix-based heap exploitation techniques where published in August 2003 [55], describing how to obtain a *write-anything-anywhere* primitive that, along with information leaks, allow for successful exploits even when ASLR is in use.

More about Windows-based heap exploitations followed in 2004 [62]. The introduction of Windows XP SP2, later that year, came with a non-executable heap. In addition, SP2 introduced heap cookies and safe heap management metadata unlink operations. Not long had to be waited for before seeing the first working exploits against Microsoft latest updates [26,6,67]. With the release of Windows Vista in January 2007, Microsoft further hardened the heap against exploitation [64]. However, as with the Unix counterpart, there were situations in which application-specific attacks against the heap could still be executed [51,107].

In 2009 and 2010 a report appeared where proof of concept implementations of almost every scenario described in [78] were shown in detail [12,13].

2.4 Avoiding Format String Vulnerabilities

Similarly to the second generation of heap attacks, but unlike classic buffer overflows, format string vulnerabilities (also known as format bugs) are easily exploited as a write-anything-anywhere primitive, potentially corrupting the whole address space of a victim process. Besides, format bugs also allow to perform *arbitrary* reads of the whole process address space. Disclosing confidential data (e.g., cryptographic material and secrets [29,98]), executing arbitrary code, and exploring the whole address space of a victim process are all viable possibilities.

Format string vulnerabilities were first discovered in 1999 while auditing ProFTPD [101], but it was in the next couple of years that they gained popularity. A format string vulnerability against WU-FTPD was disclosed on Bugtraq in June 2000 [20], while Tim Newsham was the first to dissect the intricacies of the attack, describing the fundamental concepts along with various implications of having such a vulnerability in your code.

One of the most extensive articles on format string vulnerabilities was published by Scut of the TESO Team [87]. Along with detailing conventional format string exploits, he also presented novel hacks to exploit this vulnerability.

Protection against format string attacks was proposed by FormatGuard in [28]. It uses static analysis to compare the number of arguments supplied to `printf`-like functions with those actually specified by the function's format string. Any mismatch would then be considered as an attack and the process terminated. Unfortunately, the effectiveness of FormatGuard is bound to the limits of static analysis, which leaves exploitable loopholes.

Luckily, format string vulnerabilities are generally quite easy to spot and the fix is often trivial. Moreover, since 2010, the Windows CRT disables `%n`-like directives by default. Similarly, the GNU C library `FORTIFY_SOURCE` patches provide protection mechanisms, which make format string exploitations hard. Even so, and although the low hanging fruit had been harvested long ago, the challenge of breaking protection schemes remains exciting [79].

2.5 Address Space Layout Randomization

Memory error exploitations typically require an intimate knowledge of the address space layout of a process to succeed. Therefore, any attempt to randomize that layout would increase the resiliency against such attacks.

The PaX Team proposed the first form of address space layout randomization (ASLR) in 2001 [99]. ASLR can be summarized succinctly as the introduction of randomness in the address space layout of userspace processes. Such randomness would make a class of exploits fail with a quantifiable probability.

PaX-designed ASLR underwent many improvements over time. The first ASLR implementation provided support for `mmap` base randomization (July 2001). When randomized `mmap` base is enabled, dynamically-linked code is mapped starting at a different, randomly selected base address each time a program starts, making return-into-libc attacks difficult. Stack-based randomization followed quickly in August 2001. Position-independent executable (PIE) randomization was proposed in the same month. A PIE binary is similar in spirit to dynamic shared objects as it can be loaded at arbitrary addresses. This reduces the risk of performing successful return-into-plt [73] or more generic return-oriented programming attacks [83] (see next). The PaX Team proposed a kernel stack randomization in October 2002 and, to finish their work, a final patch was released to randomize the heap of processes.

Over time, OSes deployed mostly coarse-grained—often kernel-enforced—forms of ASLR, without enabling PIE binaries. Such randomization techniques are generally able to randomize the *base* address of specific regions of a process address space (e.g., stack, heap, `mmap` area). That is, only starting base addresses are randomized, while relative offsets (e.g., the location of any two objects in the process address space) are fixed. Thus, an attacker's task is to retrieve the absolute address of a *generic* object of, say, a dynamically-linked library of interest: any other object (e.g., library functions used in return-into-libc attacks) can be reached as an offset from it.

One of the first attacks against ASLR was presented by Nergal in 2001 [73]. Although the paper mainly focuses on bypassing non-executable data protections, the second part addresses PaX randomization. Nergal describes a novel technique, dubbed return-into-plt, that enables a direct call to the dynamic linker's symbol resolution procedure, which is used to obtain the address of the symbol of interest. Such an attack was however defeated when PaX released PIE.

In 2002, Tyler Durden showed that certain buffer overflow vulnerabilities could be converted into format string bugs, which could then be used to leak information about the address space of the vulnerable process [38]. Such information leaks would become the de-facto standard for attacks on ASLR.

In 2004, Shacham et al. showed that ASLR implementations on 32-bit platforms were of limited effectiveness. Due to architectural constraints and kernel design decisions, the available entropy is generally limited and leaves brute forcing attacks a viable alternative to exploit ASLR-protected systems [90].

Finally, Fresi-Roglia et al. [47] detail a return-oriented programming [83] attack able to bypass W^X and ASLR. This attack chains code snippets of the original executable and, by copying data from the global offset table, is then able to compute the base addresses of dynamically linked shared libraries. Such addresses are later used to build classic return-into-libc attacks. The attack proposed is estimated to be feasible on 95.6% binaries for Intel x86 architectures (61.8% for x86-64 architectures). This high success rate is caused by the fact that modern OSes do not adopt or lack PIE.

A different class of attacks against ASLR protection, called heap spraying, was described first in October 2004 when SkyLined published a number of heap spraying attacks against Internet Explorer [91,92,93]. By populating the heap with a large number of objects containing attacker-provided code, he made it possible to increase the likelihood of success in referencing (and executing) it.

Heap spraying is mostly used to exploit cross-platform browser vulnerabilities. Since scripting languages like JavaScript and ActionScript are executed on the client's machine (typically in web browser clients), heap spraying has become the main infection vector of end-user hosts.

Dion Blazakis went far beyond heap spraying by describing pointer inference and JIT spraying techniques [14]. Wei et al. proposed dynamic code generation (DCG) spraying, a generalized and improved JIT spraying technique [108]. (Un)luckily DCG suffers from the fact that memory pages, which are about to contain dynamically-generated code, have to be marked as being writable *and* executable. Wei et al. found that all DCG implementations (i.e., Java, JavaScript, Flash, .Net, Silverlight) are vulnerable against DCG spraying attacks. A new defense mechanism to withstand such attacks was eventually proposed [108].

Finally, return-oriented programming, introduced in Section 2.1, may also be used to bypass non-PIE ASLR-protected binaries (as shown by [47]). In fact, for large binaries, the likelihood of finding enough useful code snippets to build a practical attack is fairly high. Recent work on protecting against these attacks involves instruction location randomization, in-place code randomization and fine-grained address space randomization [48,52,77].

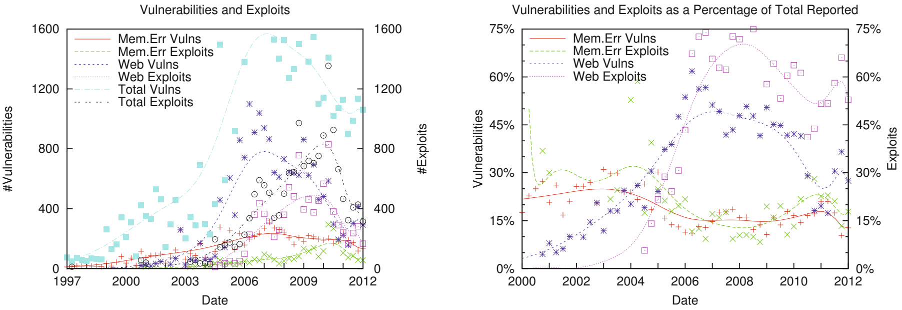
3 Data Analysis

We have analyzed statistics as well as real-life evidence about vulnerability and exploit reports to draw a final answer about memory errors. To this end, we tracked vulnerabilities and exploits over the past 15 years by examining the Common Vulnerabilities and Exposures (CVE) and ExploitDB databases.

Figure 2a shows that memory error vulnerabilities have grown almost linearly between 1998 and 2007 and that they started to attract attackers in 2003, where we witness a linear growth in the number of memory error exploits as well. Conversely, the downward trend in discovered vulnerabilities that started in 2007 is remarkable. Instead of a linear growth, it seems that the number of *reported* vulnerabilities is now reversed. It is worth noting that such a drop mirrors a similar trend in the *total* number of reported vulnerabilities. Figure 2b shows the same results as percentages.

The spike in the number of vulnerabilities that started in 2003 may well have been caused by the explosive growth of web vulnerabilities in that period (as supported by [24]).

Figure 2a shows that web vulnerabilities first appeared in 2003 and rapidly outgrew the number of buffer overflow vulnerabilities. Probably due to their simplicity, the number of working web exploits also exceeded the number of buffer overflow exploits in 2005. It is therefore plausible that the extreme growth in vulnerability reports that started in 2003 had a strong and remarkable web-related component. This seems to be reasonable as well: shortly after the dot-com bubble in 2001, when the Web 2.0 started to kick in, novel web developing technique were often not adequately tested against exploitation techniques. This was probably due to the high rate at which new features were constantly asked by end customers: applications had to be deployed quickly to keep up with competitors. This race left little time to carefully perform code security audits.



(a) Absolute (b) As % of totals

Fig. 2. Vulnerabilities and Exploits

As mentioned earlier, Figure 2a also shows a downward trend in the total number of vulnerabilities over the years 2006–2010, as reported independently in [68]. Memory errors were also affected by that trend and started to diminish in early 2007. Despite such negative fluctuations, generic and memory error-specific *exploits* kept growing linearly each month. The reasons for the downward trend of reported vulnerabilities could be manifold; it could be that fewer bugs were found in source code, fewer bugs were reported, or a combination thereof.

Assuming that the software industry is still growing and that, hence, the number of lines of code (LoC) written each month still increases, it is hard to back up the first hypothesis. More LoC results in more bugs: software reliability studies have shown that executable code may contain up to 75 bugs per 1000 LoC [9,76]. CVEs look at vulnerabilities and do not generally make a difference between plain vulnerabilities and vulnerabilities that could be exploited. Memory error mitigation techniques are unlikely to have contributed to the drop in reported vulnerabilities. Most defense mechanisms that we have discussed earlier do not result in safer LoC (for which static analysis can instead help [103]); they only prevent exploitation of vulnerable—usually poorly written—code.

However, if we look carefully at the data provided by analyzing CVE entries, as depicted in Figure 2a, we see that the number of web vulnerabilities follows the same trend as that of the total number of vulnerabilities. It seems that both the exponential growth (2003–2007) and drop (2007–2010) in vulnerabilities is related to fundamental changes in web development. It is reasonable to assume that companies, and especially web developers, started to take web programming more seriously in 2007. For instance, developers may well have become more aware of attacks such as SQL injections or Cross Site Scripting (XSS). If so, this would have raised web security concerns, resulting in better code written.

Similarly, static (code) analysis may have started to be included in the software development life-cycle. Static code analysis for security tries to find weaknesses in a program that might lead to security vulnerabilities. In general, the process is to evaluate a system (and all its components) based on its form, structure, content or documentation for non-conformance in access control, information flow, or application programming interface. Considering the high cost involved in manual code review, software industry prefers using automated code analyzers. IBM successfully demonstrated JavaScript analysis [109] by analysing 678 websites (including 178 most popular websites). Surprisingly, 40% of the websites were found vulnerable and 90% of vulnerable applications had 3rd party code. Another interesting take on static analysis can be observed in the surveys published by NIST [75,74]. A remarkable number of previously reported vulnerabilities were found in popular open-source programs using a combination of results reported by different tools. For instance, [103] reports intriguing figures showing the industry has confidence in static analysis. The software and IT industry are the biggest requester, followed by the finance sector for independent security assessments of their applications. Furthermore, 50% of the companies resubmitted 91–100% of their commercial application (after the first submissions revealed security holes) for code analysis. The growing trust of industry in static analysis could be one of the reasons for a drop in the number of reported vulnerabilities from the last few years.

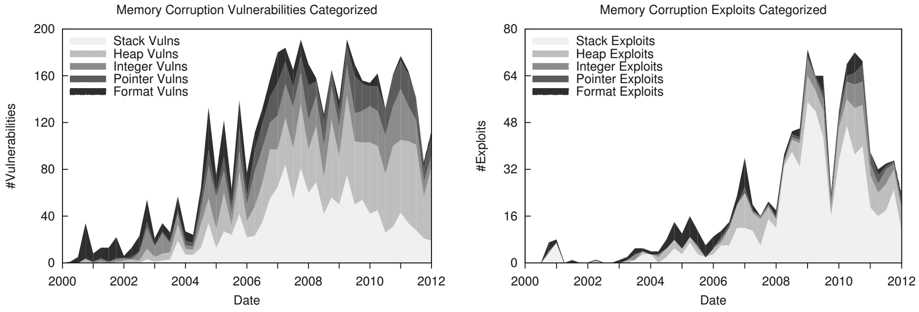
To substantiate the second hypothesis (i.e., less bugs are reported), it is necessary and helpful to have a more social view on the matter. There could be a number of reasons why people stopped reporting bugs to the community.

Empirical evidence about “no full disclosure due to bounties” advocates this statement very well. Ten years ago, the discovery of a zero-day vulnerability would have likely had a patch and a correspondence with the application authors/vendor about the fix, likely on public mailing lists. Today, big companies like Google and Mozilla give out rewards to bug hunters as long as they do not disclose the vulnerability [69]. Similarly, bug hunters may choose to not disclose zero day vulnerabilities in public, but sell them instead to their customers [43].

Where companies send out rewards to finders of vulnerabilities, useful zero-days could yield even more in underground and private markets. This business model suggests that financial profit may have potentially been responsible for the downward trend. While more and more people start buying things online and use online banking systems, it becomes increasingly interesting for criminals to move their activities to the Internet as well. Chances that issues found by criminals are reported as CVEs are negligible.

At the same time, full disclosure [113] as it was meant to be, is being avoided [50,44]. As an example for this shift in behavior, researchers were threatened for finding a vulnerability [49], or, as mentioned above, they may well sell them to third parties on private markets. This was also recently backed up by Lemos and a 2010-survey that looked at the relative trustworthiness and responsiveness of various organizations that buy vulnerabilities [60,42].

In conclusion, it is reasonable to believe that the drop in vulnerabilities is caused by both previous hypotheses. The software industry has become more mature during the last decade, which led to more awareness about what potential damage a vulnerability could cause. Web developers or their audits switched



(a) Vulnerabilities

(b) Exploits

Fig. 3. Memory errors categorized

to more professional platforms instead of their home-brew frameworks and eliminated easy vulnerabilities by simply writing better code. This growing professionalism of the software industry also contributed to the fact that bugs are no longer reported to the public, but sold to either the program's owners or the underground economy.

3.1 Categorizing Vulnerabilities and Exploits

We further categorized memory error vulnerabilities and exploits in 6 different classes (based on their CVEs descriptions): stack-based, heap-based, integer issues, `NULL` pointer dereference and format string. Figures 3a and 3b show the classification for vulnerabilities and exploits, respectively.

From Figures 3a and 3b we make the following observations which may help to draw our final conclusions. First, format string vulnerabilities were found all over the place shortly after they were first discovered. Over the years, however, the number of format string errors dropped to almost zero and it seems that they are about to get eliminated totally in the near future. Second, integer vulnerabilities were booming in late 2002, and, despite a small drop in 2006, they are still out there as of this writing (see [102]). Last, old-fashioned *stack* and *heap* memory errors are still by far (about 90%) *the most exploited* ones, counting for about 50% of all the reported vulnerabilities. There is no evidence to make us believe this will change in the near future.

4 Discussion

To answer the question whether memory errors have become a memory of the past, a few more observations need to be taken into consideration.

Impact. Let us first take a closer look at the impact of memory error vulnerabilities. After all, if this turns out to be negligible, then further research on the

Table 1. Breakdown of exploited vulnerabilities in popular exploit toolkits

Pack	Exploits	Memory Errors	Unspecified	Other	Updated
Nuclear	12	6 (50%)	3 (25%)	0 (0%)	Mar. 2012
Incognito	9	4 (44%)	1 (33%)	0 (0%)	Mar. 2012
Phoenix	26	14 (54%)	7 (27%)	5 (19%)	Mar. 2012
BlackHole	15	6 (40%)	7 (46%)	2 (13%)	Dec. 2011
Eleonore	31	18 (58%)	6 (19%)	7 (23%)	May. 2011
Fragus	14	11 (79%)	1 (7%)	2 (14%)	2011
Breeding Life	15	8 (53%)	6 (40%)	1 (6%)	?
Crimepack	19	10 (53%)	2 (11%)	7 (37%)	Jul. 2010
All	104	66 (63%)	12 (12%)	26 (25%)	

topic may just well be a questionable academic exercise. To provide a plausible answer, we analysed different exploit packs by studying the data collected and provided by contagio malware dump[1]. Table 1 shows the number of exploits (with percentages too) a given exploit pack supports and it is shipped with. The column Memory Errors reports those related to memory errors, while Unspecified refers to exploit of vulnerabilities that have not been fully disclosed yet (and chances are that some of these are memory error-related, e.g., CVE-2010-0842). Conversely, the column Other refers to exploits that are anything but memory error-related (e.g., an XSS attack).

Table 1 clearly shows that in at least 63% of the cases, memory error exploits have been widely deployed in exploit packs and have thus a large impact on the security industry, knowing that these exploit packs are responsible for large-scale hosts infections.

Support of Buffer Overflows by Design. Second, we observe that the number of memory vulnerabilities in a specific program is highly dependent on the programming language of choice. Looking closely at the C programming language, we observe that it actually needs to support buffer overflows. Consider, for example, an array of a simple C struct containing two fields, as depicted in Figure 4a. A `memset()`-like call may indeed overflow a record (Figure 4b). In this case, the overflow is not a programming error, but a desired action. Having such overflows by design, makes the programming language more vulnerable and harder to protect against *malicious* overflows. Considering that unsafe programming languages such as C and C++ are and have been among the most popular languages in the world for a long time already (as supported by the TIOBE Programming Community Indexes), careful attention should be paid by developers to avoid memory error vulnerabilities.

[1] `http://contagiodump.blogspot.com/2010/06/overview-of-exploit-packs-update.html`

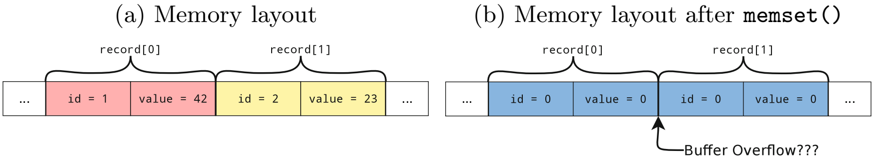


Fig. 4. Buffer overflow support in C

Deployment of Mitigation Techniques. Third, we observe that mitigation techniques are not always deployed by modern OSes. The reasons could be manifold. For instance, implementing a specific mitigation technique on legacy (or embedded) systems may require non-existent hardware support, or it may incur non-negligible overheads. Similarly, alternative mitigation techniques may require recompilation of essential parts of a system, which cannot be done due to uptime requirements or lack of source code.

Patching Behaviour. Another issue relates to the patching behaviour. Not only end users, but also system administrators appear to be lazy when it comes to patching vulnerabilities or updating to newer software versions. During our research on exploit kits, we found that even recently updated exploit packs still come with exploits for quite dated vulnerabilities, going back to 2006. This is backed up by other studies [59,100].

Motivation. Finally, even when all mitigation techniques are deployed, skilled and well-motivated attackers can still find their way into a system [41,104,106,105].

Looking back at Figure 2a, it is reasonable to say that some form of awareness has arisen among developers. On the other hand, Figure 2b shows that memory errors have a market share of almost 20%—a number that did not change much over the last 15 years, and something of which we have no evidence that it is about to change in the foreseeable future. The fact that over 60% of all the exploits reported in Table 1 are memory error-related, does not improve the scenario either.

4.1 Research Directions

Memory errors clearly still represent a threat undermining the security of our systems. We would like to conclude by sketching a few research directions that we consider both important and promising.

Information Leakage, Function Pointer Overwrites and Heap Inconsistencies. Information leakage vulnerabilities are often used to bypass (otherwise well-functioning) ASLR, enabling an attacker to initiate a return-into-libc or ROP attack [89]. Function pointer overwrites and heap inconsistencies form a class of vulnerabilities that cannot be detected by current stack smashing detectors and heap protectors. These vulnerabilities are often exploited to allow arbitrary code execution [105]. Recent studies try to address these problems by introducing more randomness at the operating system level[77,48,52] and, together with future research, will hopefully result in better protections against these classes of vulnerabilities [14].

Low-Overhead Bounds Checkers. Bounds checking aims at defining the boundaries of memory objects therefore avoiding overflows, which represent probably the most important class of memory errors. Although state-of-the-art techniques have drastically lowered the overhead imposed [111,3], the runtime and memory pressure of such countermeasures are still non-negligible.

Non-control Data Attacks. While ordinary attacks against control data are relatively easy to detect, attacks against non-control data can be very hard to spot. These attacks were first described by Chen et al. in [23], but a real-world scenario (an attack on the Exim mailserver), was recently described by Sergy Kononenko [57]. Although the attack uses a typical heap overflow, it does not get detected by NX/DEP, ASLR, W^X, canaries nor system call analysis, as it does not divert the program's control flow.

Legacy Systems and Patching Behaviour. As discussed earlier, lazy patching behaviour and unprotected legacy systems (as well as financial gain) are probably the main reasons of the popularity of exploit kits. Sophisticated patching schemes, and disincentivizing automatic patch-based exploit generation [18], should be the focus of further research.

Static and Dynamic Analysis to Detect Vulnerabilities. By using novel analysis techniques on vulnerable code, one may succeed in detecting vulnerabilities, and possibly harden programs before the application is deployed in a production environment. Research on this topic is ongoing [94,95] and may help to protect buffer overflow vulnerabilities from being exploited. However, it is necessary to extend these approaches to provide comprehensive protection against memory errors on production environments [46,30,19,84].

Sandboxing. To reduce the potential damage that an exploitable vulnerability could cause to a system, more research is needed on containment mechanisms. This technique is already widely adopted on different platforms, but even in its sophisticated forms (e.g., the Android's sandbox), it comes with weaknesses and ways to bypass it [31].

5 Conclusion

Despite half a century worth of research on software safety, memory errors are still one of the primary threats to the security of our systems. Not only is this confirmed by statistics, trends, and our study, but it is also backed up by evidence showing that even state-of-the-art *detection* and *containment* techniques fail to protect against motivated attackers [41,104]. Besides, protecting mobile applications from memory errors may even be more challenging [10].

Finding alternative mitigation techniques is no longer (just) an academic exercise, but a concrete need of the industry and society at large. For instance, vendors have recently announced consistent cash prizes to researchers who will improve on the state-of-the-art detection and mitigation techniques against memory error attacks [15], showing their concrete commitment towards a long-standing battle against memory error vulnerabilities.

References

1. Afek, J., Sharabani, A.: Dangling Pointer, Smashing the Pointer for Fun and Profit. In: Blackhat, USA (2007)
2. Akritidis, P.: Cling: A memory allocator to mitigate dangling pointers. In: Proceedings of the 19th USENIX Conference on Security (2010)
3. Akritidis, P., Costa, M., Castro, M., Hand, S.: Baggy bounds checking: an efficient and backwards-compatible defense against out-of-bounds errors. In: Proceedings of the 18th Conference on USENIX Security Symposium (2009)
4. Aleph: Smashing The Stack For Fun And Profit. Phrack Magazine (November 1996)
5. Anderson, J.P.: Computer Security Technology Planning Study, vol. 2 (October 1972)
6. Anisimov, A.: Defeating Microsoft Windows XP SP2 Heap protection and DEP bypass (January 2005)
7. Anonymous: Once Upon a Free. Phrack Magazine (August 2001)
8. Barrantes, E.G., Ackley, D.H., Forrest, S., Stefanovi, D.: Randomized instruction set emulation. ACM TISSEC (2005)
9. Basili, V.R., Perricone, B.T.: Software errors and complexity: an empirical investigation. CACM (1984)
10. Becher, M., Freiling, F.C., Hoffmann, J., Holz, T., Uellenbeck, S., Wolf, C.: Mobile security catching up? In: IEEE S&P (2011)
11. Bhatkar, S., Sekar, R., DuVarney, D.C.: Efficient techniques for comprehensive protection from memory error exploits. In: USENIX Security Symposium (August 2005)
12. blackngel: Malloc Des-Maleficarum. Phrack Magazine (June 2009)
13. blackngel: The House Of Lore: Reloaded. Phrack Magazine (November 2010)
14. Blazakis, D.: Interpreter Exploitation. In: Proceedings of the 4th USENIX Conference on Offensive Technologies (2010)
15. BlueHat, M.: Microsoft BlueHat Prize Contest (2011)
16. Bosman, E., Slowinska, A., Bos, H.: Minemu: The World's Fastest Taint Tracker. In: Sommer, R., Balzarotti, D., Maier, G. (eds.) RAID 2011. LNCS, vol. 6961, pp. 1–20. Springer, Heidelberg (2011)
17. Bray, B.: Compiler Security Checks In Depth (February 2002)
18. Brumley, D., Poosankam, P., Song, D., Zheng, J.: Automatic patch-based exploit generation is possible: Techniques and implications. In: Proceedings of the 2008 IEEE Symposium on Security and Privacy (2008)
19. Bruschi, D., Cavallaro, L., Lanzi, A.: Diversified Process Replicae for Defeating Memory Error Exploits. In: Intern. Workshop on Assurance, WIA (2007)
20. BugTraq: Wu-Ftpd Remote Format String Stack Overwrite Vulnerability (June 2000)

21. Bulba, Kil3r: Bypassing StackGuard and StackShield. Phrack Magazine (January 2000)
22. CERT Coordination Center: The CERT FAQ (January 2011)
23. Chen, S., Xu, J., Sezer, E.C., Gauriar, P., Iyer, R.K.: Non-control-data attacks are realistic threats. In: USENIX Sec. Symposium (2005)
24. Christey, S., Martin, R.A.: Vulnerability Type Distributions in CVE (May 2007)
25. Cker Chiueh, T., Hau Hsu, F.: Rad: A compile-time solution to buffer overflow attacks. In: ICDCS (2001)
26. Conover, M., Horovitz, O.: Windows Heap Exploitation (Win2KSP0 through WinXPSP2). In: SyScan (December 2004)
27. Conover, M.: w00w00 Security Team: w00w00 on Heap Overflows (January 1999)
28. Cowan, C., Barringer, M., Beattie, S., Kroah-Hartman, G.: FormatGuard: Automatic Protection From printf Format String Vulnerabilities. In: USENIX Security Symposium (August 2001)
29. Cowan, C., Pu, C., Maier, D., Hintongif, H., Walpole, J., Bakke, P., Beattie, S., Grier, A., Wagle, P., Zhang, Q.: StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks. In: Proceedings of the 7th USENIX Security Symposium (January 1998)
30. Cox, B., Evans, D., Filipi, A., Rowanhill, J., Hu, W., Davidson, J., Knight, J., Nguyen-Tuong, A., Hiser, J.: N-variant systems: a secretless framework for security through diversity. In: USENIX Security Symposium (2006)
31. Davi, L., Dmitrienko, A., Sadeghi, A.-R., Winandy, M.: Privilege Escalation Attacks on Android. In: Burmester, M., Tsudik, G., Magliveras, S., Ilić, I. (eds.) ISC 2010. LNCS, vol. 6531, pp. 346–360. Springer, Heidelberg (2011)
32. de Raadt, T.: Exploit Mitigation Techniques (in OpenBSD, of course) (November 2005)
33. Designer, S.: Getting around non-executable stack (and fix) (August 1997)
34. Designer, S.: Linux kernel patch to remove stack exec permission (April 1997)
35. Designer, S.: JPEG COM Marker Processing Vulnerability (July 2000)
36. DilDog: L0pht Advisory MSIE4.0(1) (January 1998)
37. Dowd, M.: Application-Specific Attacks: Leveraging the ActionScript Virtual Machine (April 2008)
38. Durden, T.: Bypassing PaX ASLR Protection. Phrack Magazine (July 2002)
39. Egele, M., Wurzinger, P., Kruegel, C., Kirda, E.: Defending Browsers against Drive-by Downloads: Mitigating Heap-Spraying Code Injection Attacks. In: Flegel, U., Bruschi, D. (eds.) DIMVA 2009. LNCS, vol. 5587, pp. 88–106. Springer, Heidelberg (2009)
40. Etoh, H., Yoda, K.: Protecting from stack-smashing attacks (June 2000)
41. Fewer, S.: Pwn2Own 2011: IE8 on Windows 7 hijacked with 3 vulnerabilities (May 2011)
42. Fisher, D.: Survey Shows Most Flaws Sold For $5,000 Or Less (May 2010)
43. Fisher, D.: Chaouki Bekrar: The Man Behind the Bugs (March 2012)
44. Fisher, D.: Offense is Being Pushed Underground (March 2012)
45. Flake, H.: Third Generation Exploits. In: Blackhat USA Windows Security (February 2002)
46. Flake, H.: Exploitation and State Machines: Programming the "weird machine" revisited (April 2011)
47. Fresi-Roglia, G., Martignoni, L., Paleari, R., Bruschi, D.: Surgically returning to randomized lib(c). In: ACSAC (December 2009)

48. Giuffrida, C., Kuijsten, A., Tanenbaum, A.S.: Enhanced Operating System Security Through Efficient and Fine-grained Address Space Randomization. In: Proceedings of the 21th USENIX Conference on Security (2012)
49. Goodin, D.: Legal goons threaten researcher for reporting security bug (2011)
50. Guido, D.: Vulnerability Disclosure (2011)
51. Hawkes, B.: Attacking the Vista Heap. Blackhat, USA (August 2008)
52. Hiser, J., Nguyen-Tuong, A., Co, M., Hall, M., Davidson, J.W.: ILR: Where'd My Gadgets Go? In: Proceedings of the 2012 IEEE Symposium on Security and Privacy (2012)
53. Jim, T., Morrisett, G., Grossman, D., Hicks, M., Cheney, J., Wang, Y.: Cyclone: A safe dialect of c. In: USENIX ATC (2002)
54. Jones, R.W.M., Kelly, P.H.J., Most, C., Errors, U.: Backwards-compatible bounds checking for arrays and pointers in c programs. In: Third International Workshop on Automated Debugging (1997)
55. jp: Advanced Doug lea's malloc exploits. Phrack Magazine (August 2003)
56. Kc, G.S., Keromytis, A.D., Prevelakis, V.: Countering Code-Injection Attacks With Instruction-Set Randomization (October 2003)
57. Kononenko, S.: Remote root vulnerability in Exim (December 2010)
58. Krahmer, S.: x86-64 buffer overflow exploits and the borrowed code chunks exploitation technique (September 2005)
59. Labs, M.S.: Security Labs Report, July - December 2011 Recap (Februay 2012)
60. Lemos, R.: Does Microsoft Need Bug Bounties? (May 2011)
61. Litchfield, D.: Defeating the Stack Based Buffer Overflow Prevention Mechanism of Microsoft Windows 2003 Server. In: Blackhat, Asia (December 2003)
62. Litchfield, D.: Windows Heap Overflows. In: Blackhat USA Windows Security (January 2004)
63. Lopatic, T.: Vulnerability in NCSA HTTPD 1.3 (Februay 1995)
64. Marinescu, A.: Windows Vista Heap Management Enhancements. In: Blackhat, USA (August 2006)
65. MaXX: VUDO Malloc Tricks. Phrack Magazine (August 2001)
66. McDonald, J.: Defeating Solaris/SPARC Non-Executable Stack Protection) (March 1999)
67. McDonald, J., Valasek, C.: Practical Windows XP/2003 Heap Exploitation. Blackhat, USA (July 2009)
68. Meer, H.: Memory Corruption Attacks The (almost) Complete History. In: Blackhat, USA (July 2010)
69. Mein, A.: Celebrating one year of web vulnerability research (2012)
70. 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)
71. Miller, M.: Preventing the Exploitation of SEH Overwrites (September 2006)
72. Necula, G.C., Condit, J., Harren, M., Mcpeak, S., Weimer, W.: Ccured: Type-safe retrofitting of legacy software. ACM Trans. on Progr. Lang. and Syst (2005)
73. Nergal: The Advanced Return-Into-Lib(c) exploits (PaX Case study). Phrack Magazine (December 2001)
74. NIST: The Second Static Analysis Tool Exposition (SATE) 2009 (June 2010)
75. Okun, V., Guthrie, W.F., Gaucher, R., Black, P.E.: Effect of static analysis tools on software security: preliminary investigation. In: Proceedings of the 2007 ACM Workshop on Quality of Protection (2007)
76. Ostrand, T.J., Weyuker, E.J.: The distribution of faults in a large industrial software system. In: ISSTA (2002)

77. Pappas, V., Polychronakis, M., Keromytis, A.D.: Smashing the Gadgets: Hindering Return-Oriented Programming Using In-Place Code Randomization. In: Proceedings of the 2012 IEEE Symposium on Security and Privacy (2012)
78. Phantasmagoria, P.: The Malloc Maleficarum (October 2005)
79. Planet, C.: A Eulogy for Format Strings. Phrack (November 2010)
80. Polychronakis, M., Anagnostakis, K.G., Markatos, E.P.: Comprehensive shellcode detection using runtime heuristics. In: ACSAC (2010)
81. Richarte, G.: Four different tricks to bypass StackShield and StackGuard protection (June 2002)
82. Ruwase, O., Lam, M.: A practical dynamic buffer overflow detector. In: Proceedings of NDSS Symposium (February 2004)
83. Roemer, R., Erik Buchanan, H.S., Savage, S.: Return-Oriented Programming: Systems, Languages, and Applications. ACM TISSEC (April 2010)
84. Salamat, B., Jackson, T., Gal, A., Franz, M.: Orchestra: Intrusion Detection Using Parallel Execution and Monitoring of Program Variants in User-Space. In: EuroSys (2009)
85. SANS: CWE/SANS TOP 25 Most Dangerous Software Errors (June 2011)
86. Schmidt, C., Darby, T.: The What, Why, and How of the 1988 Internet Worm (July 2001)
87. Scut: Exploiting Format String Vulnerabilities (September 2001)
88. Seifried, K., Levy, E.: Interview with Elias Levy (Bugtraq) (2001)
89. Serna, F.J.: CVE-2012-0769, the case of the perfect info leak (February 2012)
90. Shacham, H., Page, M., Pfaff, B., Goh, E.J., Modadugu, N., Boneh, D.: On the Effectiveness of Address-Space Randomization. In: ACM CCS (October 2004)
91. SkyLined: Internet Exploiter 3: Technical details (November 2004)
92. SkyLined: Internet Explorer IFRAME src&name parameter BoF remote compromise (October 2004)
93. SkyLined: Microsoft Internet Explorer DHTML Object handling vulnerabilities (MS05-20) (April 2005)
94. Slowinska, A., Stancescu, T., Bos, H.: Howard: a dynamic excavator for reverse engineering data structures. In: Proceedings of NDSS 2011, San Diego, CA (2011)
95. Slowinska, A., Stancescu, T., Bos, H.: Body armor for binaries: preventing buffer overflows without recompilation. In: Proceedings of the USENIX Security Symposium (2012)
96. StackShield: Stack Shield: A "stack smashing" technique protection tool for Linux (December 1999)
97. Symantec: Symantec report on the underground economy (2008)
98. Team, P.: Address Space Layout Randomization (March 2003)
99. The Pax Team: Design & Implementation of PAGEEXEC (2000)
100. Theriault, C.: Why is a 14-month-old patched Microsoft vulnerability still being exploited? (February 2012)
101. Twillman, T.: Exploit for proftpd 1.2.0pre6 (September 1999)
102. van der Veen, V., dutt Sharma, N., Cavallaro, L., Bos, H.: Memory Errors: The Past, the Present, and the Future. Technical Report IR-CS-73 (November 2011)
103. Veracode: State of Software Security Report, vol. 4 (December 2011)
104. VUPEN: Safari/MacBook first to fall at Pwn2Own (March 2011)

105. VUPEN: Pwn2Own 2012: Google Chrome browser sandbox first to fall (March 2012)
106. VUPEN: Pwn2Own 2012: IE 9 hacked with two 0day vulnerabilities (March 2012)
107. Waisman, N.: Understanding and Bypassing Windows Heap Protection (June 2007)
108. Wei, T., Wang, T., Duan, L., Luo, J.: Secure dynamic code generation against spraying. In: ACM CCS (2010)
109. X-Force, I.: IBM X-Force 2011 Mid-year Trend and Risk Report (September 2011)
110. Younan, Y., Joosen, W., Piessens, F.: Code injection in C and C++: A Survey of Vulnerabilities and Countermeasures. Technical Report CW386 (July 2004)
111. Younan, Y., Philippaerts, P., Cavallaro, L., Sekar, R., Piessens, F., Joosen, W.: PAriCheck: an efficient pointer arithmetic checker for c programs. In: AsiaCCS (2010)
112. Zatko, P.: How to write Buffer Overflows (1995)
113. Zetter, K.: Three minutes with rain forrest puppy (2001)

A Memory Access Validation Scheme against Payload Injection Attacks

Dongkyun Ahn[1] and Gyungho Lee[2]

[1] University of Illinois at Chicago, Chicago, IL 60607, USA
dahn6@uic.edu
[2] Korea University, Seoul, Korea
ghlee@korea.ac.kr

Abstract. The authenticity of a piece of data or an instruction is crucial in mitigating threats from various forms of software attacks. In spite of various features against malicious attacks exploiting spurious data, adversaries have been successful in circumventing such protections. This paper proposes a memory access validation scheme that manages information on spurious data at the granularity of cache line size. A validation unit based on the proposed scheme answers queries from other components in the processor so that spurious data can be blocked before control flow diversion. We describe the design of this validation unit as well as its integration into the memory hierarchy of a modern processor and assess its memory requirement and performance impact with two simulators. The experimental results show that our scheme is able to detect the synthesized payload injection attacks and to manage taint information with moderate memory overhead under acceptable performance impact.

Keywords: Memory access validation, Code injection attack, Return-to-libc attack, Return-oriented programming, Information flow tracking.

1 Introduction

When intruding into vulnerable systems, malicious parties usually inject their payload over a communication channel like a network device into a victimized process' address space. Payloads crafted for such attacks generally consist of machine code combined with control flow data in code injection attacks or control flow data followed by arguments for the existing procedure pointed to by the data, as can be seen in *return-to-libc* attacks. Despite architectural features and software mitigation approaches against such attacks, malicious parties have been able to bypass such techniques. Usually, such circumvention techniques take advantage of the inherent limitations of the base features on which those features and mitigation approaches are based - such as the coarse granularity of access control attributes or a limitation in randomization.

One noteworthy observation on exploited vulnerabilities is that control flow data that is vulnerable to compromise is still referenced for the next instruction address without any validation. For example, return addresses in an active

D. Balzarotti, S. Stolfo, and M. Cova (Eds.): RAID 2012, LNCS 7462, pp. 107–126, 2012.

stack frame are blindly referenced for the next instruction address upon exiting a sub-procedure. As is widely known, there are several mitigation approaches against stack-compromising attacks - inserting canaries [1], stack-layout reorganization, Return-Address encryption [2][3], Stack-frame allocation [4], and ASLR against return-to-libc. However, most of those protection measures still allow the processor core to fetch the next instruction address from vulnerable stack frames without verifying the authenticity of the memory word to be referenced. Stack-compromising attacks exploit this blind behavior thereby diverting control flow into locations that they want to fetch instructions from. Although ASLR approaches can mitigate threats from such attacks, as shown in [5], such mitigations that merely reduce the likelihood through randomization can be easily circumvented. Furthermore, return-oriented-programming (ROP) attacks [6] are able to synthesize a viable attack vector only with existing machine codes in a given address space. These observations imply that more complete and solid protection for control flow integrity is required.

In this paper, we propose a memory access validation scheme. The validation unit based on this scheme gathers information on memory blocks containing spurious data transferred through unreliable I/O devices, and updates the information to reflect status changes in those blocks at runtime. At the same time, the processor components related to control flow redirection ask this unit whether the memory block to be referenced for the next instruction or its address is authentic or not. The unit returns the taint status of the queried memory address to the the component for further actions like the triggering of exceptions.

Contribution. Our contribution made in this work is as follows.

- We propose a memory access validation scheme supporting the virtual memory systems of multi-tasking environments. Existing hardware components involved in control flow redirection query the validation unit authenticity of a memory block to be referenced for the next instruction or its address.
- For this validation scheme, we discuss practical issues on Information Flow Tracking (IFT) with regard to memory hierarchy. Based on our observations, we introduce a two-level approach design for memory access validation that supports small granularity as well as virtual memory systems.
- We propose two storage formats for each level of our validation scheme, and these formats are organized to support our design. We also propose integration approaches and a caching structure so as to alleviate runtime overhead.
- We apply our validation scheme to two simulators and show that our validation scheme is able to thwart the synthesized payload injection attacks and to manage taint information on memory blocks containing spurious data with low false-positive rates and 12.3 percent performance overhead.

This paper is organized as follows. We first describe the background and motivation for our validation scheme in Section 2. Section 3 describes the memory access validation scheme with taint information storage formats. We evaluate the proposed scheme in Section 4. Section 5 discusses various aspects of our validation scheme. Finally, this paper concludes in Section 6.

2 Background and Motivation

2.1 Control-Flow Integrity

Control-Flow Integrity proposed by Martín et al [7] enforces software execution to follow a path of a *Control-Flow Graph* (CFG) determined ahead of time. The CFI approach instruments vulnerable indirect branch instructions and their target locations with IDs and associated ID-check routines. Before invocation of an indirect branch, the instrumented ID-check routine compares the ID hard-coded prior to the branch with the ID marked in the target location. This comparison verifies that the control flow redirection conforms the CFG. If two IDs match, the program execution contiues to the target location. Otherwise, the ID-check routine invokes thwarting procedure. Because the ID-check routines are written with the same instruction set as the protected program, no modification is required at both the OS kernel and the processor hardware level.

Although the CFI implementation instruments only machine code, nearly half of the instrumented codes are referenced as data by the ID-check routines before instruction fetch. This is because the ID-check routine accesses the target location as data to read the ID marked in the location before control flow redirection. In viewpoint of the memory hierarchy, at least one data cache line must serve this ID check while an ID occupies only a small portion of one cache line (4 Bytes out of 32 or 64 Bytes). This inefficient data cache utilization could result in high data cache miss rates.

2.2 Spurious Raw Data in Attacks

Attacks and Spurious Raw Data. Malicious parties have devised various circumvention techniques and upgraded their payloads against many counter measures. Most of these circumvention techniques exploit limitations or vulnerabilities of the base feature of the protection measures as discussed previously. One noteworthy observation of malicious payloads is that they are injected into the victimized process' address space through unreliable I/O at runtime. However, simply incorporating a validation mechanism referring to attributes like "tainted" into existing functionalities - especially the address translation features of a virtual memory system - would not be desirable. This is because of limitations like the coarse granularity of the virtual memory system or runtime overhead from the OS kernel's frequent intervention for attribute management.

Figure 1 illustrates the coarse granularity problem. While a page frame size is usually 4K Bytes and access control attributes are applied for each page frame, stack frame sizes are usually much smaller than the page frame size and vary depending on invoked sub-procedures. In our simulations, most stack frames were smaller than 1K Bytes and varied from four Bytes to 8K Bytes. These results mean that an attribute value representing authenticity of a page frame would not be able to differentiate (b) from (c) as well as (d).

The most fundamental countermeasure that is able to address these inherent limitations appears to be the Information Flow Tracking (IFT) protection. In the

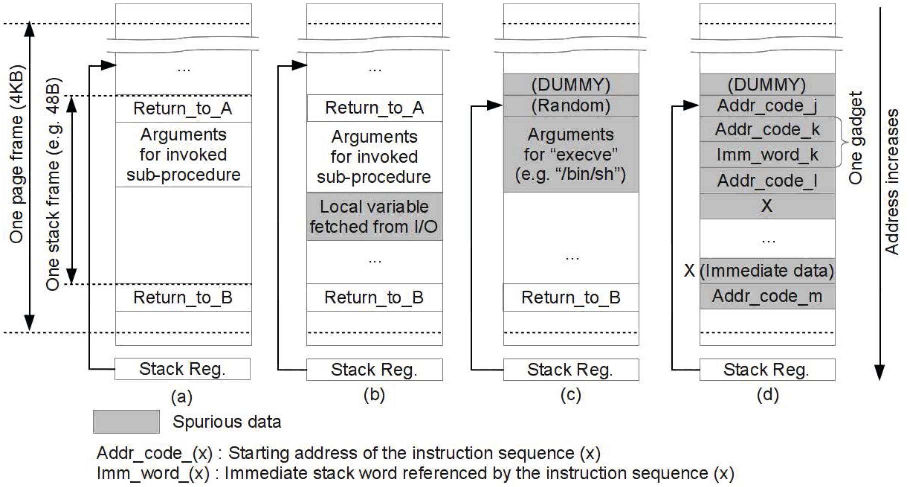

One page frame (4KB)
One stack frame (e.g. 48B)
...
Return_to_A
Arguments for invoked sub-procedure
Return_to_B
Stack Reg.
(a)
Local variable fetched from I/O
(b)
(DUMMY)
(Random)
Arguments for "execve" (e.g. "/bin/sh")
(c)
Addr_code_j
Addr_code_k
Imm_word_k
Addr_code_l
X
X (Immediate data)
Addr_code_m
One gadget
(d)
Address increases
Spurious data
Addr_code_(x) : Starting address of the instruction sequence (x)
Imm_word_(x) : Immediate stack word referenced by the instruction sequence (x)

Fig. 1. Active stack region and spurious data (a) normal active stack frame without spurious data / (b) Safe stack frame containing spurious data / (c) Stack frame overflowed by brute-force return-to-libc attack / (d) Stack frame overflowed by return-oriented-programming attack

following section, we briefly explore IFT-based protections and discuss practical issues in its integration into the memory hierarchy of modern microprocessors.

Information Flow Tracking. Information flow tracking (IFT) traces spurious data originating from unreliable sources like I/O devices throughout program execution. If spurious data or any data derived arithmetically from spurious data is referenced for control flow (such as indirect jumps), the processor triggers a trap with regard to the security policies configured by the OS kernel. Hardware-based IFT protections [8][9] augment taint bits (or tags) in registers as well as memory blocks in main memory, and some of them assigns a dedicated bus to exchange taint information across storage elements in the target system.

Minos [8] focuses on preventing attacks compromising control flow data and integrating taint tags into every storage element of a targeted architecture. Minos stores 1-bit taint tag to every 32-bit word as 33rd bit from the memory system level to the register level in the processor core. This approach accommodates synchronization between taint tags and their target memory words. However, such synchronization requires non-trivial changes in the memory access interface like a separate DRAM to contain taint tags and the widened memory bus for taint tags.

Suh et al [9] proposes a multi-granularity taint tags approach that contains taint status data in main memory. This approach can alleviate memory overhead from taint status data because one taint bit can represent taint statuses of multiple Bytes - up to a page frame size. In multi-granularity tags approach, the

OS kernel must allocate memory space for security tags when a memory block in a newly allocated frame gets tainted. The authors claim that granularity switching from one to another (for example, all-tainted-page to one-Byte granularity) occurs infrequently therefore the performance overhead is insignificant.

Raksha [10] investigates a hybrid IFT approach that takes advantage of both hardware-based and software-based techniques in order to achieve low-runtime overhead. A variation of Raksha is proposed by the same authors [11]. This approach decouples IFT functionality onto a dedicated coprocessor, thereby not requiring modifications in the design or layout of the main processor.

Software-based approaches utilize binary translation and an emulation environment to mark and trace taint data at runtime [12][13]. Unlike hardware-based IFT protections, software-based approaches dynamically translate machine code into other code instrumented with taint tracking instructions, while the existing hardware platform is utilized as is. However, the runtime overhead from instrumenting machine code could be considerable - even up to a forty fold slow down with a caching structure for translated machine codes. In [14], the authors present an interesting observation that most taint propagations are zero-to-zero, which means safe data sources to a safe destination. They take advantage of this observation to skip machine code instrumentation for basic blocks doing safe-to-safe memory transactions, thereby can reducing overhead significantly.

Our work proposes a hardware-based memory access validation scheme, therefore we continue to discuss technical issues related to hardware-based IFT works. In this paper, IFT will refer to hardware-based IFT unless otherwise noticed.

2.3 Motivation

Memory Hierarchy and IFT. In general, IFT approaches track propagation of spurious data from unreliable I/O at a fine granularity - like one byte - and trigger an exception upon detecting suspicious activity with regard to their security policies. However, the memory systems of modern microprocessors are designed to access memory in fixed-size blocks. The first problem regarding the taint tags of IFT approaches is how to store them. Obviously, augmenting every byte of the whole system with a one-bit tag is not a desirable approach because it would impose 12.5 percent storage overhead overall (1-bit for every byte).

The second problem is how to incorporate taint bit management into the existing memory hierarchy. In most microprocessor architectures, cache line managements are transparent to the software level. In the meantime, memory page frame management depends heavily on the OS kernel - from simple allocation/deallocation to sophisticated handlings like demand paging and page swapping. In such environments, incorporating a propagation-tracking functionality into existing cache controllers and the paging unit would require non-trivial and invasive modifications in both the hardware and software domains.

The third problem is about synchronization of taint information and its target memory word. Assuming that taint bits per sub-cache-line are not augmented at each byte but grouped in one block, those taint bits are just instances of special purpose data structure accessed by the IFT unit. This suggests that management of the taint status data be synchronized with that of the corresponding cache lines throughout the cache hierarchy. Such synchronization pertains to not only the cache level but also the virtual memory system level - this is because of the address space management of multi-tasking OS kernels. Like the OS kernel manages the address translation information for each address space and the paging unit traverses translation entries, the OS kernel allocates memory space for taint status data for one address space, and the IFT unit accesses it for taint status update without accessing another space's data.

Our Goal. The vulnerabilities exploited by malicious parties and the observations presented in the previous section provide us with a different view on how to handle information on spurious data and how to utilize them.

First of all, we choose memory accesses with fine granularity as our target objects for both taint status management and validations referring to taint status data. Validating memory accesses has several advantages in implementation and integration. For example, various events triggered in memory hierarchy - like cache misses - could be utilized for validation invocation.

Second, the observations on the memory hierarchy of modern microprocessors suggest that a memory access validation scheme should cope with a multi-tasking environment and support a virtual memory system. This design goal is very important, because (a) taint status would be contained in main memory to support arbitrary number of active processes and (b) taint status data sets for each address space must be strictly separated from one another to prevent the cross-contamination of taint statuses. We try to minimize modifications required to satisfy this design goal by utilizing existing hardware components and simple data storage formats.

Finally, these two design goals must be accomplished with a reasonable performance overhead. To minimize the performance overhead of a memory access validation mechanism, we take various aspects into consideration - the existing memory architecture, taint status management in main memory, and a caching structure for taint status data.

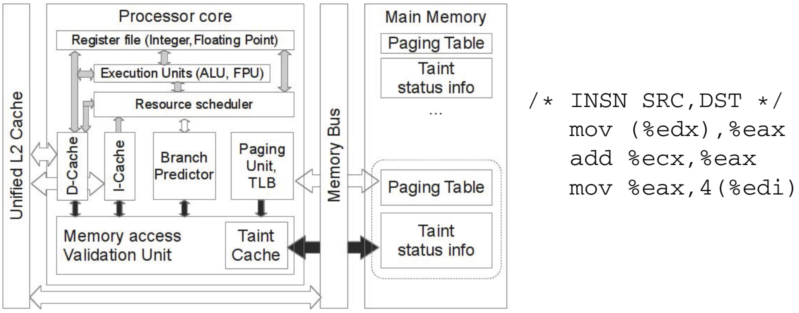
In summary, **our goal** in this paper is to propose a memory access validation scheme that prevents spurious data from influencing control flow redirection and supports the virtual memory system of multi-tasking environments.

3 Mechanism

3.1 Overview

Our validation scheme consists of two major parts: the validation unit in the processor and the taint status information in main memory. Figure 2 illustrates the overview of our validation scheme.

The validation unit is a hardware unit that gathers, updates, and refers to taint statuses of memory blocks in a program's address space. This unit gathers



```
/* INSN SRC,DST */
   mov (%edx),%eax
   add %ecx,%eax
   mov %eax,4(%edi)
```

(a) Overview of our validation scheme

(b) Taint propagation sample code for x86-32

Fig. 2. Overview block diagram and taint propagation sample code

information on memory blocks containing spurious data by monitoring I/O operations between user applications and the OS kernel. If a memory block no longer contains spurious data, the validation unit updates the gathered information so as to prevent a false alarm. Other hardware components involved in control flow redirection query this validation unit whether a memory block to be referenced is spurious or not. If so, this unit triggers an exception for the OS kernel to terminate the process under attack. Otherwise, the program execution resumes. All of these operations are triggered asynchronously by hardware events like cache misses and are transparent to software modules including the OS kernel.

The major difference between this approach and IFT approaches is that the proposed scheme validates only memory accesses directly related to control flow redirections, while taint propagation through arithmetic operations and pointer-based memory accesses are left untracked.

In Figure 2, dark gray arrows show the control/signal paths of the proposed scheme. As these arrows indicate, the signals to/from our validation unit do not go beyond L1 caches or the branch predictor. Meanwhile, the light gray arrows stand for the signal path utilized by IFT approaches. We use the sample code given with the block diagram for this taint propagation. When the processor core executes the first instruction, the IFT unit must merge the taint status of the `edx` register and that of the memory word pointed to by the same register and must transfer the merged taint status to the augmented taint bit of the `eax` register in the register file (the D-cache to the register file propagation). The unit must merge the taint tag of the `ecx` to that of the `eax` register at the second instruction (register to register propagation). Finally, the taint statuses of the two registers - `eax` and `edi` - must be merged and saved into the taint tag of the memory word indirectly pointed to by the `edi` register with a 4-Byte offset (the register file to the D-cache propagation). To support propagations like this sample code, the resource scheduler must be modified to buffer the taint tags associated with pending instructions for pipelining. This change would increase implementation complexity and power consumption.

Our validation scheme takes a rather simplified approach compared to other IFT approaches by not tracking propagation beyond the caches or the branch predictor - in other words, omitting the light gray arrows in Figure 2. This approach has advantages in simpler implementations and fewer invocations for validation at the loss of fine-grained protection available in IFT approaches. Because our scheme does not communicate with the resource scheduler, the processor's pipeline would not require modifications for buffering taint tags. Meanwhile, the dominance of safe-to-safe memory transactions observed and adopted for optimization in [14] substantiates our approach in terms of the simplification of taint status reference and tracking. According to the authors, in case of an Apache web server, about 98 percent of tag propagations are from the safe sources to safe destinations, and others like unsafe-to-safe account for less than 2 percent combined. We note that our justification for this simplification does not imply that fine-grained tracking in IFT is unnecessary and admit that our approach has limitations caused by this simplification. Section 5 discusses this issue further.

3.2 Taint Status Information Organization

Fine Granularity and Memory Page Frame. Like other IFT approaches, it is essential to support a granularity smaller than a page frame size in a memory access validation scheme (Figure 1). Moreover, the OS kernel allocates/deallocates memory page frames on the fly, and access attempts to unallocated address ranges trigger exceptions. Based on these circumstances, we propose a two-level approach for our validation scheme - the first level dealing with page frames and the second level dealing with the memory blocks in the page frames listed in the first level. Only if our validation unit finds that the page frame that a queried memory block belongs to contains spurious data, would it proceed to the next level dealing with memory blocks in the page frame. Otherwise, the queried memory block is considered “authentic” and instruction execution continues. In this work, we have chosen virtual address space as our reference address space to support multi-tasking OS kernels.

Matrix Format for Page Frame Numbers. For the first level of taint status data, the proposed scheme manages page frame numbers (PFNs) of page frames containing spurious data in matrix format - like an array of arrays. Each PFN is placed in its assigned sub-array - in matrix term a “row”, and specified bit fields of a PFN are used as the row index. There are two benefits of this format; (a) the validation unit can manage PFNs like cache lines in set-associative caches with replacement policies; (b) whether or not a page frame contains spurious data can be quickly determined by doing a linear search over its assigned row.

We design our validation scheme to manage fixed number of PFNs in each matrix - 32, 64, 128, or 256. This estimation is based on our experimental result with I/O-intensive applications (GCC toolchain, Apache web server, and sshd server) that more than 99.8 percent of I/O-active address spaces utilize fewer than 256 pages for I/O. The I/O-active address space in this work refers to an address space that directly communicates with I/O devices through system calls like `read` during execution time.

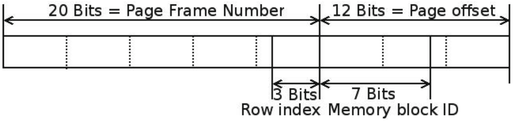


Fig. 3. Row index and memory block ID assignment example in a virtual address - 32-bit architecture with 4K Bytes page frame and 32-Byte cache line, $8 \times N$ page frame number matrix

Validation Granularity and Taint Status Information in Bitmap Format. In this work, we chose the cache line size for validation granularity. Managing the taint status at the granularity of cache line size has several advantages; (a) cache hit/miss statuses could be utilized for taint status management; (b) we could put only validated memory blocks into a cache so that cache lines in the cache no longer needed to be validated. The limitation from this coarse granularity compared to IFT approaches will be discussed in Section 5.3.

In this work, we chose the bitmap format for our second level storage format. Each bit of a bitmap indicates the taint status of the corresponding memory block (with cache line size) in a page frame that is listed in the associated PFN matrix. Because there are 32, 64, 128, or 256 PFNs in a PFN matrix, the same number of bitmaps for those PFNs will be stored in an array. For simplicity, we designed a bitmap array to be placed right next to the PFN matrix.

Matrix/Bitmap Location. Both PFN matrices and bitmap arrays must be stored in main memory and are traversed by our validation unit at runtime. In this work, we propose to concatenate our taint data (one PFN matrix and one bitmap) to one of the paging tables of a virtual address space to be protected. Such an augmented data structure facilitates synchronization of address space changes and taint status data switching. In other words, our validation unit would be able to locate the active PFN matrix and its bitmap array without an additional special purpose register for taint status data.

The location of the queried memory block's taint status is determined as follows. As explained previously, each PFN is placed in its assigned row and specified bit fields of a PFN are used as the row index in the matrix. Because each bitmap in the bitmap array corresponds to a PFN in the matrix, we use the coordination of a PFN in the matrix to calculate the index for the associated bitmap stored in the bitmap array. For the memory block level, we use the MSBs of the page offset of the memory block's address as the identifier within a page frame. Figure 3 shows an example of row assignment and memory block ID extraction.

Memory Space Requirement. The memory requirement for two taint information for one address space - one PFN matrix and one bitmap array - can be calculated with the following equation (CLS : Cache line size / PGN : Number of page frame numbers / BMP : Bitmap).

$$\underbrace{4 \times PGN}_{PFNMatrix} + \underbrace{\frac{4096}{CLS} \div 8 \times PGN}_{BMParray} \tag{1}$$

For the matrix structure, we assign one 32-bit word for PFN matrix element and utilize higher bit fields (20 bits) for PFN and remaining lower bit fields for other purposes - like checksums for PFNs and bitmaps. Depending on the matrix configuration, memory consumption of these storage formats could go up to 5K Bytes (32-Byte cache line / 256 PFNs) or 3K Bytes (64-Byte cache line / 256 PFNs) per address space. In case of the 32-bit x86 architecture, at least 8K Bytes are required to initialize one virtual address space, and this size increases by 4K Bytes for every 4M-Byte newly allocated address range. This means that the memory overhead for address space initialization would account for 62.5%/37.5% and would decrease as more page frames are allocated.

We believe that our approach in taint status data augmentation has three advantages as follows; (a) the memory access interface does not require non-trivial modifications like Minos because taint status data is contained in main memory; (b) hardware designs required for implementation - like LRU, bitmap - are already available; and (c) taint status data needs to be allocated and initialized only at once at the initialization of a protected address space.

3.3 Taint Operation

There are three operations to be processed on taint information - Paint, Erase, and Lookup. If a requested PFN is found in the PFN matrix, the Paint operation sets the bit field representing the taint status of a memory block that is written with spurious data transferred via I/O. As user applications request I/O operations to its O/S kernel through the system call interface, the straightforward approach to activating the Paint operations is to make the hardware unit handling system calls invoke the Paint operation with arguments like the starting address of a I/O buffer and its size. The Erase operation clears the bit field that corresponds to a memory block being overwritten with data from a register. The Lookup operation answers validation queries issued from other hardware components by referring to the bit field representing the taint status of a requested memory block.

The procedure after failing to locate a requested PFN in the matrix varies depending on the requested operation. Only the Paint operation puts the PFN into its assigned row in the matrix (either by finding an available spot or replacing an existing one) and initializes the linked bitmap in the bitmap array. If the Erase or Lookup operation fails to locate the PFN in the matrix, the validation unit returns a miss. The hardware component that issued the query ignores the response and resumes execution. Note that frequent misses do not necessarily mean that this matrix structure is unsuitable for managing PFNs. This is because queries that result in misses certainly include ones targeting memory pages that have never been involved in an I/O operation.

3.4 Integration into Memory Hierarchy and Return Address Stack

The basic design of this validation scheme is highly likely to incur significant overhead as every memory write will request a taint status update (with the Erase operation) in the taint information contained in main memory. In addition, frequent validation queries will also increase the performance overhead.

Our approach against overhead is to invoke the proposed validation scheme in cache miss handlers and a branch predictor. When a dirty cache line in a data cache is to be replaced with a missed cache line, a write-back procedure stores the victimized line into a next level cache before read-in of the missed cache line. By requesting the Erase operation for a replaced dirty cache line in the data cache miss handler, we can reduce the number of Erase requests.

Unlike the Erase operation, validation queries must be selectively issued. This is because validating every memory read will trigger unnecessary exceptions for memory accesses to spurious but harmless data like (b) in Figure 1. In this work, we propose two validation query points to counter two types of payload injection attacks - code injection attacks and stack-compromising attacks.

In order to counter code injection attacks, we design the instruction cache miss handler to issue validation queries. Because the injected payload(s) for a code injection attack has never been accessed for instruction fetch before, an instruction cache miss will be observed at an instruction fetch attempt to the payload after control flow redirection. One thing to be aware of with regard to this utilization approach is that, among cache miss handlers, only an instruction cache miss handler should issue validation queries.

In modern processor architectures, unified caches are widely adopted as the high-level cache - like a unified L2 cache. In general, a unified cache is larger than a low-level instruction cache and a data cache combined. This means that an instruction cache miss at a low-level cache could be served with a cache line of the high-level unified cache. In this case, the miss handler of the unified cache wouldn't be invoked for a cache line containing spurious data. If we want to utilize a unified L2 cache for access validation, we need an additional procedure for the cache controller; this procedure must distinguish hit-miss statuses with regard to the purpose of a memory access and issue a validation query accordingly.

The second query point is at a miss prediction of the Return Address Stack (RAS), and this is for countering two types of stack-compromising attacks - return-to-libc attacks and return-oriented-programming attacks. As is widely known, the processor core pushes return addresses for `call` instructions into the RAS and the branch predictor pops contained addresses out of the stack for speculative execution. This stack contains a limited number of return addresses in a circular buffer. In the following explanation, we use Figure 1 as an example and assume that the processor core has already pushed "Return_to_B", followed by "Return_to_A", into the RAS just before the attack.

In return-to-libc attacks, a stack frame is overwritten with a crafted payload through a buffer overflow. This payload contains the starting address of a library procedure to be exploited and arguments arranged for the procedure as illustrated in (c) in Figure 1. During a stack overflow, the starting address usually replaces a return address previously pushed by the processor - "Random" pointed to by the stack register for a brute-force attack. As the corrupted return address in the stack frame would differ from the corresponding entry in the RAS - "Random" in the stack frame against "Return_to_A" in the RAS, an RAS miss should be triggered to discard operands updated during speculative execution based on the branch prediction from the RAS. Therefore, validating a memory word triggering a RAS miss can detect return-to-libc attacks.

Meanwhile, the recent prevalence of 64-bit architectures makes brute-force return-to-libc attacks less threatening because of the increased entropy compared to 32-bit architectures [5]. However, the stack region is still attractive to adversaries as demonstrated in return-oriented-programming attacks [6]. Our claim is that our validation scheme is effective against those attacks as well.

The same reasoning on the interaction between the RAS and return-to-libc attacks can be applied to return-oriented-programming attacks. As noted in [6], the easiest way to place a payload filled with *gadgets* into a victimized process' address space is through stack overflow. Each gadget consists of a pointer to instruction sequences concluding with `return` and data words referenced by those instruction sequences doing a basic operation like load, store, or addition.

For example using (d) of Figure 1, assume that "Addr_code_k" points to a code snippet of (`pop %edx; ret;`). When the control flow is redirected to this snippet, the `pop` loads the value "Imm_word_k" into the `edx` register. The `ret` instruction in the snippet reads "Addr_code_l" for its next instruction address. In the given example, at least two mismatches would incur RAS miss predictions during execution of the payload - "Return_to_A" in the RAS against "Addr_code_j" in the stack frame and "Return_to_B" against "Addr_code_k". Therefore, our validation approach with the RAS miss handler is effective against ROP attacks.

The RAS must issue not only validation queries but also Erase queries in order to prevent false alarms under two following cases. In these cases, the processor core is forced to fetch return addresses for the next instruction addresses not from the RAS but from the memory word pointed to by the stack register. Without Erase queries, our validation unit would trigger false alarms for memory words that once contained spurious data and then are updated with legitimate return addresses later. One case is under the underflowed RAS buffer. Because circular buffers adopted for the RAS implementation have a limited number of entries and all of the RAS entries are discarded at various events like context switchings or miss-predictions, the RAS may have underflows. After an RAS underflow, the processor must fetch return addresses from memory word pointed to by the stack register. This approach mandates our validation unit to invoke validation queries for such return address fetches. Therefore, all of the taint statuses of memory words containing legitimate return addresses must be properly updated with Erase queries at `call` instructions to prevent false alarms.

The other case in which the processor core fetches return addresses from the stack region for the next instruction address is when a `longjmp` is utilized. Invoking a `longjmp` violates the last-in/first-out order of stack frames by bypassing multiple stack frames. As the `longjmp` implementation updates the stack register for stack frame bypassing while the RAS entries remain unchanged, RAS-miss predictions will be triggered at the `return` instruction encountered first and our validation unit must be invoked. Because of the same reason as the underflowed RAS buffer, the taint statuses for legitimate return addresses must be properly updated with Erase queries.

Legitimately invoking `longjmp` will not trigger false alarm in our validation scheme, because the `longjmp` implementation does not manipulate stack frames in main memory but restores execution contexts of the processor - i.e. hardware register values including the stack register - from the `jmp_buf` buffer. As the stack pointer register will point another uncompromised word in the stack frames, our validation scheme will not trigger false alarm for legitimate `longjmp`s.

3.5 Caching Structure

We propose a caching structure for our taint status data. The three operations described in Section 3.3 and PFN replacements are processed on this cache structure, *not* on the matrices and the bitmap arrays in main memory. Figure 2 illustrates this structure - our validation unit refers to only its internal cache for taint status data. This structure allows our validation scheme to operate while not influencing the existing (memory) cache operation like CFI (Section 2.1).

The proposed caching structure consists of a row cache and a bitmap cache. The row cache stores the most recently accessed row and is filled with all of the PFNs in the row that a requested frame number belongs to. This row-loading enables our validation scheme to promptly determine whether or not the PFN of a requested virtual address is contained in the matrix (Section 3.2). If the row index of a requested PFN does not match that of PFNs in the row cache, modified PFNs in the row cache are written back to the PFN matrix in main memory. After write-back, all PFNs in the row that the requested PFN belongs to are loaded into the row cache. If a requested PFN is found in the row cache (either after a row cache hit or row cache loading), our validation unit proceeds to the bitmap cache. Otherwise, the unit returns a "miss" and aborts the query processing. The bitmap cache has the same number of slots as the number of elements in the row cache, and each slot is allocated for the linked bitmap of each PFN in the row cache. Unlike the row cache, each slot of the bitmap cache is filled with the original bitmap in main memory only when its corresponding PFN is requested. We use this request-based bitmap loading approach because which bitmap slot will be referenced is unknown at a row cache loading.

All of the modified elements in row and bitmap caches must be written-back to the original data in main memory at the following events; (a) row cache miss; (b) context switching; (c) interrupt/exception; and (d) cache control instruction (e.g. cache flush). Three events other than row cache misses must invoke row/bitmap cache write-backs because they affect cache lines statuses.

4 Experiment

We used two simulators in our experiment - the Bochs x86 simulator and SimpleScalar. The Bochs x86 simulator was for attack simulations with the synthesized payload injection attacks and for large-scale statistics from an I/O intensive application running on a multi-tasking OS kernel. The SimpleScalar simulator with its architectural parameters was to assess the performance impact from our validation scheme.

4.1 Experimental Environment

In our simulation, the following configurations were applied to get statistics from both simulators; (a) Cache line size - 32 / 64 Bytes; (b) Matrix size - 32 / 64 / 128 / 256 PFNs; (c) Number of columns in matrix - 4 / 8 columns; and (d) Replacement policy - LRU / FIFO / Random. The row assignment exemplified in Figure 3 was used for our experiments.

We modified the Bochs simulator to have a single-level cache structure (one 32KB I-cache and one 32KB D-cache) as well as a 32-entry RAS in order to emulate the integration approach suggested in Section 3.4. The caching structure proposed in Section 3.5 was also implemented as described. `read` system calls from user applications running on the Linux OS were trapped to query Paint operations with I/O buffer addresses and their sizes (Section 3.3). On this modified simulator, we compiled the Linux kernel source codes with the GCC toolchain, and gathered statistics. We chose this process because many source code files with various sizes are read during compilation/linking and because the parallel compilation of the toolchain utilizes multi-tasking features.

We modified SimpleScalar as we did the Bochs simulator. For performance impact assessment, latencies were added to those modifications - in miss handlers for L1 I/D-caches, and miss-prediction of RAS. As the baseline SimpleScalar does not assess the performance impact from system calls, only the overhead from Lookup and Erase operations was assessed in our simulation. Instead, we programmed our SimpleScalar to flush both the row cache and the bitmap cache at every system call to create a conservative simulation environment. For latencies of row/bitmap caches, we assumed that those caches were implemented with the same circuitry as L2 caches; therefore the same latencies were used. We ran the SPEC2K benchmark suite for performance impact assessment.

4.2 Effectiveness Evaluation

We verified the effectiveness of the proposed scheme with two types of synthesized attacks on the modified Bochs simulator. For this evaluation, we programmed two benchmark suites - one for code injection attacks and one for an ROP attack. Each suite consists of two software modules - one is the virtual device driver that delivers a payload to user-space applications through the `read` system call and the other is the user application with the buffer overflow vulnerability.

Table 1. Architectural parameters for SimpleScalar. (L1 cache and RAS configurations are same as configurations for Bochs simulation).

Architectural parameters	**Value**
L2 Unified cache	256 K Bytes, 8-way set assoc., LRU
L1/L2 latency	4 / 42 cycles
DDR3 memory latency (First-chunk / inter-chunk)	72 / 2 cycles (DDR3, 1333MT/s, 8-burst-deep prefetch)
Row cache / Bitmap cache hit latency	42 / 42 cycles
Number of instructions	10,000,000

The code injection attack benchmark was the modified version of the benchmark proposed in [15] with the virtual device driver. The ROP attack benchmark was by our own and executed code snippets (implanted in the heap region) only by overflowing an stack frame with a payload containing four gadgets from [6].

The modified Bochs simulator were able to detect all of the synthesized attacks. In case of the code injection attack benchmark, our simulator detected executions of injected payloads through the I-cache miss handler. In addition, invoking validation queries at RAS miss-predictions was effective against attacks diverting the control flow redirection by compromising return addresses. The modified simulator also successfully detected all of the de-references of the code snippet addresses contained in the four gadgets through RAS miss-predictions.

4.3 Matrix Structure Evaluation

The most important metric regarding the matrix format is the miss rate of the Lookup operations, because PFN matrices have to reliably manage PFNs used for I/O. Experimental results were collected only from our modified Bochs simulator. This is because the input files for SPEC2K benchmarks running on SimpleScalar were not large enough to populate PFN matrices. For reliable statistics from our modified Bochs simulator, more than 400 I/O-active address spaces were profiled for each matrix configuration in our experiments. The percentages for each operation were; 51.34% from Lookup, 48.65% from Erase, and less than 0.001% from Paint.

Experimental results are shown in Figure 4. We found that (a) smaller matrices were susceptible to Lookup misses regardless of their replacement policies; (b) the LRU replacement policy outperformed the two other policies by orders of magnitude; and (c) a PFN matrix for one address space had to contain at least 64 entries to ensure low miss rates with LRU. This figure also suggests that the random replacement policy could be an alternative to LRU if we can allocate enough memory space for taint information (such as 256-PFN matrix) in order to achieve low miss rates in the Lookup operation.

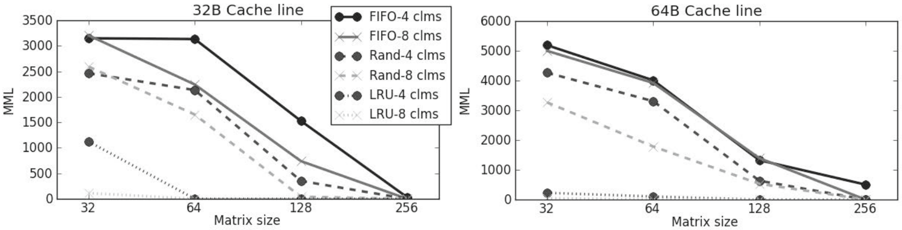


Fig. 4. Miss rates of Lookup queries - Misses per Million Lookups (smaller is better)

4.4 Row Cache/Bitmap Cache Evaluation

We evaluated the cache structure proposed in Section 3.5 with two statistics - one is the hit rates of the row/bitmap caches and the other is the distribution of the write-back size under a row cache miss. Only the results from the Bochs simulator are presented due to the same circumstance as the matrix evaluation - small input sizes for SPEC2K benchmarks for SimpleScalar.

The row cache hit rates were not high - between 0.610 and 0.709 (for 32-Byte cache line) or between 0.560 and 0.679 (for 64-Byte cache line), and 8-column matrices showed slightly higher hit rates than 4-column matrices. Meanwhile, the bitmap cache rates were high - from 0.961 to 0.998 and no significant differences were observed among replacement policies and matrix configurations.

The write-back sizes for row cache misses vary depending on how many row/bitmap cache elements have been modified. Regardless of replacement policies or the number of columns, all of the write-back size distributions were similar to other configurations; one bitmap write-back accounted for more than 99.5 percent of the total write backs. Other write backs like one new PFN with one new bitmap were less than 1 percent.

4.5 Performance Impact

We chose the IPC (Instructions Per Clock) statistics from the SimpleScalar for our performance metric, and Figure 5 shows normalized IPC values. The average IPC degradation was 12.3 percent, and the largest degradation was 48 percent. Experiments with the 32-Byte cache line size exhibited results similar to Figure 5 except for less degradation in several benchmarks.

Differences in performance degradation were mostly caused by queries and row cache misses. In case of *swim* with less than 1 percent degradation, only about 30,000 queries were issued while 10 million instructions were executed and the row cache hit rate was low - from 49.7 to 52 percent. On the other hand, *apsi* issued about 1.2 million queries, and the row cache hit rate was as high as 99.2 percent. This benchmark had no performance degradation. *Galgel* suffered the largest performance degradation; about 200,000 queries were issued, and the row cache hit rate ranged between 32.0 (64 pages) and 47.8 (128 pages) percent.

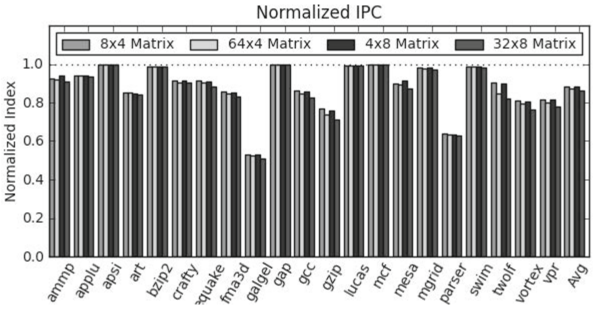


Fig. 5. IPC from SPEC2K benchmark - 64-Byte cache line, 32/256 PFN matrix, 4/8 columns

With the proposed caching structure, the read-in size for the row cache is 16 Bytes (4-column matrix) or 32 Bytes (8-column matrix) while one bitmap is 8 or 16 Bytes (Section 3.5). Although the row cache size is small compared to the cache line size (32/64 Bytes), row cache read-ins influenced the overall performance. There are several reasons for this performance degradation - (a) our row/bitmap cache structure does not have high-speed secondary storages; (b) the first-chunk latency of the DDR3 memory interface is large; and (c) row cache misses are triggered more frequently than system calls and context switchings.

In summary, the experimental results with the modified SimpleScalar and SPEC2K benchmarks show that our validation scheme imposes modest performance impact, mostly influenced by queries and row cache misses. Unfortunately, according to results presented in Section 4.4, our row cache hit rates were found to be low in kernel compilation experiment. This implies that the performance degradation in I/O-intensive applications like kernel compilation or internet browsing could be significant. We note that we tried different row index assignments from Figure 3 to put contiguous PFNs in one row so as to exploit spatial locality in the row cache. However, the experimental results exhibited much higher Lookup miss rates. Our understanding is that the virtual addresses randomized by ASLR incurred frequent thrashings in PFN matrices.

5 Discussion

5.1 System Software Support

As the OS kernel manages system resources related to multi-tasking features, the proposed validation scheme needs system software support.

The first is memory allocation for the PFN matrix and the bitmaps per address space. If a process is to be protected with the proposed validation scheme, the OS kernel has to allocate memory space for its taint information - one PFN matrix and one bitmap array. Because taint status data for one address space is concatenated to one of the paging tables as proposed in Section 3.2, the OS kernel can easily manage memory regions allocated for taint status information.

The OS kernel must initialize one PFN matrix and its associated bitmap array with zero values, because I/O activities must be logged and referenced only by our validation unit.

The second required support is to handle shared memory pages. Multi-tasking OS kernels extensively utilize shared memory pages for various purposes, like shared library procedures and the Inter-Process Communication feature. Because we assign one taint information set per address space, taint status changes occurring in memory pages of one address space are unknown to other address spaces sharing those pages. This problem could be addressed by making the OS kernel duplicate taint statuses from one address space to another at critical events like context switching. The OS kernel is responsible for this synchronization because the kernel manages the page sharing information.

5.2 False Positive

The proposed scheme has two false positives cases. This section briefly discusses how such false positives could occur and how to eliminate them.

The first false positive is from the matrix structure. Figure 4 shows the miss rates of Lookup operations with regard to matrix configurations. The miss rates shown in this figure essentially represent the false-positive rates of our validation scheme, because a hardware component which got "miss" as a response from the validation unit must continue instruction execution. This false positive can be prevented by allocating enough space for the taint status data.

The second case is caused by cache line size granularity. If a part of a string buffer allocated for spurious data happens to share the same memory block (with a cache line size) with a memory word containing a return address and an RAS miss prediction is observed due to an underflowed circular buffer, a false alarm will be triggered. During our experiments with the Bochs simulator, we observed false alarms caused by this problem. A remedy for this problem is to organize memory layouts in a way such that taint-candidate data and control flow data are placed far enough apart not to share a cache line. This work-around could increase memory consumption for an extended memory layout as a side effect.

5.3 Vulnerability and Limitations

Vulnerability of Taint Information. This validation scheme is vulnerable to rootkits compromising a target system with the same access level as its OS kernel. A plausible attack scenario is that a rootkit attack duplicates a sparsely-marked bitmap to other bitmap locations. If a rootkit identifies or crafts a sparsely-marked bitmap and populates a vulnerable process' bitmap array with this bitmap using the duplication procedure described in Section 5.1, our validation scheme would not be able to detect attacks accessing memory blocks whose taint status information has been compromised.

Limitations. The proposed validation scheme has two limitations. One limitation is that our protection scheme is unable to detect the execution of relocated foreign objects. By exploiting existing procedures or runtime environments like the JavaScript engine, malicious parties can relocate their payload(s) and redirect control flow to the payload - like heap-spraying attacks. As the proposed scheme does not track taint propagation across registers and the L1 data cache, memory blocks containing relocated foreign objects are highly unlikely to be marked as tainted. So, instruction fetch attempts to relocated payloads could be allowed without being blocked by our validation scheme.

Another limitation is that our validation scheme cannot detect attacks exploiting indirect branches other than `return`s - like `(call *)` or `(jmp *)`. During an indirect branch execution, the next instruction address contained in the data cache is loaded into the program counter either via a general purpose register (register indirect jump) or directly from the cache (memory indirect jump). Because these branches do not trigger RAS misses and our scheme does not validate data cache accesses, our validation unit is unable to detect exploitation of indirect branches for return-to-libc-style or ROP-without-returns attack [16].

6 Conclusion

In this paper, we have proposed a hardware mechanism to validate memory accesses influencing control flow redirection. The validation unit based on the proposed scheme manages the taint statuses of memory blocks for each address space at the cache line size granularity. This unit answers queries from other hardware components involved in control flow redirection. We also have proposed integration approaches and caching structures to alleviate performance overhead. Experiments with two simulators showed that proposed scheme is able to detect the synthesized payload injection attacks and to manage taint information with a limited amount of memory. Performance degradation varied from negligible to significant depending on the number of queries and row cache performance.

We have two major future works for this validation scheme. One is to augment the taint bit and to track its propagation at the register level while supporting our row/bitmap cache structure in order to address two limitations. The other work is to apply this scheme to 64-bit architecture. Because the bit width of a memory address is doubled (at most) compared to 32-bit architecture, we would need a sophisticated mechanism to alleviate the miss penalty of row cache misses.

Acknowledgment. This work was supported by Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education, Science and Technology (2012R1A1A2004615).

References

1. Etoh, H.: GCC extension for protecting applications from stack-smashing attacks, ProPolice (2003), http://www.trl.ibm.com/projects/security/ssp/
2. Frantzen, M., Shuey, M.: Stackghost: Hardware facilitated stack protection. In: Proceedings of the 10th USENIX Security Symposium, pp. 55–66 (2001)
3. Lee, G., Tyagi, A.: Encoded program counter: Self-protection from buffer overflow attacks. In: International Conference on Internet Computing, pp. 387–394 (2000)
4. Park, Y.-J., Zhang, Z., Lee, G.: Microarchitectural protection against stack-based buffer overflow attacks. IEEE Micro 26, 62–71 (2006)
5. Shacham, H., Page, M., Pfaff, B., Goh, E.-J., Modadugu, N., Boneh, D.: On the effectiveness of address-space randomization. In: Proceedings of the 11th ACM Conference on Computer and Communications Security, CCS 2004, pp. 298–307. ACM (2004)
6. Roemer, R., Buchanan, E., Shacham, H., Savage, S.: Return-oriented programming: Systems, languages, and applications. ACM Trans. Inf. Syst. Secur. 15(1), 2:1–2:34 (2012)
7. Abadi, M., Budiu, M., Erlingsson, U., Ligatti, J.: Control-flow integrity principles, implementations, and applications. ACM Trans. Inf. Syst. Secur. 13(1), 4:1–4:40 (2009)
8. Crandall, J.R., Chong, F.T.: Minos: Control data attack prevention orthogonal to memory model. In: Proceedings of the 37th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 37, pp. 221–232. IEEE Computer Society, Washington, DC (2004)
9. Suh, G.E., Lee, J.W., Zhang, D., Devadas, S.: Secure program execution via dynamic information flow tracking. SIGARCH Comput. Archit. News 32, 85–96 (2004)
10. Dalton, M., Kannan, H., Kozyrakis, C.: Raksha: a flexible information flow architecture for software security. SIGARCH Comput. Archit. News 35(2), 482–493 (2007)
11. Kannan, H., Dalton, M., Kozyrakis, C.: Decoupling dynamic information flow tracking with a dedicated coprocessor. In: DSN, pp. 105–114 (2009)
12. Newsome, J., Song, D.X.: Dynamic taint analysis for automatic detection, analysis, and signaturegeneration of exploits on commodity software. In: NDSS (2005)
13. Nethercote, N., Seward, J.: Valgrind: a framework for heavyweight dynamic binary instrumentation. SIGPLAN Not. 42, 89–100 (2007)
14. Qin, F., Wang, C., Li, Z., Kim, H.-S., Zhou, Y., Wu, Y.: Lift: A low-overhead practical information flow tracking system for detecting security attacks. In: Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 39, pp. 135–148. IEEE Computer Society, Washington, DC (2006)
15. Wilander, J., Kamkar, M.: A comparison of publicly available tools for dynamic buffer overflow prevention. In: Proc. of the 10th Network and Distributed System Security Symposium (February 2003)
16. Checkoway, S., Davi, L., Dmitrienko, A., Sadeghi, A.-R., Shacham, H., Winandy, M.: Return-oriented programming without returns. In: Proceedings of the 17th ACM Conference on Computer and Communications Security, CCS 2010, pp. 559–572. ACM, New York (2010)

DIONE: A Flexible Disk Monitoring and Analysis Framework

Jennifer Mankin and David Kaeli

Northeastern University, Boston, Massachusetts, USA
Department of Electrical and Computer Engineering
{jmankin,kaeli}@ece.neu.edu

Abstract. The proliferation of malware in recent years has motivated the need for tools to detect, analyze, and understand intrusions. Though analysis and detection can be difficult, malware fortunately leaves artifacts of its presence on disk. In this paper, we present DIONE, a flexible policy-based disk I/O monitoring and analysis infrastructure that can be used to analyze and understand malware behavior. DIONE interposes between a system-under-analysis and its hard disk, intercepting disk accesses and reconstructing a high-level semantic view of the disk and all operations on it. Since DIONE resides outside the host it is analyzing, it is resilient to attacks and misdirections by malware that attempts to mislead or hide from analyzers. By performing on-the-fly reconstruction of every operation, DIONE maintains a ground truth of the state of the file system which is always up-to-date—even as new files are created, deleted, moved, or altered.

DIONE is the first disk monitoring infrastructure to provide rich, up-to-date, low-level monitoring and analysis for NTFS: the notoriously complex, closed-source file system used by modern Microsoft Windows computing systems. By comparing a snapshot obtained by DIONE's *live-updating* capability to a static disk scan, we demonstrate that DIONE provides 100% accuracy in reconstructing file system operations. Despite this powerful instrumentation capability, DIONE has a minimal effect on the performance of the system. For most tests, DIONE results in a performance overhead of less than 10%—in many cases less than 3%—even when processing complex sequences of file system operations.

Keywords: Malware Analysis, Instrumentation, File System, Digital Forensics.

1 Introduction

As the arms race between malware creators and security researchers intensifies, it becomes increasingly important to develop tools to understand, detect, and prevent intrusions. The amount of malware has not only proliferated in recent years, but it has also become more sophisticated, employing methods to hide from or mislead malware detection mechanisms. As a result, it is critical to obtain information about malware that is as close to the truth as possible, which often

D. Balzarotti, S. Stolfo, and M. Cova (Eds.): RAID 2012, LNCS 7462, pp. 127–146, 2012.

means working at the lowest level possible. While researchers have had success using memory introspection [7,20,22], disk I/O instrumentation is also a critical tool for the analysis and detection of malware. Disk-level events provide a wealth of information about the state and history of a system. As a result, a flexible disk I/O analysis and instrumentation infrastructure provides key insight to better understand malware behavior.

In this paper, we present Dione: A **Disk I/O aNalysis Engine**. Dione is a flexible, policy-based disk monitoring infrastructure which facilitates the collection and analysis of disk I/O. It uses information from a sensor interposed between a System-Under-Analysis (SUA) and its hard disk. Since it monitors I/O outside the reach of the Operating System (OS), it cannot be misdirected or thwarted by rootkits—even those that have achieved superuser-level privilege. Dione reconstructs high-level file system operations using only intercepted metadata and disk sector addresses; while this reconstruction is performed with a high degree of accuracy, the performance impact of instrumentation is minimal. Dione only requires basic disk access information that can be obtained by many types of sensors, including both physical hardware sensors and virtualization-based sensors. It can, therefore, be used to analyze and detect malware that utilizes anti-sandbox or virtualization-evasion techniques, which have become increasingly common [3,8,19].

Rootkits, which can gain administrator privilege in order to control a system and hide themselves and other evidence of infection, often leave behind traces of disk activity, even when they eventually cover their tracks [14]. *Persistent rootkits* make changes to files on disk in order to survive reboots; this may include modifications to OS configuration files and system binaries. Even *non-persistent rootkits*, which reside purely in memory, may still present artifacts of infection through disk activity. This activity could include loading dynamic libraries, log-file scrubbing, and file time-stamp tampering [21].

In a simple world, a disk monitor could reside in the OS, where rich, high-level APIs expose semantics such as files and their properties, as well as the high-level operations which create, delete, and modify them. Unfortunately, this is not practical from a security perspective, as it is a well-understood problem that any malware that has escalated its privilege to the administrator level could then thwart or misdirect any data collection and analysis. For this reason, it is more desirable and secure to move the interposer outside the reach of the OS.

Unfortunately, housing a disk instrumentation engine outside the OS prohibits easy access to high-level constructs and operations. The **Semantic Gap** problem occurs when there is no mapping between low-level information (e.g., disk sectors and raw metadata) and high-level information (e.g., files and their properties). Fortunately, this challenge has been addressed in previous work with open-source libraries and drivers [6,26]. However, the **Temporal Gap** problem, in which low-level events across time must be reconstructed to identify high-level file system operations, has not been addressed in detail.

Unlike many low-level disk instrumentation approaches, Dione analyzes Windows systems running the NTFS file system. Furthermore, it performs *live* *updating*, resulting in a view of the file system that is always up-to-date (except for any delay as writes are flushed to disk). Dione works by pre-populating its data structures with a reconstructed view of the file system of the SUA. Then, as the SUA runs, Dione intercepts all disk accesses through the use of a sensor. For each sector accessed, Dione determines which file it belongs to and whether it has intercepted file contents or metadata. Next, Dione determines whether the file system state changed (e.g., due to a file being created, deleted, etc.). If so, it updates its high-level view of the file system state. Finally, Dione determines if any policies apply to that file, and if so, performs the appropriate action.

In this paper, we evaluate the accuracy, utility, and performance of Dione. We integrate Dione with a popular virtualization infrastructure in order to investigate the disk I/O of a virtual SUA. We also evaluate the performance of full disk instrumentation and the accuracy of Dione's live updating capability. Finally, we demonstrate the utility of Dione by instrumenting real-world malware samples and using the results to identify and analyze the malware.

2 Related Work

Much of the previous work in disk analysis focused on Intrusion Detection Systems (IDSs). Kim and Spafford's *Tripwire* monitored Unix systems for unauthorized modifications to the file system [15]. Tripwire performed file-level integrity checks and compared the result to a reference database. While it worked quite well to discover changes to files, it could only detect modifications between scans. Stolfo el al. also developed a host-based file system access anomaly detection system [27]. They utilized a file system sensor which wrapped around a modified file system to extract information about each file access. Both host-based solutions require a trusted OS. Conversely, Pennington et al. implemented a rule-based storage IDS that resided on an NFS server; their IDS monitored disk accesses for changes to specified attributes and file system operations [21].

While host-based IDSs are problematic because a privileged rootkit can override or misdirect malware detectors, IDSs based on Virtual Machine Introspection (VMI) offer both high visibility and isolation from compromised OSs. Payne et al. proposed requirements to guide any virtual machine monitoring infrastructure, and implemented XenAccess to incorporate VMI capabilities [20]. However, the disk-monitoring in their implementation can only be performed on paravirtualized OSes, such as Linux. Azmandian et al. used low-level architectural events and disk and network accesses in their machine learning-based VMI-IDS, though they did not utilize high-level disk semantics [1]. Zhang et al. presented a storage-monitoring infrastructure very similar to ours [29]. However, their monitoring framework was only implemented for FAT32 file systems, which is far less complex than NTFS and is rarely used in modern systems.

Jiang et al. also implemented a VMI-IDS, called *VMwatcher*, which incorporated disk, memory, and kernel-level events [12]. They too could not analyze the ubiquitous NTFS file system, and instead required that Windows VMs use the

Linux ext2/ext3 file system. The VMI-IDS of Joshi et al. detected intrusions before the vulnerability was disclosed [13]. However, their solution to inspecting disk accesses required invoking code in the address space of the guest itself, and subsequently performing a checkpoint and rollback.

Other researchers have acknowledged the role of disk accesses in malware intrusions by providing rootkit *prevention* solutions. With Rootkit Resistant Disks, Butler et al. provided a hardware-based solution to block accesses to sensitive directories, as long as these directories reside on a separate partition [4]. Chubachi et al. also provided a mechanism to block accesses to disk that could operate on a file-level granularity [9]. Unfortunately, they need to create a sector-based "watch-list" before the system boots and do not have a live updating capability to keep the list current as the system runs.

Previous work has also addressed the role of dynamic analysis and instrumentation for malware forensic analysis and classification, and yields information about disk activities. In-host solutions include *DiskMon* [24], part of the Sysinternals tools, and *CWSandbox* [28]; both provide disk access instrumentation capabilities for Windows systems. Similarly, Janus [11], DTrace [5], and Systrace [23] provide in-host instrumentation for Unix-based systems through system call interposition, also providing the ability to instrument disk accesses.

Given that in-host solutions can be misled or thwarted by advanced malware, more recent work has moved the analysis outside the host. Kruegel et al.'s *TTAnalyze* (later renamed *Anubis*) uses an emulation layer to profile malware, including file system activities, of a Windows guest [18]. Similarly, King et al.'s *BackTracker* uses a virtualized environment to gather process and file system-related events that led to a system compromise of a Linux guest [16]. Krishnan et al. created a whole-system analysis, combining memory, disk, and system call introspection [17]. However, their disk monitoring relies on periodic disk scans to connect blocks to files, and does not perform live updating.

3 Dione Overview

Dione is a flexible, policy-based disk I/O monitoring and analyzing infrastructure. Dione maintains a view of the file system under analysis. A disk sensor intercepts all accesses from the System-Under-Analysis (SUA) to its disk, and passes that low-level information to Dione. The toolkit then reconstructs the operation, updates its view of the file system (if necessary), and passes a high-level summary of the disk access to an analysis engine as specified by the user-defined policies. The rest of this section discusses Dione in more detail.

3.1 Threat Model and Assumptions

In our threat model, the SUA is untrusted and can be compromised, even by by malware with administrator-level privileges that can hide its presence from host-level detection mechanisms.

We assume that there is a sensor that interposes between the SUA and its hard disk and provides disk access information. This sensor can be a software sensor

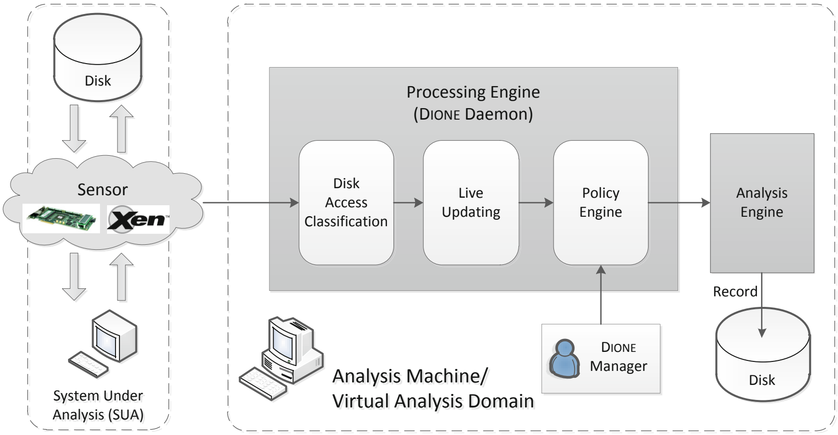

Disk
Sensor
Xen
System Under Analysis (SUA)
Processing Engine (DIONE Daemon)
Disk Access Classification
Live Updating
Policy Engine
Analysis Engine
Record
Disk
DIONE Manager
Analysis Machine/ Virtual Analysis Domain

Fig. 1. High-level overview of DIONE Architecture

(e.g., a virtualization layer) or a hardware sensor. We assume that both the sensor providing the disk access information and the Analysis Machine (that is, the machine which runs DIONE) are trusted. Therefore, in a virtualization-based solution, neither the hypervisor nor the virtual analysis domain is compromised.

3.2 DIONE Operation

There are four discrete components to DIONE: A sensor, a processing engine, an analysis engine, and the DIONE Manager. The DIONE architecture is shown in Figure 1.

The **Sensor** interposes between the SUA and its disk. It intercepts each disk access, and summarizes the access in terms of a sector address, a count of consecutive sectors, the operation (read/write), and the actual contents of the disk access (data being read, or data being written). The sensor type is flexible. It can be a physical sensor, which interposes between a physical SUA and the analysis machine, and extracts the disk access information from the protocol (e.g., SATA) command headers to send to DIONE. It can also be a virtual sensor, such as a hypervisor, which intercepts the disk I/O of a virtual SUA.

The **Processing Engine** is a daemon on the analysis machine. The multi-threaded DIONE daemon interacts with both the user and the sensor. It receives disk access information from the sensor, and performs three steps. The first step is *Disk Access Classification*; for each sector, it determines which file it belongs to (if known) and whether the access was to file content or metadata. In the *Live Updating* phase, it compares the intercepted metadata to its view of the file system to determine if any high-level changes occurred. It passes the high-level access summary to the *Policy Engine*, which determines if any policies apply to the file accessed. If so, it passes the information along to the analysis engine.

Table 1. Commands used for communication with the Dione daemon

Command	**Description**
DECLARE-RULE	Declare a new rule for instrumentation. Types of rules include: – **Record:** Record an access or file operation – **Timestamp Alert:** Alert if a timestamp is reversed. – **Hide Alert:** Alert if a file is hidden – **MBR Alert:** Alert if the master boot record is accessed.
DELETE-RULE	Delete a previously-declared rule
LIST	List all rules
APPLY	Bulk-apply declared rules to File Record data structures
SCAN	Perform a full scan of a disk image (or mounted disk partition), creating all File Records from the raw bytes and automatically applying all declared rules
SAVE	Save the state of the Dione File Record hierarchy to a file to be loaded from later
LOAD	Load the Dione File Record hierarchy from a previously-saved configuration file

The **Analysis Engine** performs some action on the information it has received from the processing engine. Currently, the analysis engine logs the accesses to a file, but future work will extend the analysis engine to perform malware classification or on-the-fly intrusion detection.

The Dione **Manager** is a command line program which the user invokes to send commands to the Dione daemon. The commands can be roughly divided into Rule Commands and State Commands and are summarized in Table 1.

Included in the Rule Command category are commands to declare, delete, list, or bulk-apply rules. The policies currently supported are summarized in Table 1. However, Dione is built to flexibly support the creation of new rule types. The State Command category contains rules to load and save a view of the state of the file system under analysis. The load step is necessary to pre-populate internal Dione data structures with a summary of the file system. This step is required before Dione will begin monitoring I/O. The goal of this stage is that Dione will already know everything about the file system before the SUA boots, so that it can immediately begin monitoring and analyzing disk I/O. This step can be accomplished with a disk scan, which reconstructs the file system from the raw bytes of the disk, or by loading a previously saved configuration file.

3.3 Live Updating

As the SUA boots and runs, new files are created, deleted, moved, expanded, shrunk, and renamed. As a result, the pre-populated view of the SUA's file system, including the mappings between sectors and files, quickly become out-of-date, reducing the accuracy of the monitoring and logging of disk I/O. The solution to this problem is *Live Updating*: an on-the-fly reconstruction of disk events based solely on the intercepted disk access information.

The next sections detail the challenges and solutions to live updating. As our implementation is initially geared toward Windows systems with the NTFS file system, and NTFS is a particularly challenging file system to perform live updating on, we will begin with an introduction to those NTFS concepts which are necessary for accurately describing the live updating implementation.

NTFS Concepts. Many of the challenges of interpreting NTFS arise from its scalability and reliability. Scalability is accomplished through a flexible disk layout and many levels of indirection. Reliability is accomplished through redundancy and by ordering writes in a systematic way to ensure a consistent result.

The primary metadata structure of NTFS is the *Master File Table*, or *MFT*. The MFT is composed of entries, which are each 1KB in size. Each file or directory has at least one MFT entry to describe it. The MFT entry is flexible: The first 42 bytes are the MFT entry header and have a defined purpose and format, but the rest of the bytes store only what is needed for the particular file it describes. In NTFS, everything is a file—even file system administrative metadata. This means that the MFT itself is a file: This file is called *$MFT*, and its contents are the entries of the MFT (therefore, the MFT has an entry in itself for itself). Figure 2 shows a representation of the MFT file, and expands *$MFT*'s entry (which always resides at index 0 in the MFT).

Everything associated with a file is stored in an *attribute*. The attribute types are pre-defined by NTFS to serve specific purposes. For example, the `$STANDARD_INFORMATION` attribute contains access times and permissions, and the `$FILE_NAME` attribute contains the file name and the parent directory's MFT index. Even the contents of a file are stored in an attribute, called the `$DATA` attribute. The contents of a directory are references to its children; these too are stored in attributes.

Each attribute consists of the standard attribute header, a type-specific header, and the contents of the attribute. If the contents of an attribute are small, then the contents will follow the headers and will reside in the MFT entry itself. If the contents are large, then an additional level of indirection is used. In this case, a *runlist* follows the attribute header. A runlist describes all the disk clusters[1] that actually store the contents of the attribute, where a *run* is described by a starting cluster address plus a count of consecutive clusters. In the example MFT of Figure 2, the contents of the `$STANDARD_INFORMATION` and `$FILE_NAME` attributes are resident. Since the content of the `$DATA` attribute is large, this attribute is not resident. Its runlist indicates that the *$MFT* data content can be found in clusters 104-107 and 220-221.

It is easy to see that a small file will occupy only the two sectors of its MFT entry. A large file will occupy the two sectors of its MFT entry, plus the content clusters themselves. Consider, then, the problem of a very large file on a highly fragmented disk: it might take more than the 1024 bytes just to store the content

[1] In NTFS terminology, a cluster is the minimum unit of disk access, and is generally eight sectors long in modern systems.

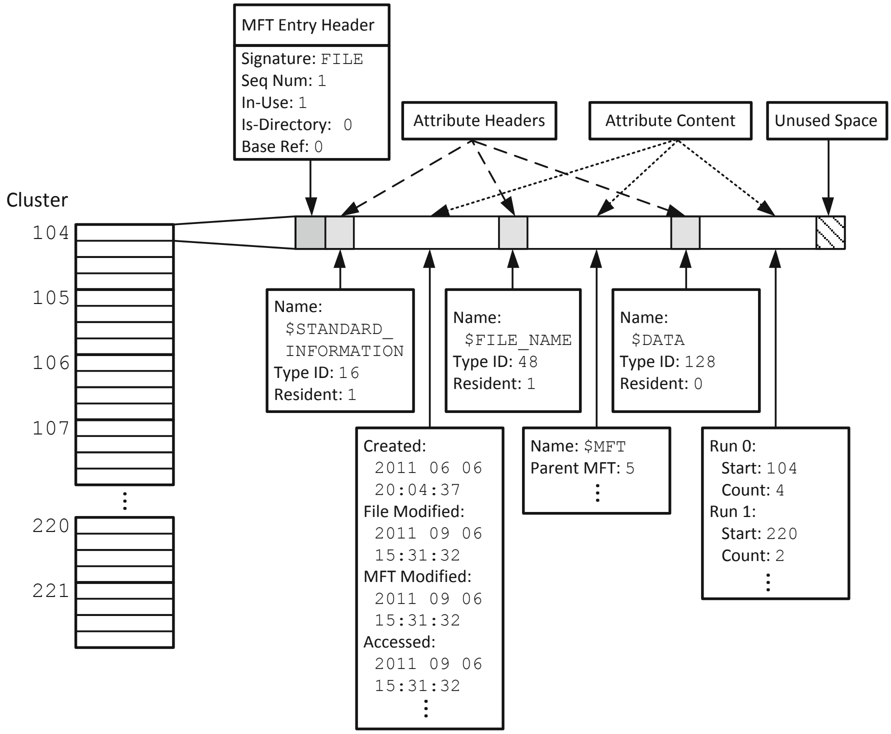


Fig. 2. Representation of the MFT, which is saved in a file called *$MFT*. The first entry holds the information to describe *$MFT* itself; the contents of this entry are expanded to show the structure and relevant information of a typical MFT entry.

runlist. In this case, NTFS scales with another level of indirection and another attribute, and multiple *non-base* MFT entries are allocated (in addition to the *base* entry) to store all attributes.

NTFS Live Updating Challenges. There are two big challenges to live updating: overcoming the Semantic Gap and the Temporal Gap. The Semantic Gap is a well-studied problem in which low-level data must be mapped to high-level data. In our case, we need to map the raw byte contents of a disk access to files and their properties. We utilize and build upon the open-source The Sleuth Kit (TSK) [6] to do much of the work to bridge the semantic gap.

The Temporal Gap occurs when low-level behaviors occurring at different points in time must be pieced together to reconstruct high-level operations. The high-level operations that Dione monitors include file creation, deletion, expansion, move/rename, and updates in MAC times and the hidden property.

The first challenge of live updating is identifying the fields in an intercepted MFT entry for which a change indicates a high-level operation. For some operations, a combination of changes across multiple intercepted MFT entries indicates that a certain high-level operation has occurred. Due to file system reliability constraints, these changes will be propagated to disk in an inconvenient ordering. Therefore, DIONE must piece together the low-level changes across time in order to reconstruct high-level events.

The biggest challenge resulting from the temporal gap is the detection of file creation. An intercepted MFT entry lacks two critical pieces of information: the MFT index of that entry, and the full path of the file it describes. For a static image, it is not a challenge to calculate both. However, in live analysis, the metadata creation will occur *before* the *$MFT* file's runlist is updated—and just like any other file, *$MFT* can expand to a non-contiguous location on disk. Therefore, it can be impossible to determine (at the time of interception) the MFT index of a newly created file. In fact, it can be impossible to determine at interception time whether a file creation *actually occurred* in the first place.

A similar challenge arises in determining the absolute path of a file. The MFT entry contains only the MFT index of that file's parent, not its entire path. If the parent's file creation has not yet been intercepted, or the intercepted parent did not have an MFT index when its creation was intercepted (due to the previously described problem), DIONE cannot identify the parent to reconstruct the path at the time of interception. This situation occurs quite frequently whenever an application is being installed. In this case, many (up to hundreds or thousands) of files are created in a very short amount of time. Since the OS bunches writes to disk in one delayed burst, many hierarchical directory levels are created in which DIONE cannot determine files' paths.

The temporal gap also proves a challenge when a file's attributes are divided over multiple MFT entries. As DIONE will only intercept one MFT entry at a time, it will never see the full picture at once. Therefore, it needs to account for the possibility of only intercepting a partial view of metadata, and to keep track of non-base entries in addition to base entries.

NTFS Live Updating Operation. Live updating in DIONE occurs in three steps. First, file metadata is intercepted as it is written to disk. Next, the pertinent properties of the file are parsed from the metadata, resulting in a reconstructed description of the file whose metadata was intercepted. Finally, DIONE uses the intercepted sector, the existing view of the file system, and the reconstructed file description from the second step to determine what event occurred. It updates the internal DIONE data structures to represent the file system change.

After intercepting an access to disk, DIONE looks at the intercepted disk contents and approximates whether the disk contents "look like" metadata (i.e., whether the contents appear to be an intercepted MFT entry). If it looks like metadata, DIONE parses the raw bytes and extracts the NTFS attributes. It also attempts to calculate the MFT index by determining where the intercepted sector falls within DIONE's copy of the MFT runlist. With this calculated index, it can attempt to retrieve a File Record. There are two outcomes of this lookup: either a valid File Record is retrieved, or no File Record matches the index.

Table 2. Summary of the artifacts for each file system operation. An MFT *index* is computed based on the intercepted *sector* and the known MFT runlist. If a file record is found with the calculated index, properties of the file record are compared with properties parsed from the intercepted metadata.
* A *replacement* is characterized by a file deletion and creation within the same flush to disk, whereby the same MFT entry is reused.

Operation	**Artifacts**
File Deletion	– *In-Use* flag **off** in intercepted MFT entry header
File Replacement*	– Creation Time: $Intercepted > FileRecord$, **OR** MFT Entry Sequence Number: $Intercepted > FileRecord$
File Rename	– File Name: $Intercepted \neq FileRecord$
File Move	– Parent's MFT Index: $Intercepted \neq FileRecord$
File Shrink/Expand	– Runlist: $Intercepted \neq FileRecord$, **OR** – Non-base entry created or deleted
Timestamp Reversal	– MAC Times: $Intercepted < FileRecord$
File Hidden	– *Hidden* flag: $Intercepted = 1$ **AND** $FileRecord = 0$

If a valid File Record is found, DIONE will compare the extracted attributes to those attributes found in the existing File Record. If any changes are detected, it will modify the File Record to reflect the changes. A summary of the semantic and temporal artifacts of each type of file operation is presented in Table 2.

If a valid File Record is not found, it means one of three things. In the first case, a new file has just been created, and it has been inserted into a "hole" in the MFT. The file creation can be verified because the intercepted sector falls within the known runlist of the MFT. In the second case, a new file has just been created, but the MFT was full, and thus it could not be inserted into a hole. The MFT index cannot be calculated, because the intercepted sector does not fall in *$MFT*'s runlist. DIONE buffers a reference to this file in a list called the *Wait Buffer*.[2] Eventually DIONE will intercept the *$MFT* file's expansion, the file creation will be validated, and the MFT index and path can be constructed. In the final case, the intercepted data had the format of metadata (e.g., the data looked like an MFT entry), but the data actually turned out to be the contents of another file. This happens for redundant copies of metadata and for the journal file *$Logfile*; additionally, a malicious user could create file contents which mimic the format of a MFT entry. In any of these cases, a reference to this suspected file—and the sector at which it was discovered—will be saved in

[2] A newly-created file will also be placed in the *Wait Buffer* if it has a valid MFT index, but its path cannot be constructed because its parent has yet to be intercepted.

the *Wait Buffer*. However, the *Wait Buffer* will be periodically purged of these File Records when their corresponding sectors are verified as belonging to a file which is not *$MFT*.

The root of trust of Dione is established and maintained by verifying the location of the MFT during the initial scan or load (the step described in Section 3.2). Dione maintains a list of all sectors that contain metadata via the runlist of the *$MFT* file. Since that runlist is only updated when *$MFT*'s metadata is intercepted (and the address of this metadata is known and unchanging), the list of sectors containing valid metadata is always verified. Therefore, when data that *looks like* metadata is encountered, it is only processed as metadata if it falls within this list. The only exception to this rule is for new file creation; as discussed above, this case is handled through the Wait Buffer. Therefore, a malicious user cannot forge metadata in order to evade or trick the system.

4 Experimental Results

Next, we evaluate the accuracy and performance of Dione and demonstrate its utility using real-world malware. Though Dione is a flexible instrumentation framework capable of collecting and analyzing data from both physical and virtual sensors, we use a Xen-based solution which utilizes the virtualization layer as a data-collecting sensor.

4.1 Experimental Setup

Our virtualization-based solution uses the Xen 4.0.1 hypervisor. Our host system contains a dual-core Intel Xeon 3060 processor with 4 GB RAM and Intel VMX hardware virtualization extensions to enable full-virtualization. The 160 GB, 7200 RPM SATA disk was partitioned with a 25 GB partition for the root directory and a 80 GB partition for the home directory. The virtual machine SUA runs Windows XP Service Pack 3 with the NTFS file system.

Xen uses a QEMU daemon to handle disk requests for a fully-virtualized (e.g., Windows) guest domain; this daemon resides in Domain 0. We implemented a sensor-side API (the *DiskMonitor*), which is linked into the Xen QEMU emulator code. The only modifications necessary to integrate Dione with Xen are to initialize the DiskMonitor and to call a function when performing a disk access. This function takes as parameters the starting sector address, the consecutive sector count, the operation type, and the actual disk contents that are read or will be written. The TrafficMonitor communicates this information to the Dione process via shared memory.

4.2 Accuracy Evaluation

In order to gauge the accuracy of live updating, we ran a series of tests to determine if Dione correctly reconstructed the file system operations for live updating. For our tests, we chose installation and uninstallation programs, as

Table 3. Breakdown of file system operations for each benchmark. The subset of file creations which wait for the delayed expansion of the MFT are also indicated. Note: The "All" test is not a sum of the individual tests, because the OS also creates, deletes, and moves files, and the number of these may differ slightly between tests.

Program	Creations	(Delayed)	Deletions	Moves	Errors
OpenOffice Install	3934	3930	1	0	0
Gimp Install	1380	1380	0	0	0
Firefox Install	152	135	71	0	0
OpenOffice Uninstall	353	62	3788	3836	0
Gimp Uninstall	5	0	1388	0	0
Firefox Uninstall	6	0	80	0	0
All	6500	6114	5986	3815	0

they perform many file system operations very quickly and stress the live updating system. We chose three open source applications (OpenOffice, Gimp, and Firefox), and performed both an installation and a uninstallation for each. We also ran an all-inclusive test that installed all three, then uninstalled all three.

These benchmarks perform a varying number of changes to the file system hierarchy. Table 3 lists each of the seven benchmarks and the number of file creations, deletions, and moves. As discussed in Section 3.3, if many new files are created at once and the MFT does not have enough free space to describe them, there is a delay between when the file creation is intercepted and when the file creation can be verified. We include the number of delayed-verification file creations in Table 3, as these stress DIONE's live updating accuracy.

For each test, we started from a clean Windows XP SP3 disk image. We executed one of the seven programs in a VM, instrumenting the file system. We shutdown the VM, and dumped DIONE's view of the dynamically-generated state of the file system to a file. We then ran a disk scan on the raw static disk image, and compared the results of the static raw disk scan to the results of the dynamic execution instrumentation. An *error* is defined as any difference between the dynamically-generated state and the static disk scan. This includes a missing file (missed creation), an extraneous file (missed deletion), a misnamed file, a file with the wrong parent ID or path, a file mislabeled as a file or directory, a file mislabeled as hidden, a file with any incorrect timestamp, or a file with an incorrect runlist. Table 3 shows the results of the accuracy tests. In each case, DIONE maintained a 100% accurate view of the file system, with no differences between the dynamically-generated view and the static disk scan.

4.3 Performance Evaluation

In order to gauge the performance degradation associated with DIONE's disk I/O instrumentation, we ran two classes of benchmarks: one dominated by file content reads and writes, and one dominated by file metadata reads and writes.

Iozone Benchmark. Iozone generates and measures a variety of file operations. It varies both the file size and the record size (e.g., the amount of data read/written in a given transaction). Because it creates very large files, reading and writing to the same file for each test, this is a content-heavy benchmark with very little metadata being processed.

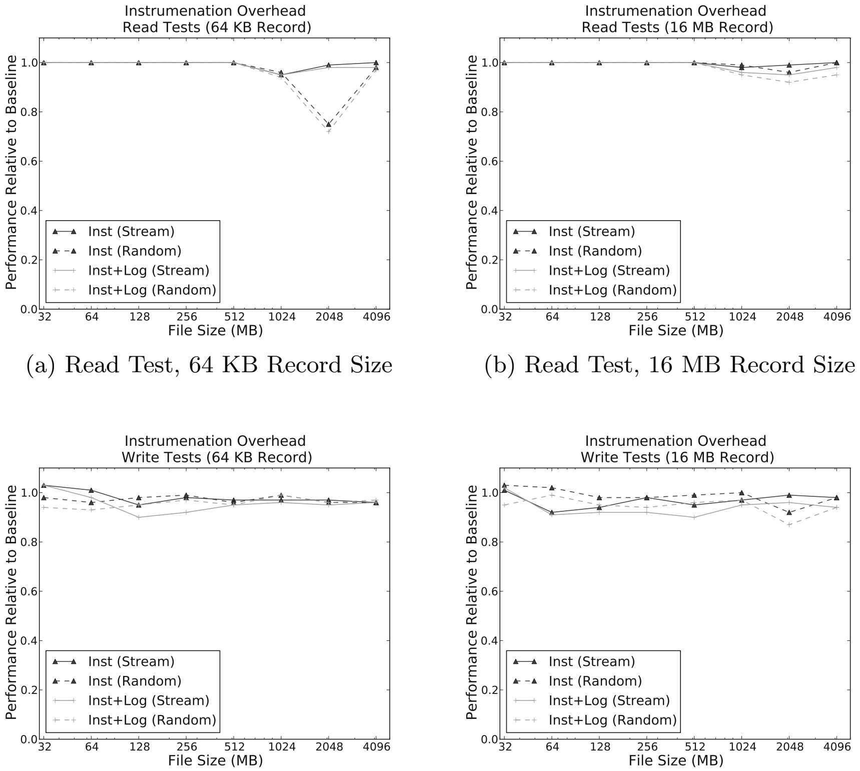
We ran all Iozone tests on a Windows XP virtual machine with a 16 GB virtual disk and 512 MB of virtual RAM. We used the *Write* and *Read* tests (which stream accesses through the file), and *Random Write* and *Random Read* (which perform random accesses). We varied the file size from 32 MB to 4 GB, and chose two record sizes: 64 KB and 16 MB. We ran each test 50 times to average out some of the variability that is inherent with running a user-space program in a virtual machine.

For each test, we ran three different instrumentation configurations. For the *Baseline* configuration, we ran all the tests without instrumentation (that is, with DIONE turned off). In the second configuration, called *Inst*, DIONE is on, and performing full instrumentation of the system. There are, however, no rules in the system, so it does not log any of these accesses. This configuration measures the minimum cost of instrumentation, including live updating. The final configuration is called *Inst+Log*. For these tests, DIONE is on and providing instrumentation; additionally, a rule is set to record every access to every file on the disk. Figure 3 shows the results of the tests. Each of the lines represents the performance with instrumentation, relative to the baseline configuration.

For the *Read* Iozone tests (Figures 3(a) and 3(b)), the slowdown attributed to instrumentation is near 0 for files 512 MB and smaller. Since the virtual machine has 512 MB of RAM, Windows prefetches and keeps data in the page cache for nearly the entire test. Practically, this means that the accesses rarely go to the virtual disk. Since DIONE only instruments actual I/O to the virtual disk—and not file I/O within the guest OS's page cache—DIONE is infrequently invoked.

At larger file sizes, Windows needs to fetch data from the virtual disk, which Xen intercepts and communicates to DIONE. At this point, the performance of instrumentation drops relative to the baseline case. In the worst case for streaming reads, DIONE's no-log instrumentation achieves 97% of the performance of the uninstrumented execution.

For the random read tests with large file sizes, there is a larger penalty paid during instrumentation. Recall that DIONE incurs a penalty relative to the amount of data accessed on the virtual disk. Therefore, the penalty is higher when more accesses are performed than are necessary. Windows XP utilizes *intelligent read-ahead*, in which the cache manager prefetches data from a file according to some perceived pattern [25]. For random reads, the prefetched data may be evicted from the cache before it is used, resulting in more accesses than necessary. This also explains why the penalty is not as high for the tests using the larger record size (for a given file size). Windows adjusts the amount of data to be prefetched based on the size of the access, so the ratio of prefetched data to file size increases with increasing record sizes. With more prefetched data, there is a higher likelihood that the data will be used before it is evicted from the cache.



(a) Read Test, 64 KB Record Size

(b) Read Test, 16 MB Record Size

(c) Write Test, 64 KB Record Size

(d) Write Test, 16 MB Record Size

Fig. 3. Performance of instrumentation, normalized to the baseline (no instrumentation) configuration for Iozone benchmarks for streaming and random read and write tests

Fortunately, this overhead is unlikely to be incurred in practice, as random-access of a 2 GB file is rarely performed.

Another observation is that the performance of Dione actually improves for streaming and random reads as file sizes grow larger than 1 and 2 GB, respectively. This is explained by considering the multiple levels of memory hierarchy in a virtualized system. As the file size grows larger than the VM's RAM, I/O must go to the virtual disk. However, the file may still be small enough to fit in the RAM of the host, as the host will naturally map files (in this case, the VM's disk image) to its own page cache. Thus, disk reads are not performed from the physical disk until the working size of the file becomes larger than available physical RAM. Since physical disk accesses are very slow, any cost associated with Dione's instrumentation is negligible compared to the cost of going to disk.

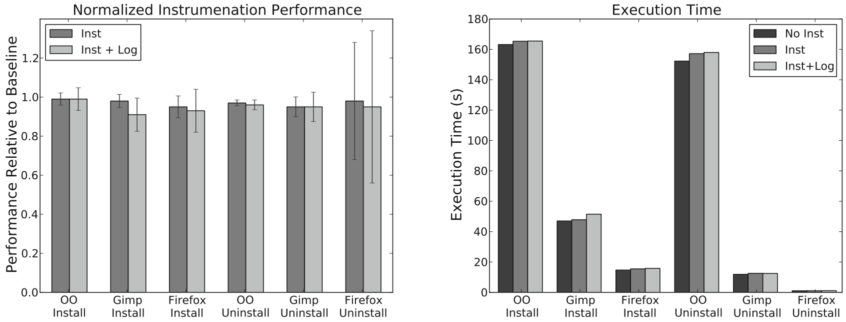
The Iozone *Write* tests (Figures 3(c) and 3(d)) show some performance degradation at small files sizes. Windows must periodically flush writes to the virtual disk, even if the working set fits in the page cache. However, the performance impact is minimal regardless of file size, with a worst-case performance degradation of 10% (though generally closer to 3%). Additionally, the random write tests do not have the same penalty associated with random reads. Since Windows only writes dirty blocks to disk, there are fewer unnecessary accesses to disk.

It is also noticeable that speedup values are sometimes greater than 1 for the 32 MB file size write tests. This would imply that a benchmark will run *faster* with instrumentation than without. In reality, this effect is explained by an optimization Windows uses when writing to disk. Instead of immediately flushing writes to disk, writes are buffered and flushed as a burst to disk. With this *Lazy-Writing*, one eighth of the dirty pages are flushed to disk every second, meaning that a flush could be delayed up to eight seconds [25]. From the perspective of the user—and therefore, the timer—the benchmark is reported to have completed. In reality, the writes are stored in the page cache and have yet to be flushed to disk. Most long-running benchmarks will have flushed the majority of their writes to disk before the process returns. However, a short-running benchmark—such as the Iozone benchmarks operating on a 32 MB file—may still have outstanding writes to flush. The time it will take to flush these will vary randomly through the tests. We reported a 21-24% standard deviation (normalized to the mean) for the baseline, instrumentation, and logging tests. This effect is examined in more detail in the next section.

For all tests, the cost of logging all accesses is relatively low, falling anywhere from 0-8%. For these tests, the root directory (under which the logs were stored) was on a separate partition than the disk image under instrumentation. Therefore, logging introduced an overhead, as the disk alternated between writing to the log file and accessing the VM's disk image. This performance penalty can be reduced by storing the log on the same partition as the disk image. Future work can also reduce the overhead by buffering log messages in memory—performing a burst write to the log—to reduce the physical movements of the disk.

Installation Benchmarks. In the second set of performance experiments, we evaluated the overhead of benchmarks that are high in metadata accesses. These tests will heavily stress the live updating part of DIONE's execution, which comprises the bulk of the computation performed in DIONE. We ran the same six install/uninstall benchmarks as the accuracy tests listed in Table 3. We ran each test ten times to average out variations inherent in running a user-space application on a virtual machine. For each run, we started from the same clean disk image snapshot. We used a Windows XP SP3 virtual machine with an 8 GB virtual disk and 512 MB of virtual RAM.

We compared the baseline execution (with no instrumentation) to full instrumentation with DIONE, with and without logging. Figure 4 shows the execution time for the three configurations, as well as the performance of DIONE's instrumentation relative to the baseline execution. As Figure 4 shows, even when the workload requires frequent metadata analysis for live updating, the overhead of instrumentation is low. Without logging, the full instrumentation of the benchmarks causes a 1-5% performance degradation.



(a) Performance of DIONE instrumentation (error bar equals one standard deviation).

(b) Average execution time with and without DIONE instrumentation.

Fig. 4. Evaluation of DIONE instrumentation for Open Office, Gimp, and Firefox Install/Uninstall benchmarks

The three benchmarks with the least penalty are OpenOffice installation and uninstallation and Gimp installation. These experience between 1-2% performance degradation for instrumentation without logging. Figure 4(b), which graphs the average execution time of the six benchmarks, provides more insight. These three benchmarks are the longest running of the six benchmarks, which is important because of how Windows performs writes to disk. As described in the previous section, writes could be delayed as long as eight seconds before they are flushed from the VM's page cache. While the program is reported to have completed, there are still outstanding writes that need to be flushed to disk. This effect is especially pronounced in any program with a runtime on the same order of magnitude as the write delay.

We can see this effect in Figure 4(a), which includes error bars showing the normalized standard deviation for the 10 runs of each benchmark. The 3 longest-running benchmarks also have the lowest standard deviations. This means that the results of these three tests are the most precise, and the average reflects the true cost of instrumentation. While two of the three shortest-running benchmarks have the highest reported cost of instrumentation, the standard deviation between tests is greater than the reported performance penalty. The execution time of the Firefox Uninstall is dwarfed by the time Windows may delay its writes—as reflected in its high standard deviation. In practice, this means that a user is unlikely to ever notice a slowdown attributed to disk instrumentation for short bursts of disk activity.

The *Inst+Log* tests show a 0-9% performance degradation compared to the baseline. In these tests, the disk image resided on the same partition as the log file. Therefore, the cost of logging to a file was lower than for the content tests.

4.4 Malware Case Studies

Next, we demonstrate the utility of Dione for forensic analysis by instrumenting two real-world malware samples and using the resulting logs to perform forensic analysis on the intrusion. In each case, we ran unlabeled malware on a clean, non-networked Windows XP virtual machine and instrumented the malware installation and the system upon reboot. We instrumented the entire file system by setting policies to record all accesses, timestamp reversals, and hide-file operations. We analyzed the logs and identified the samples by searching malware description databases based on the resulting file system operations and file names.

We found that Dione is quite useful for identifying and analyzing malware based on the intentional effects on the file system, such as the creation of files and directories. We also found that it is useful for understanding malware based on the unintentional effects on the file system, such as the loading of system libraries and the creation of system trace files. While the forensic analysis process was performed manually, future work looks to automate this process, using the disk access traces to perform automatic clustering or classification of unknown malware samples.

Backdoor.Bot The first real-world malware sample we discuss is the Backdoor.Bot malware [2], a Trojan first discovered in 2008. This malware opens a backdoor to the infected machine. It creates a directory and a process named *spoolsv*; since spoolsv is also the name of a legitimate Windows process, this malicious process is able to hide in plain sight from the average user. The malware is distributed with an image named *xmas.jpg*.

When the malware is first executed, Dione observes the creation of several files and directories. First, it creates the top-level directory, *WINDOWS\Temp \spoolsv*. This directory is created with the hidden flag already set, so that it cannot be viewed by the user. In the spoolsv directory, 12 more files are created with their hidden property already set, including the executable *spoolsv.exe*. Six other files are also created without their hidden property set, but that reside in the hidden spoolsv directory anyway. One of these files is the image file *xmas.jpg*; it is displayed to the user after the malware installs to deceive the user into believing that he simply opened an image file.

In addition to detecting infection through the intentional creation of the *spoolsv* directory and its contents, Dione can also deduce that some meaningful applications are run by the malware though unintentional file system artifacts. In order to speed up the time to load an application, Windows creates a trace file to enable fast future loading of the application. These trace files are stored in the *WINDOWS\Prefetch* directory. Therefore, the creation or access of one of these prefetch files indicates that the corresponding application has been run. Dione intercepts and records the creation of two prefetch files corresponding to *cmd.exe* and *regedit.exe*. This indicates that the malware has used `cmd` to launch `regedit` to modify the Windows Registry.

FakeAlert Defender. The FakeAlert System Defender trojan, identified by McAfee labs in 2011 [10], is "scareware" that modifies the file system in order to scare the user into purchasing an application to clean his system.

A few seconds after the malware has been executed, the user will see several error messages pop up alerting the user about different types of disk failures. As the user looks through his folders, all files will appear to have been deleted, though all directories remain. When the user reboots, the desktop is black, and it appears as if all files, directories, and even executables are lost. Instrumentation with Dione provides insight into how all of these actions are accomplished.

As the malware is executed, Dione observes that it first renames the original malicious file with a randomly-generated name with the extension *.exe.tmp*. It moves this file to the *Documents and Settings\%user_name%\Local Settings* directory, which is hidden by default. Next, it creates a randomly-named executable in the *Documents and Settings\All Users\Application Data* directory, which is also hidden by default. As it does with any newly-loaded application, Windows creates a prefetch file for the executable in *WINDOWS\Prefetch*.

Next, Dione observes that the malware performs the following steps with the goal of creating a copy of the file system hierarchy in a temporary folder. First, it creates a randomly-named directory in *Documents and Settings\%user name%\Local Settings\Temp*, and some numerically-named subfolders (e.g., *1*, *4*). Within these subfolders, the malware creates new directories, maintaining the hierarchy of the original filesystem. It then iterates through the user's existing file system hierarchy, and moves all files (not directories) into the corresponding directory under the *Temp* folder. The result is a hidden replication of the original hierarchy. While the original directory hierarchy also remains, all folders are empty, so it appears to the user that all his files have disappeared.

Once the user reboots, Dione observes the malware reversing the timestamp on the original malware executable. Finally, the malware iterates through every file and directory in the file system and changes its property to *hidden*, completing the deception that every file and directory on the disk has been deleted.

5 Conclusions

In this paper, we introduced Dione: a flexible, disk I/O instrumentation infrastructure for analyzing the ubiquitous Windows NTFS file system. Disk I/O is intercepted by a sensor, which passes disk access information to Dione for analysis. By residing outside the host, Dione is protected from the malware it is instrumenting. However, Dione has to bridge both the semantic and temporal gaps—not just reconstructing high-level semantics from low-level metadata, but also reconstructing high-level file operations from low-level events. We discussed the challenges of reconstructing disk operations, a process we call *Live Updating*, which ensures that Dione always has an up-to-date view of the file system.

We demonstrated that Dione achieves 100% accuracy in tracking disk operations and reconstructing high-level operations. We showed that despite this powerful instrumentation capability, Dione does not suffer from large performance degradation. We evaluated Dione's performance with workloads that generate a large volume of content accesses, as well as workloads that generate a high rate of metadata accesses and stress the live updating system. Dione preserves over 90% of the performance of native execution for most tests. We demonstrated the utility of Dione for forensic analysis by instrumenting two real-world malware intrusions. We showed that Dione can detect suspicious file operations that are hidden from the user, including file creations, timestamp reversals, file hiding, and the launching of applications to alter OS state.

Acknowledgments. The authors would like to thank Charles V. Wright and Joshua Hodosh for their invaluable input, guidance, and motivation for the development of Dione. Additionally, we would like to thank Dana Schaa for his continued support and feedback during the development of this work. Finally, we thank the anonymous reviewers for their comments and suggestions. This work is supported in part by a grant from MIT Lincoln Laboratory, and by Northeastern University's Institute for Information Assurance.

References

1. Azmandian, F., Moffie, M., Alshawabkeh, M., Dy, J., Aslam, J., Kaeli, D.: Virtual machine monitor-based lightweight intrusion detection. SIGOPS Operating Systems Review 45 (July 2011)
2. Virus profile: Generic backdoor!68a521cd1d46., http://home.glb.mcafee.com/virusinfo/.aspx?key=199638 (accessed on December 11, 2011)
3. Balzarotti, D., Cova, M., Karlberger, C., Kruegel, C., Kirda, E., Vigna, G.: Efficient detection of split personalities in malware. In: Network and Distributed System Security Symposium, NDSS (2010)
4. Butler, K.R., McLaughlin, S., McDaniel, P.D.: Rootkit-resistant disks. In: Computer and Communications Security (CCS), pp. 403–416. ACM (2008)
5. Cantrill, B.M., Shapiro, M.W., Leventhal, A.H.: Dynamic instrumentation of production systems. In: USENIX Annual Technical Conference, ATEC 2004. USENIX Association (2004)
6. Carrier, B. The Sleuth Kit (TSK), http://www.sleuthkit.org (accessed on October 1, 2011)
7. Case, A., Marziale, L., Richard III, G.G.: Dynamic recreation of kernel data structures for live forensics. Digital Investigation 7(suppl. 1) (2010); The Proceedings of the Tenth Annual DFRWS Conference
8. Chen, X., Andersen, J., Mao, Z., Bailey, M., Nazario, J.: Towards an understanding of anti-virtualization and anti-debugging behavior in modern malware. In: Dependable Systems and Networks (DSN), pp. 177–186 (2008)
9. Chubachi, Y., Shinagawa, T., Kato, K.: Hypervisor-based prevention of persistent rootkits. In: Symposium on Applied Computing (SAC). ACM (2010)
10. Mcafee labs thread advisory: Fakealert system defender. White-Paper, McAfee Inc. (June 2011)
11. Goldberg, I., Wagner, D., Thomas, R., Brewer, E.A.: A secure environment for untrusted helper applications: Confining the wily hacker. In: USENIX Security Symposium. USENIX Association (1996)

12. Jiang, X., Wang, X., Xu, D.: Stealthy malware detection through VMM-based "out-of-the-box" semantic view reconstruction. In: Computer and Communications Security (CCS), pp. 128–138. ACM (2007)
13. Joshi, A., King, S.T., Dunlap, G.W., Chen, P.M.: Detecting past and present intrusions through vulnerability-specific predicates. In: ACM Symposium on Operating Systems Principles (SOSP 2005), pp. 91–104 (2005)
14. Kapoor, A., Mathur, R.: Predicting the future of stealth attacks. In: Proceedings of the Virus Bulletin Conference (October 2011)
15. Kim, G.H., Spafford, E.H.: The design and implementation of tripwire: a file system integrity checker. In: Computer and Communications Security (CCS), pp. 18–29. ACM (1994)
16. King, S.T., Chen, P.M.: Backtracking intrusions. In: Symposium on Operating Systems Principles (SOSP). ACM (2003)
17. Krishnan, S., Snow, K.Z., Monrose, F.: Trail of bytes: efficient support for forensic analysis. In: Computer and Communications Security. ACM (2010)
18. Kruegel, C., Kirda, E., Bayer, U.: TTAnalyze: A tool for analyzing malware. In: European Institute for Computer Antivirus Research, EICAR (2006)
19. Lindorfer, M., Kolbitsch, C., Comparetti, P.M.: Detecting Environment-Sensitive Malware. In: Sommer, R., Balzarotti, D., Maier, G. (eds.) RAID 2011. LNCS, vol. 6961, pp. 338–357. Springer, Heidelberg (2011)
20. Payne, B.D., de, A., Carbone, M.D.P., Lee, W.: Secure and flexible monitoring of virtual machines. In: Annual Computer Security Applications Conference, ACSAC (2007)
21. Pennington, A.G., Strunk, J.D., Griffin, J.L., Soules, C.A.N., Goodson, G.R., Ganger, G.R.: Storage-based intrusion detection: Watching storage activity for suspicious behavior. In: USENIX Security Symposium (2003)
22. Petroni Jr., N.L., Fraser, T., Molina, J., Arbaugh, W.A.: Copilot - a coprocessor-based kernel runtime integrity monitor. In: USENIX Security Symposium. USENIX Assoc. (2004)
23. Provos, N.: Improving host security with system call policies. In: USENIX Security Symposium, Berkeley, CA, USA. USENIX Association (2003)
24. Russinovich, M.: DiskMon for Windows v2.01, `http://technet.microsoft.com/en-us/sysinternals/bb896646` (accessed on November 24, 2011)
25. Russinovich, M.E., Solomon, D.A.: Microsoft Windows Internals, 4th edn. Microsoft Press (2005)
26. Russon, R., Fledel, Y.: NTFS documentation. Tech. rep., Linux NTFS (2004)
27. Stolfo, S.J., Hershkop, S., Bui, L.H., Ferster, R., Wang, K.: Anomaly Detection in Computer Security and an Application to File System Accesses. In: Hacid, M.-S., Murray, N.V., Raś, Z.W., Tsumoto, S. (eds.) ISMIS 2005. LNCS (LNAI), vol. 3488, pp. 14–28. Springer, Heidelberg (2005)
28. Willems, C., Holz, T., Freiling, F.: Toward automated dynamic malware analysis using CWSandbox. IEEE Security Privacy 5(2) (March-April 2007)
29. Zhang, Y., Gu, Y., Wang, H., Wang, D.: Virtual-machine-based intrusion detection on file-aware block level storage. In: Symposium on Computer Architecture and High Performance Computing. IEEE Computer Society (2006)

AK-PPM: An Authenticated Packet Attribution Scheme for Mobile Ad Hoc Networks

Zhi Xu[2], Hungyuan Hsu[3], Xin Chen[2], Sencun Zhu[1,2], and Ali R. Hurson[4]

[1] College of Information Sciences and Technology, The Pennsylvania State University
[2] Department of Computer Science and Engineering, The Pennsylvania State University
{zux103,xvc5038,szhu}@cse.psu.edu
[3] Samsung Telecommunications America,USA
hyhsu@sta.samsung.com
[4] Department of Computer Science, Missouri University of Science and Technology
Hurson@mst.edu

Abstract. Packet traceback in mobile ad hoc networks (MANETs) is a technique for identifying the source and intermediaries of a packet forwarding path. While many IP traceback techniques have been introduced for packet attribution in the Internet, they are not directly applicable in MANETs due to unique challenges of MANET environments.

In this work, we make the first effort to quantitatively analyze the impacts of node mobility, attack packet rate, and path length on the traceability of two types of well-known IP traceback schemes: probabilistic packet marking (PPM) and hash-based logging. We then present the design of an authenticated K-sized Probabilistic Packet Marking (AK-PPM) scheme, which not only improves the effectiveness of source traceback in the MANET environment, but also provides authentication for forwarding paths. We prove that AK-PPM can achieve *asymptotically one-hop precise*, and present the performance measurement of AK-PPM in MANETs with both analytical models and simulations.

Keywords: Traceback, MANET, Probabilistic Packet Marking, Packet Source Identification, Path Reconstruction.

1 Introduction

Packet attribution includes identifying the source node of packets as well as the forwarding path from the source to the destination during the communication [1,2]. Both source and path information can help the defender to identify the attack source and locate its geographic location in many mobile ad hoc networks (MANETs) applications, such as defending Denial-of-Service (DoS) attacks [3] and false data injection attacks [4]. In business applications, packet attribution can be used in a positive way to provide the trustworthiness (or credibility) of data received by a destination node (e.g., data sink node). Data credibility is not just about who reports the data, but also the path the data comes from [5].

Many IP traceback protocols have been proposed for the Internet [6–8]. Among these, two types of IP traceback schemes dominate the literature: probabilistic packet

D. Balzarotti, S. Stolfo, and M. Cova (Eds.): RAID 2012, LNCS 7462, pp. 147–168, 2012.

marking (PPM) [8,9] and hash-based logging [7,10]. However, these IP traceback techniques are not directly applicable in MANETs due to several unique challenges in MANETs. First, packet forwarding paths in MANETs are easy to change due to node mobility [11], which causes the difficulty in reconstructing the attack path from a victim back to the attack source. Second, unlike the fixed routers in the Internet which are often assumed trusted, forwarding nodes in MANETs cannot be assumed as trusted, and compromised nodes may collude to confuse the traceback techniques. Moreover, because both the scale of a MANET and its data traffic rate are much smaller than that of the high-speed Internet, traceback in MANETs must be more efficient. So far, very little research has been done on traceback in MANETs [12–14].

In this paper, we make the first effort to quantitatively analyze the impact of node mobility on the performance of two representative traceback schemes (i.e. PPM and logging schemes). We formulate the impact of network parameters (e.g., the length of an attack path, the victim response time, and the mobility) on the *traceability* of these schemes in MANETs. Our analytical results show that (i) the traceability of both schemes decreases as node mobility increases; (ii) a PPM scheme is vulnerable to low-rate attacks, while a logging scheme performs poorly when a victim has a relatively high intrusion response time.

Further, we propose a new authenticated K-sized Probabilistic Packet Marking (AK-PPM) Scheme, which considers the efficiency and security requirements of traceback in the MANET environment. Our AK-PPM scheme stores multiple (up to K) marks within a single packet; thus, with the same number of packets received by a victim, more information about the forwarding path can be collected. Also, the AK-PPM scheme includes chained authentication mechanisms to protect the integrity of the mark sequence within a packet from being manipulated by colluding nodes in a forwarding path. We prove that is always *asymptotically one-hop precise*; that is, given enough attack packets, it can always trace to either an attack node, or the one-hop neighborhood of an attack node. We use analytical models and simulations to measure the performance of AK-PPM in MANETs of different settings.

2 Preliminaries

2.1 Network Model and Security Assumptions

In a MANET, nodes form a network on-the-fly and forward packets for one another. Nodes can establish trust through either a PKI, a Trusted Third Party (TTP), or pre-distributed shared keys. Further, any two nodes in the network, as long as knowing each other's id, can efficiently establish a pairwise key based on one of the existing schemes [15–17]. The key used in the message authentication code (MAC) generation at an intermediate node is its pairwise key share with the victim node. Therefore, malicious nodes cannot impersonate any benign node to the victim node. The link between two neighboring nodes is authenticated. During data forwarding, every packet is authenticated in a hop-by-hop fashion [18] with the pairwise key shared between neighboring nodes; thus, a malicious node cannot impersonate any good node and invalid packets are dropped right away. Such settings exist in many military MANET applications.

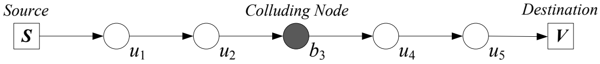


Fig. 1. An attack path $\mathcal{A}_M$ of packet M where node S and node b_3 are the compromised nodes. S injects bogus packets, and b_3 conspires to neutralize the traceback attempt.

Without loss of generality, consider a forwarding path $\mathcal{A}_M$ of packet M in Figure 1. For a node u_i located on a forwarding path between the source S and the destination V, a node u_j is called its *upstream* node if u_j is closer to S than u_i is. Similarly, u_j is a *downstream* node of u_i if u_j is closer to V. The distance of u_i from V on a path is the number of hops(i.e. nodes) between u_i and V on the path. For example, the distance of b_3 to V is 2 in the Figure 1.

In the case of identifying the attack source node in a DoS attack, it is reasonable to assume the victim (i.e., destination) node has installed an appropriate intrusion detection system (IDS) (e.g., Snort [19]) which can detect the malicious intrusion in the first place. This is a realistic assumption for intrusion detection based on attack packets received by destination node. A path may become broken for various reasons. In this study, we focus on the factor of node mobility. The *Link Duration* between two neighbor nodes on the path is defined the length of time interval during which two nodes are within each other's transmission range.

2.2 Attack Model

We assume the adversary may compromise one or multiple nodes and take full control of the compromised node(s). For example, in Figure 1, the source node S and the intermediate node b_3 are compromised and are both at the disposal of the adversary. b_3 may alter the packet's marks (if existing in the packet) or drop traceback queries. We present more details about specific attacks when introducing the proposed schemes in later Section 6.1.

Since a compromised node possesses a valid security credential, the injected packets will not be detected by its downstream nodes. However, because the links are authenticated, an attack source cannot impersonate any normal (benign) node. To hide itself, it will not put its address into the packet source field, and act as if it was a data forwarder for the packets while spoofing valid source ids. The attack source may change its location over time to hide itself.

2.3 Traceback Objectives

Ideally, a traceback procedure can identify the source node S and reconstruct the attack path $\mathcal{A}_M$. However, this goal is difficult to achieve due to two reasons.

Firstly, the source node S may never reveal its true identity so as to hide itself from traceback (a.k.a., the *first-hop* problem [13]). Thus, the best a traceback scheme can achieve is to identify the immediate downstream neighbor (e.g., u_1) of the attack source and reconstruct the path from the victim up to it. Once u_1 is identified, it relies on other online or offline analysis/detection measures (e.g., neighbor watching [20] or human intelligence) to identify the source node.

Secondly, compromised nodes on the attack path (e.g., b_3) may collude in order to confuse the traceback process. In an extreme case, b_3 may manipulate all attack packets going through it or even sacrifice itself to protect S. In this study, we assume that, from the attacker's perspective, the exposure of any one of its controlled nodes may have the same impact on the potential of future attacks. We call it a *success* when the immediate downstream neighbor of an attack source (e.g., S) or colluding node (e.g., b_3) on the attack path is identified.

3 Traceability Analysis of Existing Schemes for MANETs

Intuitively, an IP traceback scheme is not well suited for MANETs. However, no concrete analysis to quantify such intuition has been reported in the literature. A quantified analysis will clearly show how the current IP traceback protocols are susceptible in MANETs and it will serve as a metric for evaluating any new proposed traceback scheme for MANETs. As such, we will first make a traceability analysis of existing IP traceback schemes before presenting new schemes.

The common IP traceback schemes in the literature can be roughly categorized into *marking-based* schemes and *logging-based* schemes. Therefore, our discussion below will be focused on these two approaches. We define *traceability* $\mathcal{T}$ as the success rate of traceback in MANETs. $\mathcal{T}$ measures the probability that a traceback can be successfully performed before the attack forwarding path changes. We call it a success when the immediate downstream neighbor of an attack node (e.g., S or b_3) is identified.

3.1 Marking-Based Schemes

In a marking-based scheme, e.g., probabilistic packet marking (PPM) [9, 21], intermediate nodes (probabilistically) mark the packets being forwarded with partial path information, which later on allows a receiver to reconstruct the forwarding path given a modest number of the marked packets.

Scheme Description. Take the edge sampling based PPM algorithm [9] as an example. An IP traceback mark consists of a distance field and a *start-end* pair. Every intermediate node decides to either inscribe a packet (with a preset probability p), or not to mark (with probability $1 - p$). As nodes are allowed to overwrite the existing mark in the received packet, nodes closer to V will have more chance to leave their marks in packets. Relying on the relation between the distance to V and the distribution of received marks, a traceback can reconstruct the attack path with order from u_1 to V. Here we assume that each packet carries only one mark at a time,and the nodes between u_1 and V are trustworthy.

Traceability Analysis for MANETs. For an attack path $\mathcal{A}_M$ of length d, the victim can trace to the attack source only if it receives at least one packet marks from the immediate neighbor u_1 of the attack source before the path breaks up. Hence, the traceability of PPM for MANETs ,$\mathcal{T}_{ppm}$, is determined by two factors: *packet rate* γ and *path duration* PD, given a marking probability p. Let X_{u_1} denotes the number of packets that the victim has to receive before receiving the marking from u_1. The expected number of packets $E(X_{u_1})$ is:

$$E(X_{u_1}) = 1/(p(1-p)^{d-1}) \tag{1}$$

If the attacker sends the packets at a constant packet rate γ, then the expected time $T_{mark_{u_1}}$ by which the victim receives the marking from u_1 would be

$$T_{mark_{u_1}} = E(X_{u_1})/\gamma. \tag{2}$$

Sadagopan et al. [22] proposed a theoretical model to approximate the path duration based on the analysis of statistical extensive simulation results. According to [22], path duration can be approximated by an exponential distribution when the network nodes move in moderate to high velocities. The exponential random variable has the following cumulative distribution function (CDF)

$$F_{PD}(t,d) = \begin{cases} 1 - e^{\frac{-\lambda_0 dv}{R} t}, & t \geq 0 \\ 0, & t < 0 \end{cases} \tag{3}$$

where R denotes the radio transmission range, d denotes path length, v denotes the maximum velocity of a mobile node, and λ_0 is the proportionality constant.

Given the set of packets received by V, the victim V can launch a path reconstruction procedure [21] to reconstruct the order of marks(i.e., hops) on the path. Suppose that the reconstruction procedure is always performed correctly. The farthest hop u_1 will always be identified if V have received at least one mark from u_1. Thus, we define the traceability to identify u_1 as the probability that the path duration PD is greater than $T_{mark_{u_1}}$. The u_1-traceability for PPM is

$$\mathcal{T}_{ppm_{u_1}} = 1 - F_{PD}(T_{mark_{u_1}}, d) \tag{4}$$

According to Equation 2 and 3, we may derive the traceability $\mathcal{T}_{ppm,u_1}$ as

$$\mathcal{T}_{ppm,u_1}(d,\gamma) = \exp\{\frac{-\lambda_0 v}{R \cdot p(1-p)^{d-1}} \cdot \frac{d}{\gamma}\} \tag{5}$$

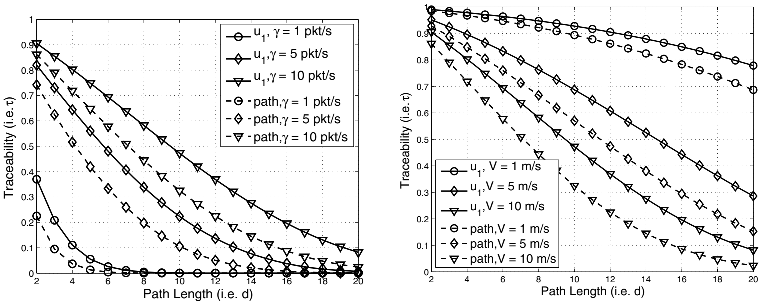
The problem of marking every intermediate node in the path can be formalized as a *Coupon Collector's Problem with sample size as one* [21, 23]. Briefly, the number of trials required to select one of each of d coupons (i.e. hops in our case) can be estimated as $d(H(d) + O(1))$, where $H(d) = 1/1 + 1/2 + ... + 1/d$.

Note that, in works such as [21], $H(d)$ was replaced by $\ln(d)$. However, we know that $H(d) \to \ln(d)$ if and only if $d \to \infty$. As the number of hops d in MANETs is small, the replacement may cause errors. For example, when d is less than $e \approx 2.7148$, the value of $\frac{\ln(d)}{p(1-p)^{d-1}}$ will be less than $\frac{1}{p(1-p)^{d-1}}$. Thus, we use $H(d)$ in our study instead of $\ln(d)$. Because the mark of one node may be overwritten by downstream nodes in a PPM scheme, the upper bound of the number of packets required to collect marks of all hops can be estimated as

$$E(X_{path(d)}) < H(d)/(p(1-p)^{d-1}) \tag{6}$$

We derive the lower bound of path traceability $\mathcal{T}_{ppm,path(d)}$, i.e., the probability of reconstructing an entire path of length d within the path duration PD.

$$\mathcal{T}_{ppm,path(d)}(d, \gamma) > \exp\{\frac{-\lambda_0 v}{R \cdot p(1-p)^{d-1}} \cdot \frac{d \cdot H(d)}{\gamma}\} \tag{7}$$



(a) $\mathcal{T}_{ppm,u_1}$ vs. d and γ

(b) $\mathcal{T}_{ppm,u_1}$ vs. d and velocity

Fig. 2. Traceability $\mathcal{T}_{ppm,u_1}$

Traceability Evaluation. To demonstrates the limitation of PPM in MANETs, we show the impacts of packet rate and mobility on the traceability $\mathcal{T}_{ppm,u_1}$ and $\mathcal{T}_{ppm,path}$ in Figure 2. The parameters applied in experiments are: $\lambda_0 = 0.59$, R = 250m, v = 10 m/s (in Figure 2(a)), $p = \frac{1}{20}$, $\gamma = 10$ pkt/sec (in Figure 2(b)).

From Figure 2(a) and (b), one can conclude that the traceability of a PPM scheme decreases when d increases. Also, the efficiency of traceability drops when the packet rate decreases or the mobility of network increases. For example, an attacker may control the packet rate at 1 pkt/s and launch the attack at six or seven hops away from the victim to avoid traceback with PPM schemes. Since d and γ affect $\mathcal{T}_{ppm}$ reversely and v, R, and p are environment variables, we consider the traceability as a function of d and γ, i.e., $\mathcal{T}_{ppm,u_1} = f(d, \gamma)$. Unfortunately, both the path length d and the packet rate γ are controlled by the adversary. The victim can basically do nothing to improve the traceability in the PPM scheme. Therefore, the application of PPM traceback in MANETs, i.e. $\min_{d,\gamma} \mathcal{T}_{ppm,u_1}(d, \gamma)$, overwhelmingly favors the attacker.

3.2 Logging-Based Schemes

In a logging-based scheme, intermediate nodes record the message digest of a forwarded packet; thus, every packet leaves a trail on the path from its source to its destination. To reduce the storage overhead for keeping the message digests, a typical space-efficient data structure called *Bloom filter* can be used. A practical architecture, *Source Path Isolation Engine* (SPIE) (also known as the *hash-based traceback*) was proposed in [7], in which the SPIE-deployed routers audit the traffic and digest the invariant portion of each packet for later queries. When the victim identifies an attack, it launches a traceback query to a traceback agent that is authorized to poll each of the intermediate nodes. Each polled node identifies the false packet trail by looking it up in its own Bloom filter and reports the result to the agent. The traceback agent rebuilds the attack graph with the help of the information about the network topology.

Scheme Description. To conduct on-line traceback for MANETs,the victim quickly responds to an attack by initiating a traceback process. The traceback query is expected to rapidly propagate along the reverse attack path, hoping that it can reach the neighbor of the attack source before the attack path breaks up. After processing the packet M, V finds that M is an attack packet(e.g., containing malicious code), so it sends out a traceback request along the reverse path of $\mathcal{A}_M$.

Traceability Analysis for MANETs. Analogous to the PPM based scheme, the path duration PD also controls traceability here. Thus, we consider the traceback behavior from two points of view: *path duration* and *traceback latency*. In logging-based schemes, a traceback is successful when the traceback latency is smaller than path duration PD. The traceback latency consists of the end-to-end propagation delays plus the victim IDS response time t_{resp}. Unlike in the PPM based schemes, path reconstruction and u_1 identification here are performed altogether in one round of traceback query. Therefore, we define the *traceability* $\mathfrak{T}_{log}$ with regard to a detected attack packet x as $\mathfrak{T}_{log} = \Pr\{PD > t_{SV} + t_{VS} + t_{resp}\}$. As a traceback process will only be initiated after the victim node receives the attack packet x successfully, we can rewrite the *traceability* formula as

$$\mathfrak{T}_{log} = \Pr\{PD > t_{SV} + t_{VS} + t_{resp} | PD > t_{SV}\}. \tag{8}$$

As described previously, PD can be modeled as an exponential random variable that is *memoryless*. Thus, Equation 8 can be reduced to

$$\mathfrak{T}_{log} = \Pr\{PD > t_{VS} + t_{resp}\} \tag{9}$$

Assuming the end-to-end delay is uniformly distributed and is proportional to the path length d (i.e., $t_{SV} \propto d$ and $t_{VS} \propto d$), the traceability $\mathfrak{T}_{log}$ will then be a function of d and t_{resp}. Based on Equation 3, we have

$$\mathfrak{T}_{log}(d, t_{resp}) = 1 - F_{PD}(t_{VS} + t_{resp}) = \exp\{\frac{-\lambda_0 dv}{R}(k \cdot d + t_{resp})\} \tag{10}$$

where k is the average delay in an intermediate node. Clearly, $\mathfrak{T}_{log}$ decreases as the node mobility (i.e., v) increases. Similar to the way we treat v and $\mathfrak{T}_{ppm}$, because v and t_{resp} affects $\mathfrak{T}_{log}$ in the same trend and v is an environment variable, we study the impact of t_{resp} on $\mathfrak{T}_{log}$ instead.

Traceability Evaluation. Compared with a marking-based scheme, the logging-based online traceback scheme is more fair. The victim node controls the response time t_{resp} and the attacker only controls the length of the attack path d. According to Equation 10, we notice that $\mathfrak{T}_{log} \searrow 0$ as $d \nearrow \infty$. Thus,$\mathfrak{T}_{log}$ unfairly favors the adversary when the path length d is unbounded. But, the longer the attack path is, the more likely the attack packet will be lost due to the changes in topology.

3.3 Review of Factors

Basing on our analytic results on PPM and logging-based schemes, several factors should be considered when designing traceback schemes for MANETs.

Attack Packet Rate. In MANETs, the attack packet rate required to launch DoS or DDoS attacks may be much lower [24]. Firstly, the targeted mobile node is more resource restricted, requiring lower attack packet rate to exhaust its resources (e.g., bandwidth consumption attack). Secondly, to protect the attack source from being exposed or from exhausting resources, the attacker may reduce the attack packet rate. Therefore, traceback schemes in MANETs have to adapt to the cases with low attack packet rates. PPM schemes may find it difficult to receive markings from u_1 and all other nodes on the path. In contrast, logging-based schemes are more resistent to low-rate attacks.

Communication Overhead. Due to the limited bandwidth in MANETs, a traceback scheme should introduce little traffic bandwidth overhead so as to prevent the downgrade of network services. PPM schemes generate little communication overhead, especially when marks are stored in the packet header. Logging-based schemes, however, generate much more communication overhead by sending traceback queries and receiving responses from the network.

Resource Costs. One assumption made in most existing traceback schemes is that all involved nodes are willing to cooperate during the traceback, by marking or logging the packets. In MANETs, the willingness of nodes may be affected by the cost for cooperation in the traceback, such as the cost of bandwidth, computing power, storage, and battery power. Thus, traceback schemes should avoid excessive workload on mobile nodes in order to make the scheme feasible to deploy in MANETs. In this case, PPM schemes usually demand comparatively lower resource consumption on nodes than logging-based schemes do.

Speed of Traceback. Improving the speed of traceback will help reduce the damage of DoS attacks to the network. Moreover, in MANETs, the speed of traceback is also important to identify the attack source and locate its geographic location. For example, the location information can be used to physically isolate or remove the attack source after traceback . The longer time it takes by the traceback the less chance the attacker can be found. Unfortunately, both PPM schemes and logging-based schemes have drawbacks in terms of speed. PPM schemes have to wait passively until enough marked attack packets have been received. On the other hand, logging-based schemes actively send traceback queries to the network but have to wait until receiving enough responses. Moreover, when the response time of the IDS in the victim is large, the attack path could become broken when a traceback is launched.

In general, PPM schemes seem to be more adaptable than logging-based schemes because the resource consumption of nodes and network is a critical issue in MANETs. However, improvements are needed on existing PPM schemes.

4 Improving Traceback Efficiency with Multiple Marks

Most existing marking based schemes store marks in the packet header [21]. Due to the fixed size of IP header space, the amount of routing information that can be carried in an IP packet is limited. For example,the single bit based PPM schemes in [25] and [26] require a huge number of packets for a successful path reconstruction. Differently, in MANETs, the packet format is relatively flexible, making it feasible for MANET designers to allocate more space in the packet header for multiple marks. Therefore, we may store multiple marks within a single packet to improve the efficiency of traceback. In untrusted MANETs, enabling multiple marks in a packet faces two challenges.

Challenge I: How to Determine the Number of Allowed Marks in the Packet? The number of marks allowed per packet is a key factor to the traceback efficiency. The more marks allowed, the more information a packet can carry about the forwarding path. Thus, if the number of marks is too small (e.g., single mark in [9]), both source identification and path reconstruction will require a huge amount of packets. To another extreme, if the number of marks is unrestricted (e.g., nested marking approach in [27]), the amount of payload information in a packet will be affected.

Challenge II: How to Protect the Integrity of Marks and Their Ordered Sequence in a Packet under the Colluding Attacks? A malicious intermediate node on the forwarding path may attempt to alter the marks carried by packets, in order to hide the source node and itself. Security mechanisms are needed in the mark design to protect the integrity of mark sequence and allow the victim node to detect the manipulation if existing.

We will address these two challenges in the following sections one by one.

5 K-Sized Probabilistic Packet Marking Scheme

In this section, we present a base scheme of K-sized Probabilistic Packet Marking (K-PPM) scheme, which improves the efficiency of packet traceback by allowing multiple marks in a packet. The proposed K-PPM scheme can be applied in trusted MANETs where the intermediate nodes are trusted. In the next section, we present an extended scheme that provides protection in untrusted MANETs.

5.1 Scheme Design

K-PPM scheme consists of two phases. In the first phase, every node in the MANET inscribes packets it is forwarding with a predefined probability p. Whenever source attribution is needed (e.g., DoS attack detected), a path reconstruction algorithm will be executed at the destination node based on the marks carried in received packets.

Marking Scheme. In the proposed K-PPM scheme, every packet contains a K-sized queue (i.e.,Q), which is managed by the First-In First-Out (FIFO) replacement algorithm. Each mark in Q consists of two node IDs. One is the ID of the current node which is forwarding this packet (i.e.,rcv), and the other is the ID of the node from whom the current node receives this packet (i.e., sdr).

Initially, when a packet leaves the source S, the queue Q is empty. When the packet arrives at an intermediate node u_i on the path, the node places a mark in the packet with a preset probability p. If the node decides to mark, it will generate a new mark containing its node ID i as rcv, and append this mark to the end of Q. If the queue is full already when arriving, the first (i.e., oldest or leftmost) mark in the queue will be discarded so that a new mark can be appended if u_i decides to mark. On the other hand, if u_i decides not to mark the packet, it will pass the packet to its downstream neighbor with no modification.

Path Reconstruction. With the collected marks, we first build a directed graph $G = (V, E)$: V consists of the set of nodes whose IDs appeared in received marks (either as a sdr or a rcv) at the destination node; and E consists of edges created by two rules:

Rule One: Assign a directed edge $j \rightarrow i$ if there exists a received mark within which the sdr is node j and the rcv is node i;

Rule Two: Assign a directed edge $j \rightarrow i$ if node i is the sdr of a received mark and node j is the rcv of its left adjacent mark in Q. Because nodes appearing in the left adjacent mark must be upstream nodes in the forwarding path.

In G, let I denote the set of nodes whose in-degree is 0, and let O denote the set of sink nodes whose out-degree is 0. Suppose that all packets traveled through the same forwarding path. With a sufficient amount of received packets, the sizes of I and O will be narrowed down to 1. The only node left in I will be output as the *identified source node* and the longest path from the node in I to the node in O will be output as the *identified forwarding path*.

With insufficient number of received packets, the accuracy of source identification and path reconstruction may be affected. First, the longest path may not traverse all nodes in G. In this case, we utilize the distribution of received marks to infer the order of nodes. Because the marking process is probabilistic with permission of dequeue, nodes closer to the destination node will have more chance to keep their marks in the packets. Due to the space limit, please refer to [9] for the detailed procedure. Second, the size of I may not be one. In this case, I represents a hotspot where any one of them has equal chance to be the real source node. Thus, extra investigation, e.g., neighbor monitoring [20] and physical security [28], can be conducted on nodes in I.

5.2 Improvement for u_1 Identification

We analyze the traceability of the proposed K-PPM scheme in the same setting as in Figure 1. Let $E_k(X_{u_1})$ denote the expected number of packets that the victim has to receive before receiving the mark from u_1. In the K-PPM scheme, the mark from u_1 will remain in the packet as long as the packet is later marked by no more than $K-1$ downstream nodes. If more than $K-1$ nodes mark the packet after u_1, the mark from u_1 will be discarded from the queue according to the FIFO policy. Thus, the expected number of packets, $E_k(X_{u_1})$, can be computed as:

$$E_k(X_{u_1}) = \frac{1}{p\sum_{i=0}^{k-1}(\binom{i}{d-1}p^i(1-p)^{d-1-i})}. \tag{11}$$

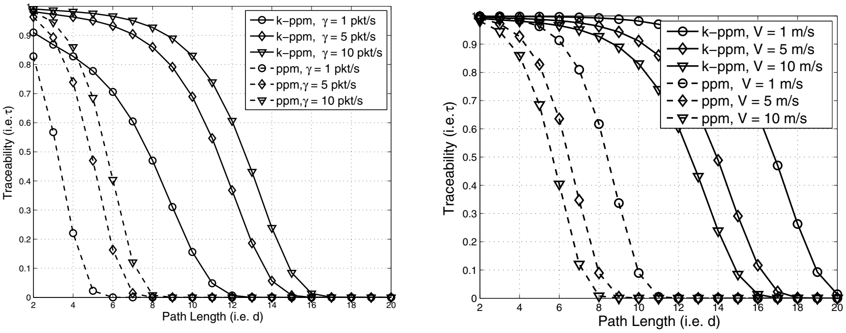
In Equ. 11, the value of $0 \leq i \leq k-1$ represents the number of marks within the packet, other than that from u_1. We claim that the K-PPM always requires less expected packets to receive a mark from u_1 (i.e., $E_k(X_{u_1}) \leq E(X_{u_1})$). The proof is as follows: When $k = 1$, we see that $E_k(X_{u_1}) = \frac{1}{p\sum_{i=0}^{k-1}(\binom{i}{d-1}p^i(1-p)^{d-1-i})} = \frac{1}{p(1-p)^{d-1-i}} = E(X_{u_1})$.

Thus, we show that, when $k = 1$, $E_k(X_{u_1})$ and $E_(X_{u_1})$ are equivalent. When $k > 1$, $\sum_{i=0}^{k-1}(\binom{i}{d-1}p^i(1-p)^{d-1-i})$ is always greater than $\sum_{i=0}^{0}(\binom{i}{d-1}p^i(1-p)^{d-1-i})$, thus greater than $p(1-p)^{d-1-i}$. ■

Accordingly, we derive the traceability $\mathcal{T}_{k-ppm,u_1}$ with respect to node u_1 as

$$\mathcal{T}_{k-ppm,u_1}(d,\gamma) = \exp\{\frac{-\lambda_0 v}{R \cdot p \sum_{i=0}^{k-1}(\binom{i}{d-1}p^i(1-p)^{d-1-i})} \cdot \frac{d}{\gamma}\} \tag{12}$$

In Figure 3(a) and 3(b), we compare the K-PPM scheme with PPM scheme in terms of u_1 traceability. The default setting in these two figures are $\lambda_0 = 0.59$, R = 250m ,$K = 4$, $p = 0.5$,v = 10 m/s(in Figure 3(a)), $\gamma = 10$ pkt/sec(in Figure 3(b)). Clearly, the K-PPM scheme can greatly improve the traceability in all mobility and attack packet rate settings.



(a) $\mathcal{T}_{k-ppm,u_1}$ vs. d and γ

(b) $\mathcal{T}_{k-ppm,u_1}$ vs. d and V

Fig. 3. Traceability $\mathcal{T}_{k-ppm,u_1}$

5.3 Improvement for Path Reconstruction

We formulate the path reconstruction problem in K-PPM scheme as a *Coupon Collector's Problem with random sample size* [29, 30]. In our case, the sample size is a random integer value between 0 and the size of queue, i.e., K. Following the approximation equation given in [30], we present the approximation of expected number of packets required to reconstruct a path with d intermediate nodes as:

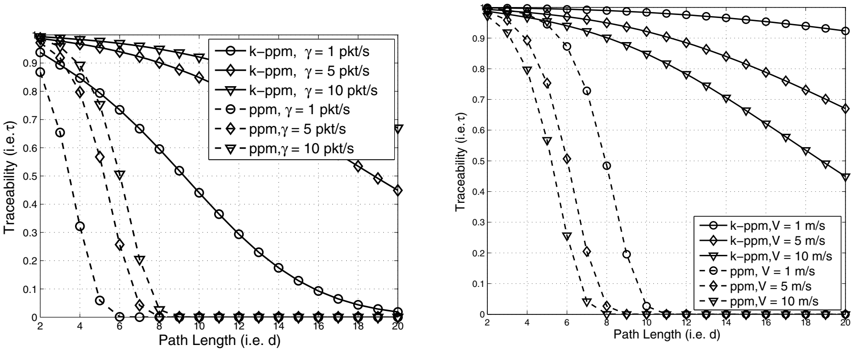
$$E_k(X_{Path_d}) \approx \frac{\sum_{i=0}^{d-1}\frac{1}{d-i}}{\sum_{i=0}^{d-1}\frac{1}{d-i}Pr\{L > i\}} + \frac{\sum_{r=1}^{d-1}\frac{1}{d-r}Pr\{L > r\}\sum_{j=1}^{r}1/(d-j+1)}{[\sum_{i=0}^{d-1}1/(d-i)Pr\{L > i\}]^2} \tag{13}$$

In the above Equation 13, L represents the number of marks carried by a packet and $Pr\{L > i\}$ represents the probability that L is greater than a value i. In our case, L is restricted by K and d, the length of path. This approximation is accurate when K is small with respect to d according to [29]. Thus, the value of $Pr\{L > i\}$ for $0 \leq i \leq d$ can be computed by:

- when $d \leq K$,
  - if $i > d$, $Pr\{L > i\} = 0$;
  - if $i \leq d$, $Pr\{L > i\} = 1 - \sum_{j=0}^{i} Pr\{L = j\}$;
  - $Pr\{L = i\} = \binom{i}{d} p^i (1-p)^{d-i}$ for $0 \leq i \leq d$;
- when $d > K$
  - if $i > K$, $Pr\{L > i\} = 0$;
  - if $i \leq K$, $Pr\{L > i\} = 1 - \sum_{j=0}^{i} Pr\{L = j\}$;
  - $Pr\{L = i\} = \binom{i}{d} p^i (1-p)^{d-i}$ for $0 \leq i < K$; $Pr\{L = K\} = 1 - \sum_{j=0}^{K-1} Pr\{L = j\}$;

In Figure 4(a) and 4(b), we present an example of comparison with $p = 0.5$, $k = 4$, $\lambda_0 = 0.59$, R = 250m , v = 10 m/s(in Figure 4(a)), $\gamma = 10$ pkt/sec(in Figure 4(b)). Through the comparison, we show that, the proposed K-PPM is more resistent to the increase of mobility and the decrease of packet rate, thus it is more suitable to be applied in MANETs than PPM scheme.

Moreover, by examining the Equations (12) and (13), the traceability of a K-PPM scheme decreases when d increases. Also, the efficiency of traceability drops when the packet rate decreases or the mobility of network increases. Luckily, the destination node can increase K to improve the traceability in the K-PPM scheme. Therefore, the K-PPM scheme is a more fair scheme compared with a PPM scheme, in which both the u_1 identification and path reconstruction can be described as: $\min_{d,\gamma} \max_K \mathcal{T}_{k-ppm}(d, \gamma, K)$.



(a) $\mathcal{T}_{k-ppm,path}$ vs. d and γ

(b) $\mathcal{T}_{k-ppm,path}$ vs. d and velocity

Fig. 4. Traceability $\mathcal{T}_{k-ppm,path}$

5.4 Parameter Selections

Of all parameters, K and p are defined by the network administrator. K is decided according to the tradeoff between traceback efficiency and overhead. The impact of p on traceability varies, depending on the path length. If p is too small, not enough marks can be collected at the victim. If it is too big, marks from nodes closer to the attack source are likely to be dequeued. In both cases, the traceability will be low. To maximize the usage of allocated space while avoiding too few markings, it is recommended that p be set according to equation $K = p * d_{est}$, where d_{est} denotes the estimated path length in a specific MANET. Detailed evaluations are presented in Section 7.

6 AK-PPM Scheme

In this section, we present an extended scheme, named AK-PPM (authenticated K-sized Probabilistic Packet Marking), to perform the traceback with the same efficiency as the proposed base scheme in untrusted MANET environments.

We first introduce the possible attacks toward the K-PPM scheme, then present the detailed design of AK-PPM scheme, and finally analyze its security.

6.1 Possible Attacks

The integrity of mark sequence in packets is crucial to the correctness of source attribution. Therefore, in AK-PPM, we focus on colluding attacks that could be launched by a malicious intermediate node on the forwarding path at the integrity of mark sequence in a packet. The purpose of these attacks is to conceal the real source node of a forwarding path. A colluding node succeeds if it can launch the attack without being noticed by the verifier at V.

Specifically, we consider the following possible attacks: (1) No-Mark Attacks: the colluding node may remove all marks in Q; (2) Mark Altering Attacks: the colluding node may modify the marks; (3) Mark Re-ordering Attacks: the colluding node may reorder the existing marks in Q; (4) Mark Insertion Attacks: the colluding node may insert at least one faked mark in Q; (5) Mark Deletion Attacks: the colluding node may arbitrary drop marks in Q; (6) Prefix Removal Attack: the colluding node may dequeue marks from the head of Q when Q is not full, dequeue one or many marks without adding its mark, or dequeue multiple marks while only adding its own mark. (7) Suffix Removal Attack: the colluding node may remove one or many marks from the tail of Q. (8) Jamming Attack: the colluding node(s) may jam Q by faked or legitimate marks (when the attack has control of multiple nodes in the network).

6.2 Scheme Objectives

The objective of AK-PPM scheme is to detect those attacks and identify either the real source node or the colluding node. Note that simply computing a single MAC over the entire packet (e.g., as in the nested marking scheme [27]) does not solve the attacks in our case, because of the dequeue operation in the K-PPM scheme. Briefly, if a mark is dequeued, all MACs including this dequeued mark will not be verifiable by the receiver.

Therefore, the AK-PPM scheme extends the base scheme by introducing a new chained authentication mechanism to protect the integrity of the mark sequence within a packet from being manipulated by colluding nodes in a forwarding path. Through security analysis in the end of this section, we prove that is always *asymptotically one-hop precise*; that is, given enough attack packets, it can always trace to either an attack (i.e., source or colluding) node, or the one-hop neighborhood of an attack node.

6.3 Revised Mark Design

In AK-PPM, an intermediate node will still mark the packet with the probability p. However, the mark format will be different. In Figure 5, we explain the details of AK-PPM with $K = 2$. Initially, S sends out a message M towards the destination V with a random variable $F = F_0$. Let H_k be a MAC function. If an intermediate node u_i, whose previous hop is u_j, decides to inscribe the packet, it will update the value of $F = F \oplus H_{k_i}(M|u_i)$, here k_i is the pairwise key shared between u_i and V. Note that a node can easily compute its pairwise key shared with another node given each other's id [15–17]. The MAC of $M|u_i$ serves as its "footprint" and we will see in the verification phase how it helps detect the marks in Q from being removed starting from the end. It also inserts its own mark $mark_i$ in the queue Q. In its mark, u_i includes its id and its previous hop u_j, the position I_i ($1 \leq I_i \leq K$, starting from the leftmost) of its mark in Q after adding its mark, a MAC $H_{k_i}(mark_j)$ over the previous mark $mark_j$ in Q (or a MAC of M if no previous mark exists), and a second MAC $H_{k_i}(M|F|u_j|u_i|I_i|H_{k_i}(mark_j))$. The new $mark_i$ will be appended at the end of Q. Note that here the first MAC in the mark provides an authenticated link to the previous mark; as a result, all the marks in a packet are protected in a chained structure. Removing any existing mark in the middle of Q will cause the chain to be broken and hence detected. If Q is already full, its first mark will be dequeued. If the node u_i decides not to inscribe the packet, it will simply forward the received packet to the downstream neighbor node without any change.

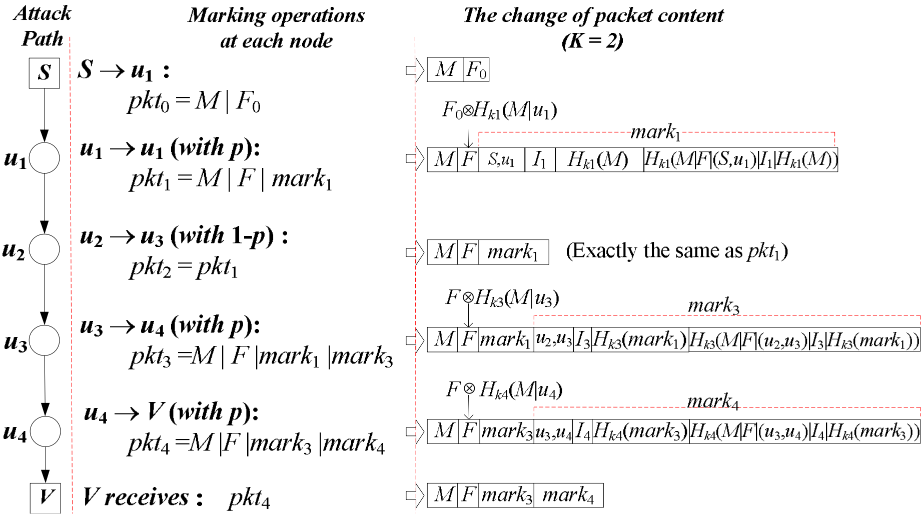

Attack Path
Marking operations at each node
The change of packet content (K = 2)
S
$S \rightarrow u_1$:
$pkt_0 = M | F_0$
M F_0
$F_0 \otimes H_{k1}(M|u_1)$
$mark_1$
u_1
$u_1 \rightarrow u_1$ (with p):
$pkt_1 = M | F | mark_1$
M F S,u_1 I_1 $H_{k1}(M)$ $H_{k1}(M|F|(S,u_1)|I_1|H_{k1}(M))$
u_2
$u_2 \rightarrow u_3$ (with 1-p) :
$pkt_2 = pkt_1$
M F $mark_1$
(Exactly the same as pkt_1)
$F \otimes H_{k3}(M|u_3)$
$mark_3$
u_3
$u_3 \rightarrow u_4$ (with p):
$pkt_3 = M | F | mark_1 | mark_3$
M F $mark_1$ u_2,u_3 I_3 $H_{k3}(mark_1)$ $H_{k3}(M|F|(u_2,u_3)|I_3|H_{k3}(mark_1))$
$F \otimes H_{k4}(M|u_4)$
$mark_4$
u_4
$u_4 \rightarrow V$ (with p):
$pkt_4 = M | F | mark_3 | mark_4$
M F $mark_3$ u_3,u_4 I_4 $H_{k4}(mark_3)$ $H_{k4}(M|F|(u_3,u_4)|I_4|H_{k4}(mark_3))$
V
V receives : pkt_4
M F $mark_3$ $mark_4$

Fig. 5. An example of marking procedure

6.4 Mark Verification

As forwarding nodes in MANETs cannot be assumed as trusted, we have to verify the integrity of marks carried in received packets before using them in traceback. For each received packet, the destination V verifies the marks in Q in a backward order starting from the end of Q. Given a mark $mark_i$ whose marker is u_i and the previous hop is u_j (shown in the beginning of the mark), V first identifies its pairwise key k_i shared with u_i. With k_i, it computes the value $MAC_x = H_{k_i}(M|F|u_j|u_i|I_i|MAC_i^1)$ where MAC_i^1 is the first MAC in $mark_i$, and then compare MAC_x with the second MAC in $mark_i$. If the two MAC values are the same, we know the mark $mark_i$ has not been tampered. Thus, we can use its first MAC to verify whether the link between the current mark $mark_i$ and the left adjacent mark $mark_j$ is authenticated. Specifically, we compute the value of $H_{k_i}(mark_j)$ and compare it with the first MAC in $mark_i$. If the results match, it means $mark_j$ was the last mark in Q when u_i was adding its own mark $mark_i$ at the time of packet forwarding.

If the mark $mark_i$ itself is not tampered, we continue to check if there exist illegal enqueue/dequeue operations in Q using the position variable I_i. As mentioned in the marking phase, according to our policy a node is allowed to dequeue the first mark of Q if and only if Q is already full when it tries to insert its own mark at the end of Q. Thus, if Q in the received packet is not full, the position of each mark, contained in each mark, should be exactly the same as its current position in Q. If Q is full, the position value in the last mark must be K. As a result, a valid position sequence $\{I_i\}$ will be either K or fewer consecutive items from the list $\{1, 2, ..., K, K\}$ depending on whether Q is full or not. If an inconsistency is detected, there must be some violation of the enqueue/dequeue policy (e.g., dequeue marks when Q is not full) in the stored marks. For example, when $K = 3$, it could be $\{1, 2\}$ (when the queue is not full), $\{1, 2, 3\}$ (when the queue is full but no dequeue happened), $\{2, 3, 3\}$ or $\{3, 3, 3\}$ (when dequeue happened). On the other hand, $\{2, 3\}$ is not legitimate because it indicates a malicious node in the path removed the first mark without adding its own one.

At last, we verify the value of F, which is used to prevent a malicious intermediate node from removing marks from the end of Q without being detected. During the marking process, every node u_i inscribing the packet has updated F with its "footprint", i.e. $H_{k_i}(M|u_i)$. Since F is part of input to the second MAC in the last mark, if the mark is not tampered, F will be authenticated. We will then compute $F = F \oplus H_{k_i}(M|u_i)$, which should be the F used in the calculation of the second MAC in the left adjacent mark. Iteratively, we can derive the value of F used in the construction of each mark in Q in the reverse order. This is possible because $\oplus$ is symmetric. If an intermediate malicious node removed the last mark in Q, it will also need to update F correctly. Otherwise none of the marks can be verified. This will require the malicious node to know the secret key shared between the previous marker and the destination node to compute the "footprint" correctly.

6.5 Security Analysis

Mark Integrity Assurance. We prove that the compromise of integrity of Q will always be detected in the proposed *AK-PPM* scheme.

Claim 1: Dropping marks from the end of Q will be detected.

Proof: Every node inscribing the packet updates F with its "footprint". If an intermediate malicious node removes marks from the end of Q, the inconsistency on value F will be detected in mark verification.

Claim 2: Dropping marks from the beginning of Q will always be detected.

Proof: The position variable I_i in a mark shows the status of Q when this mark is enqueued. As mentioned in mark verification, inconsistency can be detected on I_i if illegal dequeue operations were done by an intermediate malicious node.

Claim 3: Any enqueue/dequeue operation that violates the enqueue/dequeue policy will be detected.

Proof: The same as the proof of the previous claim.

Claim 4: Removing or inserting marks in the middle of Q will be detected.

Proof: The first MAC in every mark within Q provides an authenticated link to its previous mark. Consider a mark $mark_i$ and its previous mark $mark_j$ in Q. If an intermediate malicious node deletes $mark_j$ or adds a new mark after $mark_j$, it will be detected by $mark_i$ in mark verification.

Traceback Capacity Analysis. We prove that *AK-PPM is always asymptotically one-hop precise no matter whether the attack path is trusted or untrusted.*

Definition 1 (Trusted Attack Path): an attack path is trusted if all intermediate nodes in this attack path between attack source S and destination V are legitimate nodes.

Definition 2 (Untrusted Attack Path): an attack path is untrusted if there exist at least one colluding node in the attack path.

Definition 3 (One-hop precise): A traceback scheme is *one-hop precise* if it can always trace to either an attack node (i.e., the attack source or a colluding node in the path), or the one-hop neighborhood of an attack node.

Definition 4 (Asymptotically One-hop precise): A traceback scheme is *asymptotically one-hop precise* if it can always achieve one-hop precision when enough attack packets are received at destination node V.

Theorem 1. *The AK-PPM scheme is asymptotically one-hop precise if the attack path is trusted.*

Proof: First of all, in AK-PPM, the destination node V is able to receive marks from every node in the trusted path when enough attack packets are received, because every intermediate node in the attack path will inscribe the packet with the probability p. Further, the enqueue/dequeue operation allows the nodes closer to V to inscribe the packet event when Q is full.

Secondly, we prove that AK-PPM is asymptotically consecutively traceable. Consider two consecutive legitimate forwarding nodes u_j and u_i. As we have just proved, V is able to receive marks from both u_j and u_i. When $K \geq 2$, V will be able to receive packets which carry marks from both u_j and u_i. As both nodes are neighbors in the path, their marks must be neighboring with the same order in Q. With the chained structure, an authenticated link from u_j to u_i can be verified. Therefore, if we can trace to u_i, we can trace to u_j as well.

In [27], the author has proven that a scheme can achieve one-hop precision if and only if it is consecutively traceable. Therefore, the proposed AK-PPM scheme is asymptotically one-hop precise.

Theorem 2. *AK-PPM is asymptotically one-hop precise if there exists only one colluding node in the attack path.*

Proof: Let b denote the colluding node. For packets passing through b, it will have two choices: either tamper the existing marks in Q in order to hide S from traceback, or not tamper the marks. If b decides to tamper the marks, according to Theorem 1, we can trace to b because the sub-path between b and V is a trusted path. If b decides not to tamper the marks, S will be traced. In either case, an attack node is identified and we achieve our goal.

Theorem 3. *AK-PPM is asymptotically one-hop precise if there exist more than one colluding node in the attack path.*

Proof: If all colluding nodes do not tamper the marks within packets, S will be traced. Otherwise, let b denote the colluding node closest to V, which tampers the marks within packets it forwarded. As we have shown in Theorem 2, we will be able to trace to b. Again, in either case, an attack node is identified and we achieve our goal.

Corollary 1. *AK-PPM is always asymptotically one-hop precise.*

Proof: With Theorem 1, 2, and 3, we come to the Corollary 1 that, AK-PPM is always asymptotically one-hop precise.

6.6 Parameter Selections

The proposed AK-PPM requires $K * Size(Mark) + Size(F)$ of extra space in every packet. Each mark consists of two node IDs, a position index up to K, and two MAC values, so its size is $2|ID| + \lfloor \log_2(K) \rfloor + 2|H()|$. The MAC value can be generated by a secure one-way hash function. The detailed settings are decided by the network administrator. Suppose that $K = 3$, $|ID| = 8$ bits, $|H()| = 16$ bits, $|F| = 16$ bits. The per-packet overhead of AK-PPM is $(16 + 3 * (16 + 2 + 32)) = 166$ bits = 21 Bytes. Suppose that we do not change the IP header and store all marks within the packet payload. For a packet of size 512 bytes, the overhead rate is 4%. To further lower the marking overhead, we may make a tradeoff between security and performance. For example, we may reduce the size of the MACs contained in the marks to one byte. In the previous example, it will give us the per packet overhead of 15 bytes. Because the number of attack packets is not big in an MANET, a one-byte MAC could be sufficient to filter out forged packets.

6.7 Anonymous ID

In the AK-PPM scheme, an intermediate node will include its node ID in generated marks. If the included node ID is in plaintext, the colluding node on the path may selectively drop only packets which contain marks generated by nodes close to S

(i.e., *selective dropping attack*). So that the later traceback will end at an upstream node of the colluding node that is far from S. To avoid other nodes knowing who have inscribed the packet, an intermediate node may include an anonymous ID instead of its real ID in plaintext. Depending on the initiator of traceback, the design of anonymous ID can be different. If the traceback will be launched by the network administrator, the intermediate node may use a pseudo ID that is verifiable by the network administrator only. If the traceback will be launched by V and the intermediate node shares a paired key with V, we may use the paired key to encrypt the node ID and use the encrypted ID in the mark. Further, we may add randomness in the anonymous ID (e.g., by applying *Counter-Mode Encryption*) to prevent attackers from mapping the anonymous IDs with their real ID/nodes by observing the packet traffic.

7 Simulation and Evaluation

We use simulations to show (1) how to select proper values for parameters K and p, and (2) the improvement of traceability by the proposed AK-PPM scheme.

7.1 Simulation Settings

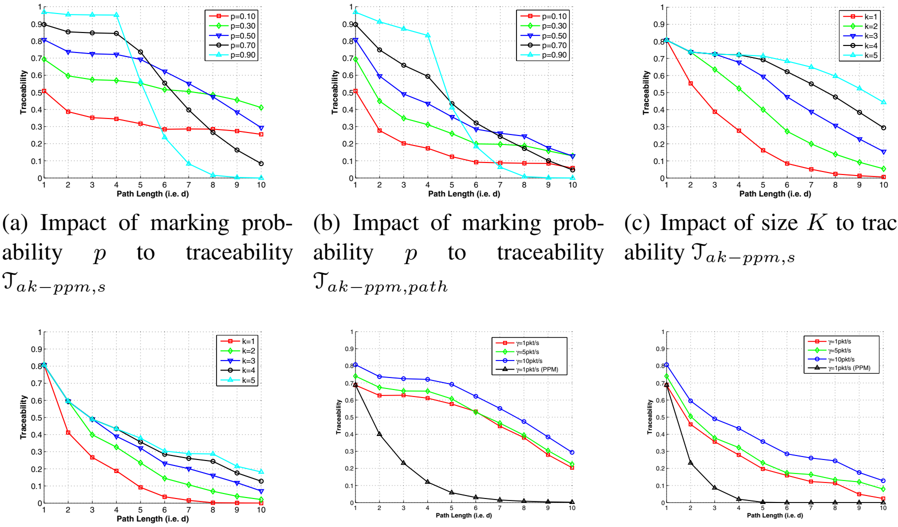
Our simulations are based on the GlomoSim 2.03 simulator. Major simulation parameters are listed below: physical link bandwidth (2 Mbps), transmission range ($R = 250m$), number of nodes (100), territory ($3000m \times 3000m$), MAC layer protocol (IEEE 802.11). The AK-PPM scheme is implemented in the network layer. In the network layer, we apply the DSR routing protocol.

In the simulation, we randomly chose two nodes as the attack source and the destination node, respectively. The attack node sends packets to the destination node in a constant rate (i.e., γ). γ is chosen from 1, 5, and 10 pkt/sec to simulate different attack intensities. An intermediate node follows the marking policy proposed in the AK-PPM scheme with probability p chosen in the range of 0.05 and 1.0. In addition, K is chosen from 1 to 5. We evaluate the performance of the AK-PPM protocol while changing these three parameters. By default, $\gamma = 10$pkts/s, $p = 0.5$, and $K = 4$. For each parameter setting, we ran the simulation for at least 20 times. Each simulation run lasts two hours in simulation time. Every node moves in the random waypoint mobility model at a speed uniformly distributed between 0 and $10m/s$.

7.2 Simulation Results

Figure 6 shows the simulation results for traceability of both source and path.

Parameter p: As shown in Figure 6(a) and (b), increasing p does not necessarily improve the traceability. The curves can be divided into two parts. When $d \leq K = 4$, because no mark overwriting happens, a larger p will certainly improve the traceability. However, when $d > K$, the traceability will also be affected by possible mark dequeuing operations. In this case, when p is large, the probability that marks of u_1 and other



(a) Impact of marking probability p to traceability $\mathcal{T}_{ak-ppm,s}$

(b) Impact of marking probability p to traceability $\mathcal{T}_{ak-ppm,path}$

(c) Impact of size K to traceability $\mathcal{T}_{ak-ppm,s}$

(d) Impact of size K to traceability $\mathcal{T}_{ak-ppm,path}$

(e) Impact of packet rate γ to traceability $\mathcal{T}_{ak-ppm,s}$

(f) Impact of packet rate γ to traceability $\mathcal{T}_{ak-ppm,path}$

Fig. 6. Simulation Results

nodes far from V are overwritten will be high. For example, we see a swift downgrade of traceability when $p = 0.9$ in both figures.

Parameter K: the size of queue K represents the space in a packet reserved for source attribution. In Figure 6(c) and (d), p is set as 0.5 and we show the impact of K to the traceability in the proposed AK-PPM scheme. When $K = 1$, the AK-PPM scheme is the same as the PPM scheme. From these two figures, we can see that increasing K will obviously improve traceability, especially for source identification. However, the improvement is not significant when the d is small. In summary, to maximize the usage of allocated space while avoiding insufficient marking, it is recommended that p and K be set according to equation $K = p * d_{est}$, where d_{est} denotes the estimated path length in a specific MANET.

Low Packet Rate γ: the packet rate γ will be small in cases, such as low rate DoS attacks [24], In Figure 6(e) and (f), we show the impact of the attack packet rate on traceability when $K = 4$ and $p = 0.5$. For comparison, we also show the simulation result of PPM scheme on the same traces. As we can see, compared to the PPM scheme, the downgrade of traceability for AK-PPM is less significant, meaning that the AK-PPM scheme is more resistent to low-rate attacks.

8 Related Work

About traceback schemes in MANETs, Thing and Lee [12] conducted simulation studies to investigate the feasibility of applying SPIE, PPM, and ITrace protocols in MANETs.

However, no quantitative study about node mobility is presented and the factor of mobility is considered by simulations. Zarai et al. [31] and Kim et al. [32] proposed cluster-based traceback schemes only for MANETs with trusted nodes. Kim and Helmy [14] proposed a traceback scheme in MANETs, named $SWAT$. Their scheme utilizes the small world model and sends the attack traffic signature to neighbor nodes of a victim which observe similar attack traffic. The major drawback of SWAT is the large amount of communication cost during the traceback. Huang and Lee [13] first introduced the concept of hotspot-based traceback and proposed traceback schemes to reconstruct the attack path in MANETs. Again, the communication overhead in the network in the major drawback. Recently, Hsu et al. [33] proposed a hotspot-based traceback protocol for MANETs, which divides a forwarding path dynamically into multiple smaller interweaving fragments. Unlike previous traceback schemes, our proposed *AK-PPM* scheme is much light-weighted and is secure in untrusted MANETs. It places little computation workload to the intermediate nodes. Once an attack is identified, no additional query is required to reconstruct the attack path.

In addition, [34] and [27] adopt multiple marks but neither takes mobility into consideration. In [34], the authors proposed a router stamping scheme for wired networks, and discussed the tradeoff between the number of marks allowed in a packet and the performance of identifying the u_1 node on the path. However, all intermediate nodes on the attack path are assumed trusted and no security mechanism was presented to protect marks. Also, path reconstruction was not discussed in this work. In [27], the authors proposed a Probabilistic Nested Marking approach for traceback in sensor networks, which protects the integrity of multiple marks stored in packets using cryptographic techniques. However, the number of marks in the proposed scheme is unrestricted, making it difficult to packet format design. When the path is long, the large amount of marks will take too much space in the packet. Also, there exists a flaw in the proposed nested marking scheme which could allow a colluding node in the path to remove the marks in the end without being detected. We propose a new authenticated K-sized Probabilistic Packet Marking (AK-PPM) scheme, which improves the efficiency of traceback in the untrusted MANET environment.

9 Conclusion

In this paper, we made the first effort to quantitatively analyze the impacts of node mobility, attack packet rate, and path length on the traceability of well-known IP traceback schemes. We then presented an K-PPM and an AK-PPM scheme for source attribution in trusted and untrusted MANET environments, respectively. The proposed schemes can improve the efficiency of source identification and forwarding path reconstruction.

Acknowledgments. This work was supported in part by the NS-CTA grant from the Army Research Laboratory (ARL), the U.S. NSF CAREER 0643906, and the U.S. NSF Grant No. IIS-0324835. The views and conclusions contained here are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either express or implied, of ARL or NSF.

References

1. Snoeren, A.C., Kohno, T., Savage, S., Vahdat, A., Voelker, G.M.: Collaborative research: Nets-find: Privacy-preserving attribution and provenence. Technical report, University of California, San Diego and University of Washington (2010)
2. Hunker, J., Hutchinson, B., Margulies, J.: Role and challenges for sufficient cyber-attack attribution. Technical report, Instituite for Information Infrastructure Protection (2008)
3. Mirkovic, J., Reiher, P.: A taxonomy of ddos attack and ddos defense mechanisms. SIGCOMM Comput. Commun. Rev. 34, 39–53 (2004)
4. Ye, F., Luo, H., Lu, S., Zhang, L.: Statistical en-route filtering of injected false data in sensor networks. In: Proc. of Infocom (2004)
5. Wang, X., Govindan, K., Mohapatra, P.: Provenance-based information trustworthiness evaluation in multi-hop networks. In: Proc. of GLOBECOM 2010 (2010)
6. Dean, D., Franklin, M., Stubblefield, A.: An Algebraic Approach to IP Traceback. ACM Trans. on Information and System Security 5, 119–137 (2002)
7. Snoeren, A., Partridge, C., Sanchez, L., Jones, C., Tchakountio, F., Kent, S., Strayer, W.: Hash-Based IP traceback. In: Proc. of the ACM SIGCOMM, pp. 3–14 (2001)
8. Song, D.X., Perrig, A.: Advanced and authenticated marking schemes for IP traceback. In: IEEE Infocom 2001, pp. 878–886 (2001)
9. Savage, S., Wetherall, D., Karlin, A., Anderson, T.: Network support for IP traceback. ACM Trans. on Networking 9(3), 226–237 (2001)
10. Sung, M., Xu, J., Li, J., Li, L.: Large-scale ip traceback in high-speed internet: practical techniques and information-theoretic foundation. IEEE/ACM Trans. Netw. 16, 1253–1266 (2008)
11. Jeong, J., Guo, S., Gu, Y., He, T., Du, D.: TBD: Trajectory-Based Data Forwarding for Light-Traffic Vehicular Networks. In: ICDCS 2009, pp. 743–757 (2009)
12. Thing, V., Lee, H.: Ip traceback for wireless ad-hoc networks. In: Proc. of Vehicular Technology Conference, VTC 2004-Fall (2004)
13. an Huang, Y., Lee, W.: Hotspot-based traceback for mobile ad hoc networks. In: Proc. of WiSec 2005, pp. 43–54 (2005)
14. Kim, Y., Helmy, A.: SWAT: Small world-based attacker traceback in ad-hoc networks. In: Proc. of MobiQuitous 2005, pp. 85–96 (2005)
15. Liu, D., Ning, P.: Establishing pairwise keys in distributed sensor networks. In: Proc. of the ACM Conference on Computer and Communications Security 2003, pp. 52–61 (2003)
16. Du, W., Deng, J., Han, Y., Varshney, P.: A pairwise key pre-distribution scheme for wireless sensor networks. In: Proc. of CCS 2003, pp. 42–51 (2003)
17. Blundo, C., Santis, A., Herzberg, A., Kutten, S., Vaccaro, U., Yung, M.: Perfectly-Secure Key Distribution for Dynamic Conferences. In: Brickell, E.F. (ed.) CRYPTO 1992. LNCS, vol. 740, pp. 471–486. Springer, Heidelberg (1993)
18. Zhu, S., Xu, S., Setia, S., Jajodia, S.: LHAP: a lightweight network access control protocol for ad hoc networks. J. of Ad Hoc Networks 4, 567–585 (2006)
19. Sourcefire, Inc.: Snort, http://www.snort.org/
20. Marti, S., Giuli, T., Lai, K., Baker, M.: Mitigating routing misbehavior in mobile ad hoc networks. In: Proc. of MobiCom 2000, pp. 255–265 (2000)
21. Savage, S., Wetherall, D., Karlin, A., Anderson, T.: Practical network support for ip traceback. SIGCOMM Comput. Commun. Rev. 30, 295–306 (2000)
22. Sadagopan, N., Bai, F., Krishnamachari, B., Helmy, A.: Paths: analysis of path duration statistics and their impact on reactive manet routing protocols. In: Proc. of MobiHoc 2003, pp. 245–256 (2003)

23. Feller, W.: An Introduction to Probability Theory and Applications, 3rd edn., vol. 1. John Wiley & Sons Publishers, New York (1968); vol. 2, 2nd edn. (1971)
24. Kuzmanovic, A., Knightly, E.W.: Low-rate tcp-targeted denial of service attacks. In: Proc. of SIGCOMM 2003, pp. 75–86 (2003)
25. Adler, M.: Tradeoffs in probabilistic packet marking for IP traceback. In: Proc. of STOC 2002, pp. 407–418 (2002)
26. Goodrich, M.: Efficient packet marking for large-scale IP traceback. In: Proc. of the 9th ACM CCS Conference, pp. 117–126 (2002)
27. Ye, F., Yang, H., Liu, Z.: Catching ”moles” in sensor networks. In: Proc. of ICDCS 2007, p. 69 (2007)
28. Stajano, F., Anderson, R.: The resurrecting duckling: security issues for ubiquitous computing. Computer, 22–26 (2002)
29. John, E., Kobza, S.H.J., Vaughan, D.E.: A survey of the coupon collectors problem with random sample sizes. Methodology and Comp. in Applied Probability 9, 1387–5841 (2007)
30. Sellke, T.M.: How many iid samples does it take to see all the balls in a box? The Annals of Applied Probability 5, 294–309 (1995)
31. Zarai, F., Rekhis, S., Boudriga, N., Zidane, K.: Sdppm: An ip traceback scheme for manet. In: Proc. of ICECS 2005, pp. 1–4 (2005)
32. Kim, I.Y., Kim, K.C.: A resource-efficient ip traceback technique for mobile ad-hoc networks based on time-tagged bloom filter. In: Proc. of ICCIT 2008, pp. 549–554 (2008)
33. Hsu, H., Sencun Zhu, A.H.: A hotspot-based protocol for attack traceback in mobile ad hoc networks. In: Proc. of ASIACCS 2010, pp. 333–336 (2010)
34. Thomas, W., Doeppner, P.N., Klein, A.K.: Using router stamping to identify the source of ip packets. In: Proc. of CCS 2000, pp. 184–189 (2000)

Paying for Piracy? An Analysis of One-Click Hosters' Controversial Reward Schemes

Tobias Lauinger[1], Engin Kirda[1], and Pietro Michiardi[2]

[1] Northeastern University, Boston, USA
[2] Eurécom, Sophia-Antipolis, France

Abstract. One-Click Hosters (OCHs) such as Rapidshare and now defunct Megaupload are popular services where users can upload and store large files. Uploaders can then share the files with friends or make them publicly available by publishing the download links in separate directories, so-called direct download or streaming sites. While OCHs have legitimate use cases, they are also frequently used to distribute pirated content. Many OCHs operate affiliate programmes to financially reward the uploaders of popular files. These affiliate programmes are controversial for allegedly financing piracy, and they were prominently cited in the criminal indictment that lead to the shutdown of Megaupload, once among the world's 100 largest web sites. In this paper, we provide insights into how much money uploaders of pirated content could earn on a range of direct download and streaming sites. While the potential earnings of a few uploaders are non-negligible, for most uploaders these amounts are so low that they cannot rationally explain profit-oriented behaviour.

Keywords: One-Click Hosting, Piracy, Uploader Income, Affiliate Programmes.

1 Introduction

Piracy is the most common illicit activity on the Internet. Every day, millions of people use P2P networks or One-Click Hosters (OCHs) such as Hotfile, Rapidshare and formerly Megaupload to share copyrighted content without permission. File sharing based on OCH works in a division of labour: OCHs provide the storage but no search functionality, and external direct download or streaming sites host searchable repositories of download links pointing to the OCHs.

OCHs are large businesses financed through advertisement and subscription fees; several of them are among the 100 largest web sites worldwide. Because OCHs have various legitimate use cases, they claim immunity against their users' copyright infringements under the U.S. Digital Millennium Copyright Act.

However, many OCHs also operate controversial affiliate programmes in order to attract new paying members. These affiliate programmes financially reward uploaders based on the number of downloads and member subscriptions that they generate. For instance, Megaupload used to reward one million downloads with $ 1,500 and WUpload used to pay up to $ 40 per one thousand downloads. These affiliate programmes are controversial for allegedly encouraging users to

D. Balzarotti, S. Stolfo, and M. Cova (Eds.): RAID 2012, LNCS 7462, pp. 169–189, 2012.

upload copyrighted content and thereby funding piracy. For instance, Megaupload's former affiliate programme and their knowledge that affiliates uploaded pirated content were a central element of the criminal indictment[1] that lead to the seizure of Megaupload's assets, the detention of its operators, and the shutdown of the site on 19 January 2012.

In this paper, we investigate how much money uploaders can earn by illegally uploading pirated content and posting download links on a range of direct download and streaming sites. The order of magnitude of an uploader's income tells us whether the affiliate programme and the associated rewards should be considered as a major factor in the uploader's motivation, or if they could be seen as just a minor concomitant effect.

Measuring uploader income is a challenging task: Almost no OCH reports how often a file was downloaded, and most direct download and streaming sites do not display how often a download link was clicked. Furthermore, even if these data are known, nothing reveals whether an uploader actually participates in an OCH's affiliate programme.

We tackle this problem in the following way: We crawl three large direct download/streaming sites that make click data available. Using the click data, we compute an uploader's maximum income for the links posted on the site under the assumption that every click generated a valid download, and that the uploader participated in the affiliate programme. In order to estimate how many clicks correspond to an actual download, we correlate the click data with the number of downloads on the few OCHs that make download data available.

Our results show that most uploaders earn next to nothing; they do not exhibit apparent profit-oriented behaviour. However, we also observe that a handful of uploaders upload large numbers of files each day and generate so much traffic that they could earn up to a few hundred dollars per day. For these uploaders, at least some degree of profit-oriented behaviour is probable.

Our findings have implications on proposed anti-piracy measures such as the U.S. draft bill SOPA and similar projects in other countries that aim at interrupting the revenue stream of piracy: Such measures, by definition, can affect only profit-oriented actors. Given that we observe a large number of altruistic uploaders, these measures run the risk of having only little effect overall.

In this paper, we make the following contributions:

- We are the first to use large-scale empirical data to estimate the distribution of uploader income through affiliate programmes. We contrast the income with indicators for the effort invested by uploaders. This tells us about the motivations of uploaders with respect to profit seeking or altruism.
- We are the first to provide insights into how the shutdown of Megaupload and the associated cancellations of other OCHs' affiliate programmes affected illegal uploader income. This gives us ground truth to judge the success of anti-piracy measures that aim to curb piracy by removing financial incentives.

[1] Superseding indictment, U.S. v. Kim Dotcom et al., 1:12-cr-00003-LO (E.D. Va., Feb. 16, 2012) at ¶ 58; ¶ 73 g–j, v, y, bb, jj, pp, qq, uu, ppp, qqq, www, xxx; and ¶ 102.

2 Background

One-Click Hosters (OCHs) have various legitimate use cases, such as storing backups or exchanging large files instead of sending them as email attachments. Because the purpose of this paper is to measure illegal uploader income relating to piracy, we focus the background information given in this section on illicit file sharing and on ways of monetising pirated content.

2.1 OCH-Based File Sharing and Streaming

One-Click Hosters such as Rapidshare, Megaupload, Hotfile or Mediafire provide web-based storage for potentially large files. Users can upload files through a simple web interface. For each uploaded file, the OCH provides a unique download link to the uploader. Because most OCHs do not make uploaded files public or offer search capabilities, uploaders seeking to publish their files need to post the corresponding download links on third-party web sites. There is a great variety of such sites, ranging from general-purpose discussion boards and blogs to more specialised content indexing sites, so-called direct download sites. These sites offer a catalogue of links, supplied by site staff and sometimes independent users, including categories such as movies, TV shows, games, music, ebooks, and software. So-called streaming sites index movies and TV shows using an embedded video player provided by OCHs such as Megavideo, VideoBB and Putlocker. In the following, we will use the term link or indexing site to refer to all types of "underground" web sites that are specialised in supplying links to pirated content hosted on OCHs.

As Fig. 1 shows, relationships between OCHs and indexing sites can be complex: Some uploaders spread their links over many indexing sites. An individual indexing site typically contains several copies of the same content hosted on different OCHs, and sometimes even several "mirror" copies of the same file hosted on the same OCH. Instead of posting the original download link, some uploaders use URL shorteners or "link protection services". The purpose of these services is to protect download links against automated extraction by web crawlers run by copyright holders to automatically take down files that infringe their copyright. Sometimes, these services are also used to better monetise links, such as by displaying advertisements before redirecting the user to the OCH.

2.2 OCH Affiliate Programmes

One-Click Hosters usually offer a free, advertisement-based service and a premium subscription service. In order to convert free users into paying members, the free service is artificially limited in the bandwidth, and free users need to wait between consecutive downloads. According to the indictment[2], Megaupload

[2] Superseding indictment, U.S. v. Kim Dotcom et al., 1:12-cr-00003-LO (E.D. Va., Feb. 16, 2012) at ¶ 4.

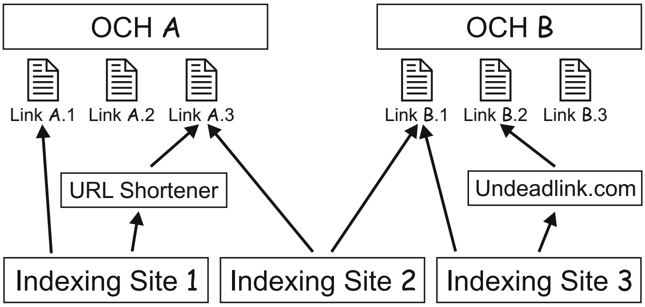


Fig. 1. The OCH ecosystem: Indexing sites can refer to a range of OCHs, the same download link can be posted on several indexing sites, mirror copies of the same file can be hosted on different OCHs or even on the same OCH, and links can be hidden behind a layer of redirection by using URL shorteners, for instance

received at least 150 million dollars in subscription fees and 25 million dollars for advertising between September 2005 and 5 January 2012.

There are hundreds of competing OCHs. In order to attract user traffic and generate membership sales, most OCHs offer affiliate programmes for uploaders and indexing sites. Affiliate programmes differ widely in the amounts paid, but they are always a combination of these basic building blocks:

Pay Per Download (PPD). A small amount of money is paid for each (full) download, such as $ 15 for 1000 downloads. Often, the amount differs according to the country of the downloader; Table 1 shows as an illustration the rates that were paid by WUpload until late November 2011. Some OCHs use different affiliate "levels" to weigh the payouts according to the past performance (which includes the conversion rate: premium sales per traffic). In most cases, only uploaders can participate in PPD programmes.

Table 1. PPD rates for WUpload, per 1000 downloads, retrieved on 30 October 2011. Country group A: *US, UK, DE*. Group B: *AU, AT, BE, CN, DK, FI, FR, IE, IT, JP, NL, NZ, NO, SA, SG, SE, CH, AE*. Group C: *BR, BG, CY, CZ, GR, HU, IR, KW, LV, LT, LU, PL, PT, QA, RO, RU, ZA, ES, TR*. Group D: *All others*. WUpload discontinued the programme in late November 2011.

Size/Country	A	B	C	D
1–50 MB	$ 5	$ 3	$ 2	$ 1
51–100 MB	$ 12	$ 8	$ 5	$ 3
101–250 MB	$ 19	$ 15	$ 12	$ 5
251–400 MB	$ 27	$ 20	$ 18	$ 7
401–2048 MB	$ 33	$ 26	$ 22	$ 10
2048+ MB	$ 40	$ 28	$ 24	$ 12

Pay Per Sale (PPS). A commission is paid for each premium sale or extension of subscription ("rebill"). The amounts paid are the same across all countries, and both uploaders and website owners can participate. For instance, WUpload used to reward uploaders with 70 % of new premium subscriptions in their PPS-only affiliate programme. Webmasters could earn 10 % of the sales to visitors that came from the webmaster's site.

Sometimes, uploaders can choose from different "formulas" such as PPD only, PPS only, or 50 % of PPD + 50 % of PPS. Not surprisingly, new OCHs tend to pay more generously, either through higher rates, or by running "promotions" during which each affiliate's payout is doubled, for instance. In the aftermath of the Megaupload shutdown, many OCHs discontinued their affiliate programmes (including VideoBB, Fileserve and Filepost), converted their affiliate programme into PPS only (Uploaded), disabled file sharing functionality (Filesonic and later WUpload) or decided to shut down voluntarily (X7).

Indexing sites can generate income through advertising, the PPS component of OCH affiliate programmes, by uploading files themselves (and fully leveraging OCH affiliate programmes), and sometimes by collecting donations. For the purpose of this paper, however, the revenue of OCHs and indexing sites is considered out of scope as we focus on uploader income through the PPD component of OCH affiliate programmes.

3 Methodology

Estimating uploader income is a difficult task because the sale and download transactions rewarded in affiliate programmes cannot easily be observed from an outsider's perspective. Sales data are kept secret by all OCHs, and only a few OCHs report the number of downloads of each file. A few indexing sites display how often a file has been "downloaded", which in reality means how often the link has been clicked.

In this paper, we focus on uploader income through the PPD component of affiliate programmes because it is the only type of income that we can measure empirically and on a large scale. We estimate uploader income by extracting the links posted on three large indexing sites along with click-through counters that are displayed on these sites. Whenever possible, we compare this data with ground-truth download data that a few OCHs supply in their APIs.

3.1 Data Sources

The income through PPD depends on the number of files an uploader has, how often each file is downloaded, and what amount the OCH pays for each download. The latter information can be obtained from the OCHs' websites since most OCHs openly advertise their affiliate programmes, if they have one, and allow any user to join. Data about the number of downloads is much more difficult to obtain; most OCHs and indexing sites do not make it publicly available.

To prepare our study, we visited the most popular indexing sites in a range of countries and checked what metadata they published. For our study, we retained three sites that counted the number of clicks of each link:

- `Dpstream.net` is the largest streaming site in France and contains movies and TV shows. For our study, we crawled the movies section only. While the site did not make any click data available, around half of the movies were hosted on VideoBB, an OCH that reports view data for their videos. (Today, the site uses a different set of OCHs.) In our analysis, we use the ground-truth view data instead of the (unavailable) click data.
- `Iload.to` is the largest direct download site in Germany (it is preceded only by two streaming sites). It consists of a directory of links that are provided by staff, a separate exchange board with user uploads, and various other community functions. We focussed on the section with staff uploads because it displayed the number of clicks of each link. The content published on the site includes movies, TV shows, music, ebooks, games, software, and pornographic material.
- `Redlist-ultimate.be` is a Belgian file sharing community with a large index of movies, TV shows, music, ebooks, games, and software. Links can be submitted by registered users only, and there are various filter rules and staff intervention to keep the index organised. Each link is annotated with additional information such as the name of the uploader and the number of clicks. The site is not as popular as the two other sites, but it publishes valuable information about registered users, such as the number of uploads and downloads, and the total time spent logged in. Out of the registered users, 79 % report France as their country.

The vast majority of the content posted on these indexing sites is being commercially exploited and is sometimes even available before the official release date in stores. During our measurements, we witnessed only a dozen content items that seemed to be shared legitimately, and their popularity was low compared to the remaining (pirated) content on the sites.

3.2 Data Sets

To obtain data sets with the links posted on indexing sites, along with the corresponding click data, we performed a series of crawls on the three indexing sites mentioned above. Table 2 lists the key characteristics of these three sites and the data sets that we extracted from them.

For `dpstream` and `redlist`, we carried out a series of *full crawls* during which we extracted all the existing content and metadata. (Our `dpstream` data set is restricted to VideoBB links in the movies section of the site.) We repeatedly performed full crawls during one month. For `redlist`, we performed an additional series of crawls in March 2012, slightly less than two months after Megaupload had been shut down, to assess the impact of this event on the file sharing ecosystem.

Due to the very high number of content objects (movies, TV show episodes etc.) indexed on `iload`, a full crawl would have taken too long to complete.

Table 2. The indexing sites crawled for this study and the types of data available on these sites. Media content is broken down into downloads and streams. Click data is provided by the indexing site; OCH views (or downloads) are ground truth collected from the respective OCH. `Dpstream` is limited to movies hosted on VideoBB and uses OCH views instead of click data. For `iload`, clicks and payout refer to the first 30 days in the lifetime of all objects that are added on a single day.

Name	`dpstream`	`iload`	`redlist-oct`	`redlist-mar`
Alexa Rank	1507 *(FR: 70)*	2976 *(DE: 144)*	15405 *(FR: 735)*	
Downloads	✗	✓	✓	
Streams	✓	few	✗	
Crawl Start	18 Feb 2011	4 Apr 2011	3 Oct 2011	8 Mar 2012
Crawl End	23 Mar 2011	10 Jul 2011	5 Nov 2011	22 Mar 2012
Crawl Type	full	new content	full	
Click Data	✗	✓	✓	
Uploader Data	✓	few	✓	
OCH Views	✓	few	✗	few
# Content	10,950 total	421 added/day	114,475 total	43,418 total
# Links	11,026 total	7,674 added/day	358,297 total	109,492 total
# Clicks/day	16,349	223,691 (future)	140,996	148,090
$ Payout/day	32.70	1,010.50 (future)	184.79	1028.48
Comments	films/VideoBB	future 30 days	pre/post Mega* shutdown	

Instead, we crawled only the *new content* that was added to the site: We requested the site's RSS feed every hour to discover new content. At the same time, for all discovered objects, we periodically (and repeatedly) retrieved the associated pages to track the evolution of the number of clicks. We ran this experiment for around three months, until `iload` stopped publishing click data.

Our crawler was capable of detecting more than 500 different link types from 300 different OCHs. In order not to distort the click count when extracting links from the indexing sites, the crawler kept track of its requests and we adjusted the final click data accordingly. For each discovered link that referred to an OCH that made download data available, we furthermore retrieved the number of downloads from the OCH's API every two days.

To extract information about the OCHs' affiliate programmes, we visited the websites of more than 50 OCHs used on the three indexing sites in October 2011 and again in March 2012. Several OCHs modified their affiliate programmes during our study. For instance, Megaupload discontinued their affiliate programme in summer 2011, thus we use their rates for `iload` but not for `redlist`.

The amounts paid per download are often differentiated by the file size and by the country of the downloader, as illustrated for WUpload in Tab. 1. To look up a consistent payout value for all files, we make the following assumptions: For links found on `dpstream` and `redlist`, we assume all downloaders to be located in France; for `iload`, we use the payout amounts for Germany. These assumptions correspond to the countries where most of the sites' users come from.

We furthermore assume a constant file size of 101 MB because this is a common (and conservative) value on file sharing sites [9]. For streaming links, we assume a video length of 90 minutes because most of the streaming links found in our data sets correspond to movies. (Most TV shows only have download links.)

3.3 Ethics

All the data in our data sets was collected from public sources that are accessible to every Internet user. Our data sets contain no IP addresses or real names; the most private information that we possess are the (publicly visible) user names of the users who posted links, and in some cases the user names of the file uploaders. However, these names are freely chosen by the users and we have no means to map these user names to a real-world identity. Therefore, our analysis does not negatively affect the privacy of any individual uploader.

3.4 Metrics

The direct way to infer the income of uploaders is to use view or download data supplied by the OCH and multiply it with the PPD amount. Unfortunately, only the `dpstream` data set has a representative amount of OCH-provided view data. For the other data sets, we infer the income indirectly through the number of clicks observed on indexing sites.

To approximate an uploader's income generated by PPD programmes, we define the *value* v of a link $l \in L$ as follows:

$$v_\alpha(l) = \text{clicks}(l) \times \alpha \times \text{payout}(\text{och}(l)) \enspace , \tag{1}$$

where clicks(l) is the amount of clicks reported on the indexing site for a given time frame, α is the click-download ratio, that is, the fraction of clicks that result in a valid download, and payout(och(l)) is the amount of money paid by the OCH of the link for one download. Note that the value of a link is different from the uploader's income because it refers to *potential* income that depends, among others, on the actual value of α. We discuss this issue in more detail in Sect. 3.5.

We express the number of clicks as daily averages. For `dpstream` and `redlist`, our data sets contain a sequence of full crawls, as shown in Fig. 2 for two crawls. In the regular case, we have one observation of c_d clicks in the first crawl at time t_d, and another observation of $c_e \geq c_d$ in the second crawl at time t_e. We compute the average number of clicks per day as $\frac{c_e - c_d}{t_e - t_d}$. Note that we consider only links present in the first crawl; links that are added at a later time will be discarded. Similarly, if a link is deleted before we can take a second snapshot, we cannot compute the number of clicks. On `redlist`, a full crawl took between six and ten days to complete.

The sites that we have crawled contain tens to hundreds of thousands of links, and not all of the links receive a click between two successive crawls. Therefore, we use the first crawl to determine the set of links that will be considered, and

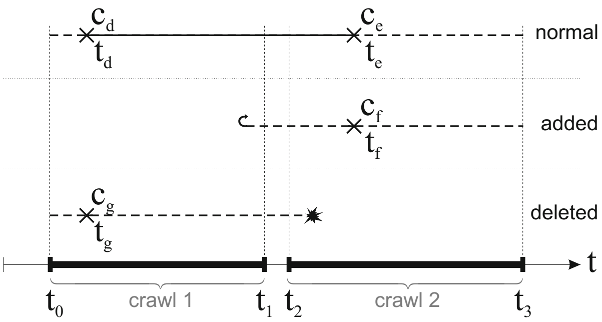


Fig. 2. Click computation for *full* crawls (`dpstream` and `redlist`)

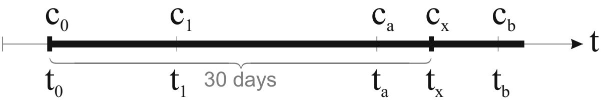


Fig. 3. Click computation for crawls for *new* content (`iload`)

the click counts observed in the *last* crawl to compute daily averages[3]. This trade-off permits us to improve accuracy for unpopular content while not penalising popular content (with quickly decaying popularity) too much.

While the data for `dpstream` and `redlist` covers the existing content on the site (both old and new), the data set for `iload` contains only new links that were added to the site. For this site, we use a different definition of "average daily clicks". As diagrammed in Fig. 3, we start tracking a new link when it is published at time t_0, and we take successive snapshots of the number of clicks c_i at time t_i, $i \geq 0$. Our goal is to estimate how many clicks c_x a new link generates in the first $t_x = 30$ days of its lifetime. In contrast to the full crawls, the click count snapshots are taken in different time intervals, according to the degree of utilisation of the crawler. In order to obtain an accurate estimate of c_x, we perform linear interpolation between the latest click count c_a observed before t_x, and the earliest click count c_b observed after t_x. The estimated value for the click count after thirty days is then $c_x = c_a + (c_b - c_a) \cdot \frac{t_x - t_a}{t_b - t_a}$.[4] This metric defines the value of a link with respect to the number of clicks that the link will generate in the first thirty days of its lifetime. We can use this metric to compute for each day how much *future value* an uploader generates by adding new links to the site. We can furthermore average over all days to obtain the *daily future value* generated by adding new links to the site.

To summarise these metrics, for `dpstream` and `redlist`, we compute for each existing link how many clicks it receives per day. On `iload`, we characterise the

[3] In the case that a link is deleted in the meantime, we use the latest click observation that we have, but divide by the total time span between the first and the last crawl, that is, around 23 days for `redlist-oct`.

[4] If the link is deleted before t_x and we have no observation c_b, we simply use $c_x = c_a$.

dynamics of the page by computing not only how many links are added to the site each day, but also how many clicks these new links generate in the first thirty days of their lifetime.

3.5 Limitations

Due to the methodology we have chosen, we can compute the distribution of uploader income, but we cannot know if an uploader actually participates in the affiliate programme. Yet, previous work has shown [4] how rapidly OCHs that discontinued their affiliate programmes lost user traffic, which suggests that those affiliate programmes were a driving factor behind these OCHs' popularity.

In most of our data sets, the click-download ratio α is unknown. If users post their links on various indexing sites in addition to the three sites of our data sets, it is possible that $\alpha > 1$. On the other hand, $\alpha < 1$ if visitors click on the link without downloading the file, as it can happen when the file was deleted from the OCH. Furthermore, some OCHs count only completed downloads and take into account only one download per day and IP address. We address this issue by using OCH-provided ground truth on download data in `dpstream`. For `iload` and `redlist`, we compute the *maximum value* of a link on the indexing site as the payout generated by the indexing site's traffic with $\alpha = 1$. This definition ignores the payout contribution due to traffic from other indexing sites and assumes that every click generates a download. In the few instances where both click and download data is available, we can estimate α and scale down the maximum link value to obtain a more realistic approximation.

Since content uploaders and link posters are not necessarily the same person, what we estimate in this paper is *how much the links are worth* that users post on indexing sites. We refer to this as (potential) uploader income because it is what uploaders can make if they are interested.

For practical reasons, we need to make a range of simplifying assumptions, such as a static file size and downloader country. Furthermore, we do not consider any payout threshold (which can be up to $ 200 for some OCHs) that prevents uploaders with low income from being paid. For this reason, the results that we provide in this paper should be seen as best-effort approximations that hold on the long term.

4 Results

Each of our data sets provides us with insights into different aspects of the monetisation of pirated content: `Dpstream` gives us a global view on the distribution of uploader income based on ground-truth data (Sect. 4.1). `Iload` shows the value of individual links and assesses their depreciation over time (Sect. 4.2). `Redlist` allows us to characterise individual uploaders, including the effort that they put into their activity and their importance to the functioning of the site (Sect. 4.3). The second `redlist` crawl furthermore provides us with insights into the effects of the Megaupload shutdown on the money-making opportunities in the file sharing ecosystem (Sect. 4.4).

4.1 Uploader Income

To compute the income distribution of uploaders, we consider the VideoBB links posted in the movies section of dpstream, France's largest piracy-based streaming site. VideoBB links are available for approximately half of the movies. Because we use view data retrieved directly from VideoBB, we expect our results to be very close to what VideoBB actually paid to participating affiliates.

Figure 4 ranks the site's users by their income and plots the users' share of the site-wide income and total number of VideoBB links. From a global point of view, the income is concentrated on a few uploaders. For instance, the top 4 uploaders earn more than 30 % of the total income. The top 50 users receive almost 80 % of the total income and provide around 70 % of the links. One could argue that anti-piracy measures targeting the top 50 uploaders seem promising as the site would lose a large portion of its links. While true in this specific case, we show in Sect. 4.3 that this intuition is wrong in a more general scenario.

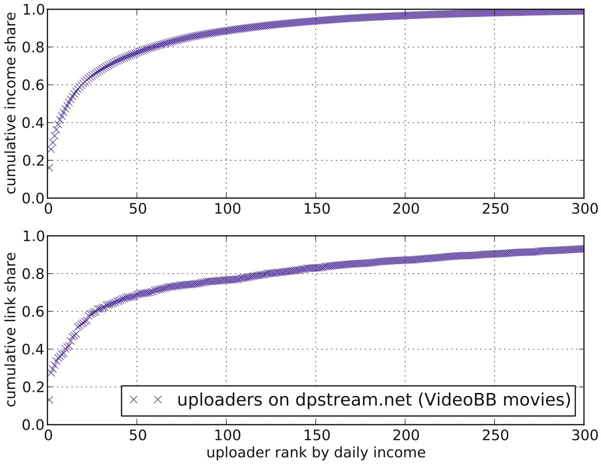


Fig. 4. Value and link share in dpstream: Associates each of the 585 uploaders, ranked by their income, with their share of the site-wide income of $ 32.70 per day (above) and the fraction of the total 11,026 links that they provided (below). A few users generate most of the value and most users earn almost nothing. While the users with the highest income also provide most links, the share of links provided is disproportionately small compared to the income share.

With a site-wide daily payout of $ 32.70, the potential earnings of individual uploaders are surprisingly low: 60 % of the users post content that is worth less than one cent per day, and even the top uploader can earn only $ 5.26 per day. While the low income in absolute terms appears to preclude profit-oriented uploader behaviour, the dpstream data set does not reveal much about the effort associated with an uploader's activity, that is, how often an uploader needs to provide new links in order to have a steady stream of revenue. Furthermore, it is unclear where uploaders are based and whether the amount of their income should be assessed according to western standards or to those of a developing country. We come back to these issues in the following sections.

4.2 The Value of a Link

We use the `iload` data set to analyse the popularity of content objects and the choice of OCH made by the uploaders. The popularity of content objects such as movies, episodes of TV shows, games or ebooks is important because only popular objects can yield any significant payout. The choice of OCH is crucial because it determines how well an object's popularity can be monetised.

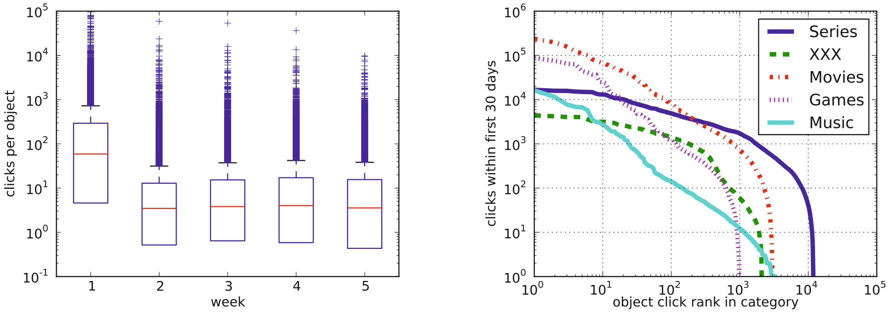
In the data set, we keep track of new objects being added to the site. Because `iload` is specialised in the timely publication of releases leaked by the Warez Scene[5], we can assume that the content is "fresh" when it is posted and analyse how its popularity evolves over time. Figure 5(a) shows a box plot of the weekly click distribution per content object. The popularity of content decreases very quickly: While the median number of clicks is 59 in the first week, it drops to less than 5 clicks per week in the following weeks. The 99th percentile drops from 4,383 in the first week to 300−366 in the following weeks. Even though the click distribution exhibits outliers that continue to receive more than 10,000 clicks per week, the vast majority of content becomes "worthless" after only one week. As a result, uploaders wishing to make money need to regularly post fresh content.

When looking at the popularity of individual objects as shown in Fig. 5(b), it becomes clear that the site posts a lot of relatively unpopular content. For instance, 25 % of all new movies receive less than 100 clicks in the first 30 days; only 3 % of the movies receive more than 10,000 clicks in the same time span. An object with a few hundred clicks per month makes a couple of dollars at most and cannot generate any noticeable income through advertising either. Note furthermore that each object uploaded on `iload` corresponds to several alternative links and mirror copies on at least a half dozen OCHs. We argue that for such objects, even if automated, the cost of uploading can hardly be amortised by the income generated by these objects. The reason for posting unpopular objects might rather be a matter of prestige.

This issue becomes even more acute at the level of granularity of individual links: Within the first 30 days, a single link can make up to $ 335.29. However, only the top 20 links achieve a potential payout of more than $ 100 in their first 30 days. The median, even if considering only links that received at least one click, is merely 2 cents for 30 days. Only by adding 421 new objects (7,674 new links) every day can `iload` achieve a significant income: For all content posted at most 30 days ago, the combined PPD income is up to $ 1,010.50 per day.

Figure 6 breaks down the recent content objects' clicks and value by OCH. Although Megaupload, Uploaded, X7 and Fileserve are the most popular OCHs with uploaders, only Megaupload is equally popular with downloaders. In fact,

[5] For an introduction to the Warez Scene, refer to [8], [7] and [3].



(a) Content hotness on `iload`: 75 % of the objects receive less than 20 clicks per week once the object is older than one week.

(b) Object popularity on `iload`: Most of the new objects receive only few clicks within the first 30 days.

Fig. 5. Content object popularity on `iload`: Popularity (a) per week and (b) by category

the OCHs most popular with downloaders pay the least competitive rates or nothing at all to uploaders. (OCHs without payout account for 24.1 % of the links and 30.6 % of the clicks.) Most links (95 %) seem to be uploaded by staff of the site and their policy to provide links to OCHs with low or no payout reduces the potential income of the site. While this finding might suggest that the site does not attempt to maximise PPD profit, one should keep in mind that the overall popularity of the site might suffer if the users' favourite OCHs are not offered, which is particularly important for users who have paid for premium services on one OCH.

So far, the value of links computed for `iload` was based on $\alpha = 1$. `Iload` posts a small number (16,553) of VideoBB streams. For these links, we estimate $\alpha \approx 0.40$ by linear regression as shown in Fig. 7(a) (correlation coefficient 0.69, $R^2 = 1$), which means that the actual payout is significantly lower than what the click count alone would suggest. In the `redlist-mar` data set, we estimate $\alpha \approx 0.73$ (correlation coefficient 0.65, $R^2 = 0.90$) for Files-Save (Fig. 7(b)) and $\alpha \approx 1.75$ (correlation coefficient 0.65, $R^2 = 0.99$) for Fufox (Fig. 7(c)). Here, $\alpha > 1$ suggests that those links are also posted on sites other than `redlist`. Note, however, that none of the latter two OCHs rewards uploaders with cash.

Many large downloads are split into smaller parts. In order to reassemble the original file, the downloader needs to download all parts of such a group of links. However, Fig. 7(d) shows that there is a difference of 32 % (correlation coefficient 0.87, $R^2 = 1$) between the link with the lowest and the link with the highest click count. In other words, only 32 % of the users who were interested in a file proceeded to download it entirely. These results illustrate that in most cases, $\alpha = 1$ induces a conservative upper bound on the actual number of downloads.

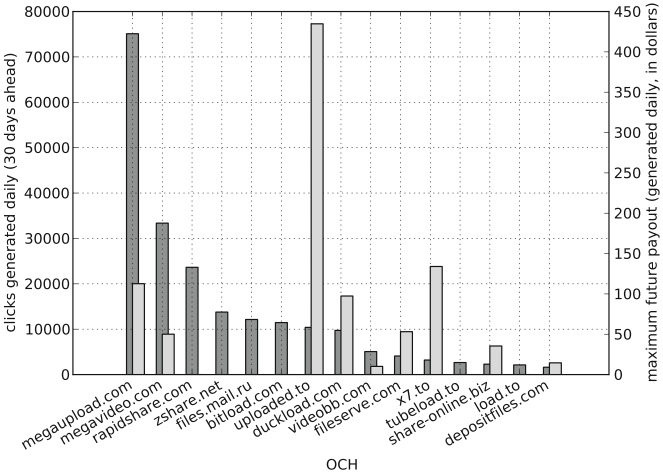


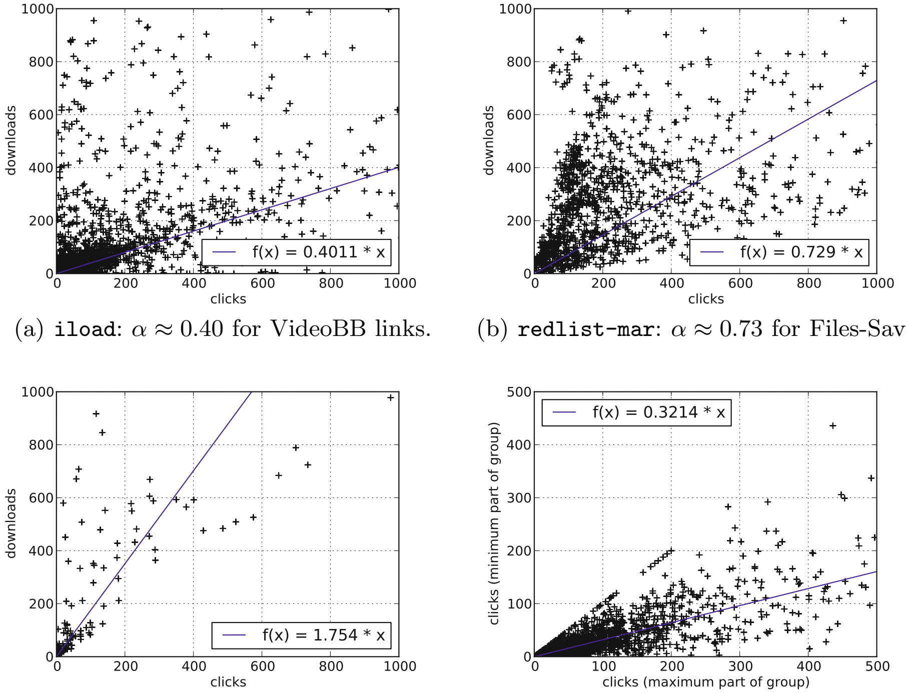
Fig. 6. Clicks (dark grey) and value (light grey) of links on iload for the top 15 OCHs: Some OCHs pay much higher rates than others, and the site also makes use of OCHs that do not pay any rewards at all

To summarise, most objects make money only for a limited time (one week) and need to be replaced regularly in order for the uploader to earn a regular income. The choice of OCHs and especially the large quantities of highly unpopular objects suggest that iload is not maximising its profitability.

4.3 Characteristics of Top Content Uploaders

Redlist contains insightful data about the users registered on the site. We use this data to answer whether uploading can be profitable—how much work in terms of uploaded files and online time the best earning uploaders carry out, and how much their potential income might be worth to them, by looking at which countries these uploaders come from. Furthermore, we investigate how essential the top uploaders are to the functioning of the site.

We use the redlist-mar data set because it is more recent and reflects better the current state of the site after the shutdown of Megaupload. It contains 101,300 registered users, out of which 7,960 logged in at least once during the week of the crawl and 275 posted at least one link. The median number of links that downloaders click on is slightly larger than the median number of links that uploaders post (Fig. 8). However, the activity distribution of uploaders is more heavy-tailed with a few uploaders posting more than 100 links every day. Similarly, 30 % of all active uploaders and 70 % of the 50 highest earning uploaders spend more than one hour logged in per day, whereas this is the case for only 4 % of the users who do not post links. These numbers illustrate that the top uploaders invest a significant effort into their activity.



(a) **iload**: $\alpha \approx 0.40$ for VideoBB links.

(b) **redlist-mar**: $\alpha \approx 0.73$ for Files-Save.

(c) **redlist-mar**: $\alpha \approx 1.75$ for Fufox links.

(d) **redlist-mar**: $\alpha \approx 0.32$ for groups.

Fig. 7. Estimating the click-download ratio α by linear correlation: (a)–(c) use ground truth obtained from the OCH, (d) uses the difference between the most and least frequently clicked link in multipart downloads

The median income for the top 50 highest earning uploaders is $ 11.74 for a median of 1.6 hours spent logged in and 10 files posted each day. (The top uploader earns $ 113.17 for an online time of around 8 hours and 200 files uploaded each day.) While this daily income would be worthwhile for an uploader based in a developing country, Fig. 9 shows that the vast majority of uploaders come from western countries, notably France. For reference, the current minimum legal wage in France is $ 12.50 per hour. This indicates that even the top uploaders earn relatively little compared to the work that they are doing.

Table 3(c) displays the overlap between the 50 uploaders with the highest income, links and clicks, respectively. Around 36 % of the users who provide most links are not among the best earning users. The fact that these uploaders do not imitate the behaviour of the best earning uploaders suggests that even the top uploaders do not all aim to maximise their income.

To assess the importance of the 50 highest earning uploaders for content availability on the site, we count how many content objects would become unavailable if all links provided by these users were removed. This corresponds to a scenario

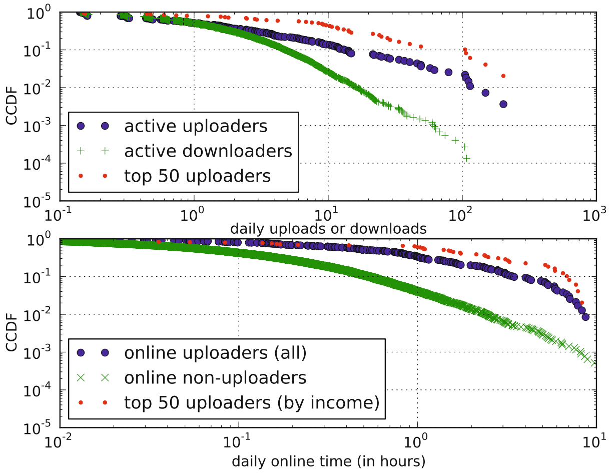


Fig. 8. User activity in `redlist-mar`: CCDF of the number of uploads and downloads per day (above) and the time spent logged in every day (below) for all users who logged in at least once during the crawl. For comparison, the plots include the 50 highest earning uploaders as an additional curve. Uploaders (and especially the top 50 uploaders) exhibit a more heavy-tailed activity than downloaders.

Table 3. Set intersections of the top 50 uploaders ranked by income (I), number of links (L) and number of clicks (C) in the two `redlist` data sets

(a) between `oct11` and `mar12`

	Income	**Links**	**Clicks**
Income	14		
Links		7	
Clicks			5

(b) within `oct11`

	Links	**Clicks**
I	21	19
L		41

(c) within `mar12`

	Links	**Clicks**
I	32	42
L		37

where the 50 highest earning uploaders stop uploading when the corresponding OCHs discontinue their affiliate programmes. We find that excluding the top 50 uploaders would remove 80 % of the total income and 58.5 % of all links, but only 39.7 % of the content objects and 21.7 % of the traffic: Many content objects have alternative download links provided by other users, and the content objects that have only links provided by the top uploaders are relatively unpopular overall. Consequently, anti-piracy measures aimed at disrupting economic upload incentives would have a limited effect on this site.

In summary, even most of the top 50 highest earning uploaders earn less than the minimum legal wage in their home country. Furthermore, `redlist` is rather resilient against the exclusion of its top 50 uploaders.

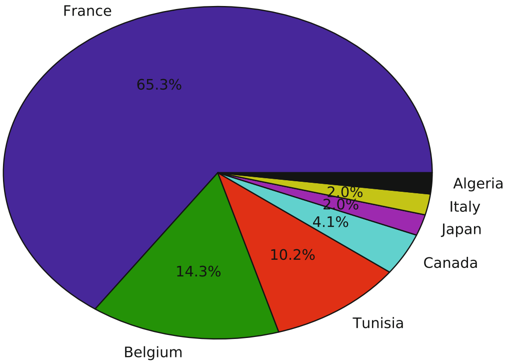


Fig. 9. Top 50 uploader's countries in `redlist-mar`: The vast majority of the highest earning uploaders come from western countries

4.4 The Impact of the Megaupload Shutdown

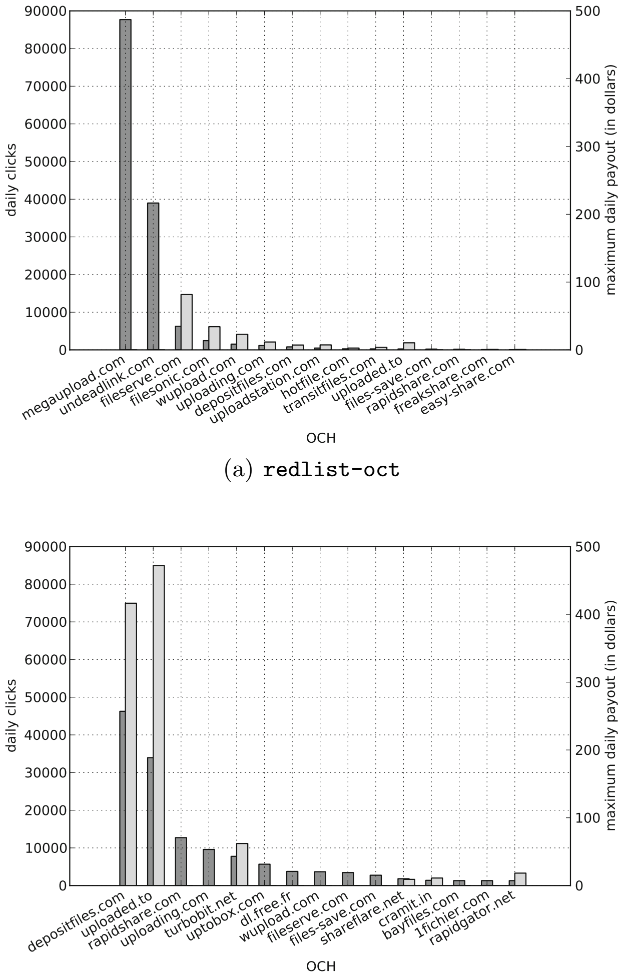
In October 2011, before the shutdown, Megaupload was the most popular OCH on `redlist` as shown in Fig. 10(a). Because Megaupload had already ended its affiliate programme at that time, `redlist` generated just $ 184.79 per day.

After the shutdown of Megaupload, several other OCHs discontinued their affiliate programmes. Figure 10(b) shows that in March 2012, `redlist` used more OCHs than before that did not pay any rewards at all. Rapidshare, an OCH that had previously lost popularity due to its anti-piracy measures [4], regained significant popularity. Somewhat paradoxically, however, the shutdown of Megaupload lead to a more than fivefold increase in the daily income (up to $ 1,028.48) because Depositfiles and Uploaded, two OCHs with competitive affiliate programmes, became the two most popular OCHs on the site.

Overall, the number of available content objects decreased drastically by 62 % after Megaupload was closed, but the site quickly recovered and even increased its total click traffic by 5 %. These events illustrate that in the OCH ecosystem with its current diversity, even the shutdown of a major actor does not durably slow down the pace of file sharing.

5 Discussion

Our measurements show that the potential income of most uploaders is very low. Hence, these uploaders must have a different incentive rather than money. On the other hand, a few uploaders can earn significant amounts of money. This mix of uploader motivations has implications on proposed anti-piracy measures.



(a) `redlist-oct`

(b) `redlist-mar`

Fig. 10. The daily number of clicks (dark grey) and daily maximum payout (light grey) for the Top 15 OCHs on `redlist` (a) before and (b) after the shutdown of Megaupload. Note that `dl.free.fr` is not shown in `redlist-oct` because the crawler did not recognise these links at that time.

A recent focus in copyright enforcement appears to be on money flows [5]. For instance, the Stop Online Piracy Act (SOPA) proposed in the U.S. contains provisions to prevent advertising and payment services from processing payments in relation to online piracy. While principally aimed at site operators, profit-oriented uploaders are indirectly affected by their dependence on affiliate programmes. However, as we have shown in this paper, profit-oriented uploaders are a small minority of all uploaders, and they are not essential for the ecosystem to survive. The majoritarian altruistic uploaders are not affected by this class of measures as long as sites remain available where they can upload and share their files.

More generally, our findings suggest that the overall impact of the OCHs' affiliate programmes on piracy activities may be overstated: Most users upload content despite earning next to nothing. Discontinuation of the affiliate programmes would deprive profit-oriented pirates of their illegal income, but it seems that these programmes are not the *main* driving force behind OCH-based piracy.

6 Related Work

Previous work in the area of OCH [1], [6], [9] focusses on network and workload-level measurements such as file sizes, download speeds, and the service architecture. These studies are partially based on network traces, and partially on crawls of indexing sites similar to our work. While some of the works depict OCH as an emerging alternative to BitTorrent for piracy, they do not deal with money-making opportunities or uploader motivations.

The closest work to ours is a short technical report published recently by Zubin Jelveh and Keith Ross [4]. The authors use payment screenshots posted in a webmaster forum to analyse the range of uploader income through Filesonic's PPD and PPS affiliate programme. In contrast to our work, the results by Jelveh and Ross reflect actual payouts. Based on 151 earnings screenshots covering 2,653 days, they report an average uploader income of $ 33.69 per day (minimum $ 0, maximum $ 226.27). This income range is generally consistent with what we find in our study. Beyond what we can analyse with our methodology, Jelveh and Ross find that income through PPD averages $ 21.12 as opposed to $ 46.10 through PPS. While providing actual ground truth data, the data set analysed by Jelveh and Ross suffers from a selection bias: Most uploaders do not make their income public. Furthermore, it is unknown where and how often uploaders post links, and what content they upload. While our methodology can only give an estimation of the actual uploader income, we compute a much more representative distribution of the income over uploaders. Furthermore, we use more comprehensive information about uploaders and content to calculate the value of links and the effort behind uploading, and we thereby obtain hints at the financial or altruistic motivations of uploaders.

Cuevas et al. [2] study the characteristics of initial seeders in BitTorrent. They find evidence for major initial content uploaders behaving in a non-altruistic way. Their result differs from our work in two ways: Firstly, BitTorrent does not have a direct mechanism to financially reward content uploaders; profit is usually generated by using uploads as a way to advertise external websites or to distribute malware. Secondly, it is common for OCH-based uploaders to copy a file and re-upload it on the same (or another) OCH. Therefore, OCH indexing sites often have a high number of alternative downloads for the same content, which decreases the potential income for individual uploaders.

For a recent news article [5], Joe Karaganis conducted an anonymous income survey among BitTorrent site operators. Summarising the results, Karaganis characterises these sites as "financially fragile but low cost operations, dependent on volunteer labor, subsidized by users and founders, and characterized by a strong sense of mission to make work more widely available within fan communities".

7 Conclusion

There is no black and white answer to uploader income in OCH-based file sharing: Most uploaders can earn only trivial amounts of money through OCHs' affiliate programmes and can be characterised as altruistic. A small number of very active uploaders, however, can earn in the order of hundred dollars per day and are more likely to be motivated by financial gain. Yet, the OCH file sharing ecosystem does not depend on these uploaders; most of the popular content would remain available if the links provided by the highest earning uploaders were excluded. The implication is that even in the OCH ecosystem with its money-mad reputation, anti-piracy measures that are premised on profit-driven uploader behaviour might not be as effective as the content industry believes. In order to sustainably address piracy, a holistic approach would be required that also removes incentives for altruistic uploaders, and for downloaders in general.

Acknowledgement. The authors thank Sy and Laurie Sternberg for their generous support.

References

1. Antoniades, D., Markatos, E., Dovrolis, C.: One-click hosting services: A file-sharing hideout. In: Proc. IMC 2009 (November 2009)
2. Cuevas, R., Kryczka, M., Cuevas, A., Kaune, S., Guerrero, C., Rejaie, R.: Is content publishing in BitTorrent altruistic or profit-driven? In: Proc. Co-NEXT 2010 (November 2010)
3. Howe, J.: The shadow Internet. Wired 13(01) (January 2005)
4. Jelveh, Z., Ross, K.: Profiting from piracy: A preliminary analysis of the economics of the cyberlocker ecosystem. Tech. rep., NYU-Poly, Brooklyn, USA (January 2012)

5. Karaganis, J.: Meganomics: The future of "follow-the-money" copyright enforcement (January 2012), http://torrentfreak.com/meganomics-the-future-of-follow-the-money-copyright-enforcement-120124/
6. Mahanti, A., Williamson, C., Carlsson, N., Arlitt, M.: Characterizing the file hosting ecosystem: A view from the edge. In: Proc. IFIP PERFORMANCE (October 2011)
7. McCandless, D.: Warez world. Telepolis (July 2001)
8. Rehn, A.: The politics of contraband: The honor economies of the warez scene. Journal of Socio-Economics 33(3), 359–374 (2004)
9. Sanjuàs-Cuxart, J., Barlet-Ros, P., Solé-Pareta, J.: Measurement based analysis of one-click file hosting services. Journal of Network and Systems Management (May 2011)

Proactive Discovery of Phishing Related Domain Names

Samuel Marchal, Jérôme François, Radu State, and Thomas Engel

SnT - University of Luxembourg, Luxembourg,
firstname.lastame@uni.lu

Abstract. Phishing is an important security issue to the Internet, which has a significant economic impact. The main solution to counteract this threat is currently reactive blacklisting; however, as phishing attacks are mainly performed over short periods of time, reactive methods are too slow. As a result, new approaches to early identify malicious websites are needed. In this paper a new proactive discovery of phishing related domain names is introduced. We mainly focus on the automated detection of possible domain registrations for malicious activities. We leverage techniques coming from natural language modelling in order to build pro-active blacklists. The entries in this list are built using language models and vocabularies encountered in phishing related activities - "secure", "banking", brand names, etc. Once a pro-active blacklist is created, ongoing and daily monitoring of only these domains can lead to the efficient detection of phishing web sites.

Keywords: phishing, blacklisting, DNS probing, natural language.

1 Introduction

The usage of e-commerce, e-banking and other e-services is already current practice in the life of modern Internet users. These services handle personal and confidential user data (login, password, account number, credit card number, etc.) that is very sensitive. As a result, threats emerged for which attackers attempt to steal this data and use it for lucrative purposes. An example of these threats is phishing, a criminal mechanism employing both *technical subterfuge* and *social engineering* to abuse the naivety of uninformed users. Phishing mainly targets (75%) financial and payment activities and its cost is estimated to many billion of dollars per year[1].

Phishing attacks leverage some techniques such as e-mail spoofing or DNS cache poisoning to misdirect users to fake websites. Attackers also plant crimeware directly onto legitimate web server to steal users data. However, the two last techniques require to penetrate web servers or change registration in DNS server, which might be difficult. Most often, phishers try to lure Internet users by having them clicking on a rogue link. This link seemed to be trustworthy because it contained a brand name or some keywords such as *secure* or *protection*.

[1] http://www.brandprotect.com/resources/phishing.pdf, accessed on 04/04/12.

D. Balzarotti, S. Stolfo, and M. Cova (Eds.): RAID 2012, LNCS 7462, pp. 190–209, 2012.

Current protecting approaches rely on URL blacklists being integrated in client web browsers. This prevents users from browsing malicious URLs. Google Safe Browsing[2] or Microsoft Smart Screen[3] are two examples and their efficiency has been proved in [14]. However, as reported in [21], the average uptime of phishing attacks is around 2 days and the median uptime is only 12 hours. Due to this very short lifetime, reactive blacklisting is too slow to efficiently protect users from phishing; hence, proactive techniques must be developed. The previous report also points out that some phishing attacks involve URLs containing unique number in order to track targeted victims. The only common point between these unique URLs remains their domain name; as a result domain name blacklisting should be more efficient and useful than URL blacklisting. Moreover, it emphasizes that one maliciously registered domain name is often used in multiple phishing attacks and that each of them use thousands of individual URLs. As a result, the identification of only one phishing domain name can lead to protect Internet users from tens of thousand malicious URLs.

According to recent reports [21, 1] from the Anti Phishing Working Group (APWG), the number of phishing attacks is fast growing. Between the first half of 2010 and the first half of 2011 the number of phishing attacks raised from 48,244 to 115,472 and the number of dedicated registered domains from 4,755 to 14,650. These domains are qualified as maliciously registered domains by the APWG. These counts highlight the trend that attackers prefer to use more and more their own maliciously registered domains rather than hacked named domains for phishing purposes. Moreover, observations reveal that malicious domain names and particularly phishing ones are meaningful and composed of several words to obfuscate URLs. Attackers insert some brands or keywords that are buried in the main domain name to lure victims, as for instance in `protectionmicrosoftxpscanner.com`, `google-banking.com` or `domainsecurenethp.com`. As a result, this paper focuses on the identification of such phishing domain names that are used in URL obfuscation techniques.

This paper introduces a pro-active domain monitoring scheme that generates a list of potential domain names to track in order to identify new phishing activities. The creation of the list leverages domain name features to build a natural language model using Markov chains combined with semantic associations. We evaluate and compare these features using real malicious and legitimate datasets before testing the ability of our approach to pro-actively discover new phishing related domains.

The rest of this paper is organized as follows: Section 2 describes the design of the architecture and the steps to follow to generate malicious domain names. Section 3 introduces the datasets used for the validation and experimentation. In section 4, differences between malicious and legitimate domains are analyzed and domain name generation is tested in some real case studies. Finally, related

[2] `http://code.google.com/apis/safebrowsing/`, accessed on 04/04/12.

[3] `http://windows.microsoft.com/en-US/internet-explorer/products/ie-9/features/smartscreen-filter`, accessed on 04/04/12.

work is discussed in section 5. We conclude in section 6 and point out the further research to be done.

2 Modeling a Phisher's Language

Phishers are human and will generate names for their domains using some simple patterns. They will use names that are similar to legitimate domain names, append some other words that come from a target vocabulary and leverage some domain specific knowledge and expertise. Thus, we argue that pro-active monitoring can emulate this process and generate potential domains to be tracked permanently. This tracking can be done on a daily basis and thus detect new phishing sites. This requires however to generate domain names that are or will be involved in phishing activities. These names follow a model build on statistical features. The domain names considered in our work are composed of two parts, the top level domain (TLD) and the second level domain also called *main domain*. In this approach, the TLD can be either only one level domain "`.com`" or more "`.org.br`", we refer to Public Suffix List[4] to identify this part of the URL and the *main domain* is considered as the label preceding the TLD. For the rest of the paper these domains (main domain + TLD) will be called *two-level-domains*. Assuming a dataset containing domain names and URLs such as:

- `www.bbc.co.uk`
- `wwwen.uni.lu/snt/`
- `secan-lab.uni.lu/index.php?option=com_user&view=login`

Features are extracted only from the *two-level-domains*, which are respectively `bbc.co.uk`, with `bbc` the *main domain* and `co.uk` the TLD, for the first one and `uni.lu` for the two others, with `uni` the *main domain* and `lu` the TLD. The domain names generated are also *two-level-domains*.

2.1 Architecture

An overview of our approach is illustrated in Figure 1 where the main input is a list of known domains related to malicious activities. Based on that, the first stage (1) decomposes the name and extracts two main parts: the TLD and the *main domain*. Then, each of these two is divided into words (2). For TLD, a simple split regarding the dot character is sufficient but for the second, a real word segmentation is required to catch meaningful words. As illustrated here with a small example, *macromediasetup.com/dl.exe*, the following words are extracted:

- TLD: `com`
- *main domain*: `macro`, `media`, `set`, `up`

[4] `http://publicsuffix.org`, accessed on 08/03/12.

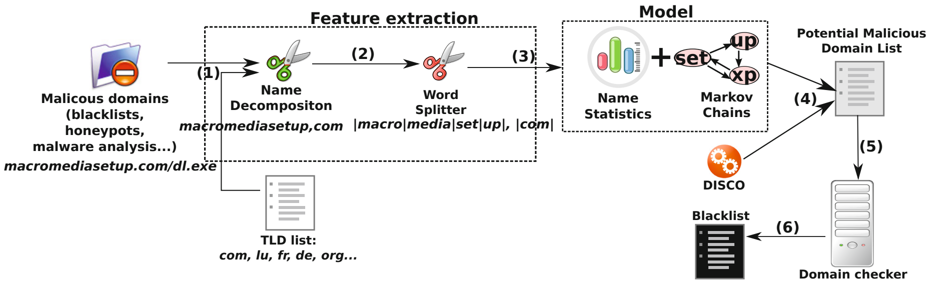


Fig. 1. Proactive Malicious Domain Name Discovery – Overview

These features are then used to build a model (3) by computing statistics, as for example the length distribution of the *main domain* in words as well as a Markov chain for representing probabilistic word transitions. The statistics and Markov chain model are computed for each level. Next, this model is combined (4) with semantic extensions. This leads to generating similar words (only for the *main domain*) and a list of potential malicious domain is built. These latter are checked online (5) for potential phishing activities. The online validation is not described in this paper, but can be done with various techniques: signature-based approach, honeypots, manual analysis, etc. Hence, our experiments are based on publicly available blacklists for cross-validation (see section 3).

2.2 Features Extraction

Features: Given a set of *two-level-domains* as $D = \{d_1, ..., d_p\}$, a set of words as $W = \{w_1; ....; w_p\}$ and a set of domain levels $L = \{l_1; l_2\}$ where l_1 is the *TLD* and l_2 is the *main domain*, we define:

- $\#len_{l,n}$ the number of domains $d \in D$ having the l^{th} level ($l \in L$) composed of n words
- $\#word_{l,w}$ the number of domains $d \in D$ containing the word $w \in W$ at the level $l \in L$
- $\#fisrtword_{l,w}$ the number of domains $d \in D$ having the l^{th} level ($l \in L$) starting with the word $w \in W$
- $\#biwords_{l,w_1,w_2}$ the number of domains $d \in D$ containing the consecutive words w_1 and w_2 ($(w_1, w_2) \in W^2$) at the level $l \in L$

The following list groups the features extracted from a list of domains or URLs:

- $distlen_{l,n}$: the distribution of the length $n \in \mathbb{N}$ expressed in word for a level $l \in L$ and defined as:

$$distlen_{l,n} = \frac{\#len_{l,n}}{\sum_{i \in \mathbb{N}} \#len_{l,i}} \tag{1}$$

- $distword_{l,w}$: the distribution of the number of occurrences of a word $w \in W$ at the level $l \in L$ and defined as:

$$distword_{l,w} = \frac{\#word_{l,w}}{\sum_{i \in W} \#word_{l,i}} \tag{2}$$

- $distfirstword_{l,w}$: the distribution of the number of occurrences of a word $w \in W$ as first word for the level $l \in L$ and defined as:

$$distfirstword_{l,w} = \frac{\#fisrtword_{l,w}}{\sum_{i \in W} \#fisrtword_{l,i}} \tag{3}$$

- $distbiwords_{l,w_1,w_2}$: the distribution of the number of occurrences of a word $w_2 \in W$ following the word $w_1 \in W$ for the level $l \in L$ and defined as:

$$distbiwords_{l,w_1,w_2} = \frac{\#biwords_{l,w_1,w_2}}{\sum_{i \in W} \#biwords_{l,w_1,i}} \tag{4}$$

Word Extraction: The *main domain* of DNS names can be composed of several words like `computeraskmore` or `cloud-anti-malware`. Using a list of separating characters, as for instance "-" is too restrictive. We have thus used a word segmentation method, similar to the one described in [22]. The process is recursive by successively dividing the label in 2 parts that give the best combination, *i.e.* with the maximum probability, of the first word and the remaining part. Therefore, a label l is divided in 2 parts for each position i and the probability is computed:

$$P(l,i) = P_{word}(pre(l,i))P(post(l,i)) \tag{5}$$

where $pre(l,i)$ returns the substring of l composed of the first i characters and $sub(l,i)$ of the remaining part. $P_{word}(w)$ returns the probability of having the word W equivalent to its frequency in a database of text samples.

TLDs are split in different labels using the separating character ".".

2.3 Domain Names Generation Model

The generator designed for domain generation is mainly based on an n-gram model. Coming from natural language processing an n-gram is a sequence of n consecutive *grams*. These *grams* are usually characters, but in our approach, *grams* are words. We especially focus on bigrams of words that are called *biwords*. These couples of words are further used to build a Markov chain through which *two-level-domains* are generated.

Markov Chain: A Markov chain is a mathematical system that undergoes transitions from one state to another. Each possible transition between two states can be taken with a transition probability. Two Markov chains are defined in the domain generation model, one for each level, l_1 and l_2. The states of the Markov chains are defined as the words $w \in W$ and the probability of transition between two words w_1 and w_2 for the level $l \in \{1; 2\}$ is given by $distbiwords_{l,w_1,w_2}$. A part of a created Markov chain is given in Table 1 for some transitions, and the associated probabilities, starting from the word *pay*. In order to generate new names the Markov chain is completed with additional transitions that have never been observed - this technique is called additive smoothing or Laplace smoothing. For each state s, a small probability (0.05) is assigned for transitions to all the words that have been observed at the level l and for which s does not have any transition yet. This probability is shared between the words of the level l according to the distribution $distword_{l,w}$. The same method is applied for states s that do not have any existing transitions. In this case, their transitions follow the probability given by the distribution $distword_{l,w}$.

Table 1. Example of Markov chain transitions for the state *pay*

Transition	per	z	for	secure	bucks	bill	process	pay	account	soft	page	...
Probability	0.13	0.1	0.06	0.06	0.06	0.03	0.03	0.03	0.03	0.03	0.03	...

For *two-level-domains* generation, the first state is randomly initialized using $distfirstword_{l,w}$, the number of transitions that must be completed in the Markov chain is randomly determined using $distlen_{l,n}$. Given these two parameters by applying n steps from word w in the Markov chain, a label is generated for the level l.

Semantic Exploration: The words composing the *main domains* of different malicious domains often belong to the one or more shared semantic fields. Given some malicious domain names such as *xpantiviruslocal.com*, *xpantivirusplaneta.com*, *xpantivirusmundo.com* and *xpantivirusterra.com*, it clearly appears that they are related. Applying the word extraction process, from all of these domains, the words “xp”, “anti” and “virus” will be extracted and the four words “local”, “planeta”, “mundo” and “terra” will be extracted from each of them. These four words are closely related, particularly the three last ones. As a result, given one of these domains, the remaining three could be found as well. However, even if this intuitive conclusion is obvious for human, it is more complicated to implement it in an automatic system.

For this purpose, DISCO [12] is leveraged, a tool based on efficient and accurate techniques to automatically give a score of relatedness between two words. To calculate this score, called similarity, DISCO defines a sliding window of four words. This window is applied to the content of a dictionary such as Wikipedia[5] and the metric $\|w, r, w'\|$ is calculated as the number of times that the word w' occur r words after the word w in the window, therefore $r \in \{-3; 3\} \setminus \{0\}$. Table 2 highlights an example of the calculation of $\|w, r, w'\|$ for two sample pieces of text. Afterwards the mutual information between w and w', $I(w, r, w')$ is defined as:

$$I(w, r, w') = log\frac{(\|w, r, w'\| - 0.95) \times \|*, r, *\|}{\|w, r, *\| \times \|*, r, w'\|} \quad (6)$$

[5] http://www.wikipedia.org, accessed on 04/04/12.

Finally, the similarity $sim(w_1, w_2)$ between two words w_1 and w_2 is given by the formulae:

$$sim(w_1, w_2) = \frac{\sum_{(r,w)\in T(w_1)\cap T(w_2)} I(w_1, r, w) + I(w_2, r, w)}{\sum_{(r,w)\in T(w_1)} I(w_1, r, w) + \sum_{(r,w)\in T(w_2)} I(w_2, r, w)} \qquad (7)$$

where $T(w)$ is all the pairs $(r, w') \mid I(w, r, w') > 0$.

Using this measure and given a word w_1, DISCO returns the x most related words ordered by their respective similarity score $sim(w_1, w_2)$. Based on the words extracted from the *main domain*, DISCO is used to compose new labels for the *main domain*.

Table 2. Example of co-occurrence counting (2 windows centered on *services*)

position	-3	-2	-1	0	+1	+2	+3
sample 1	a	client	uses	services	of	the	platform
sample 2	the	platform	provides	services	to	the	client

$\|\|services, -3, a\|\| = 1$	$\|\|services, -3, the\|\| = 1$
$\|\|services, -2, client\|\| = 1$	$\|\|services, -2, platform\|\| = 1$
$\|\|services, -1, uses\|\| = 1$	$\|\|services, -1, provides\|\| = 1$
$\|\|services, 1, of\|\| = 1$	$\|\|services, 1, to\|\| = 1$
$\|\|$**services,2,the**$\|\| =$ **2**	$\|\|services, 3, client\|\| = 1$
$\|\|services, 3, platform\|\| = 1$	

A complete example of label generation is illustrated in Figure 2 for the level 2 (*main domain*) with (1), the selection of the length of the label in words, and (2) the selection of the first word that starts the label. The Markov chain is applied for the remaining words to generate (3). For each word at the step (2) and (3), DISCO is applied to generate other words. The same scheme generates TLD for the level 1 without using DISCO.

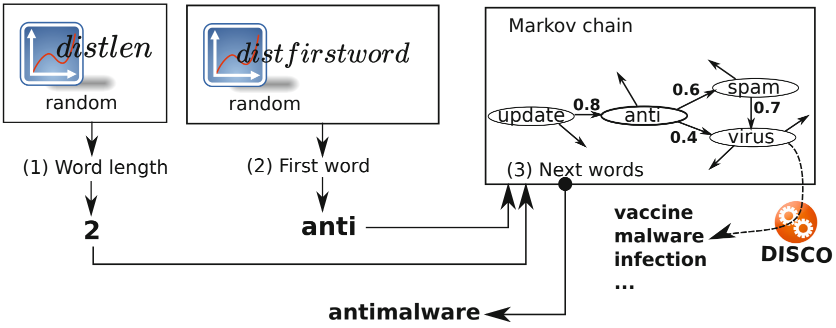


Fig. 2. Main domain generator

3 Dataset

For assessing our approach, two datasets are selected. The first one is a malicious dataset composed of domain names from which maliciousness has been confirmed. The second dataset is a legitimate dataset containing non-malicious domain names. In a first step, these will be used to show that the features introduced in section 2.2 allow to discriminate phishing domain names from legitimate ones. In a second step, the malicious dataset will be used to show the efficiency of the generation of phishing domain names.

3.1 Malicious Dataset

To compose the dataset of malicious domain names, three freely downloadable blacklists are used. These have been selected because each of them proposes an historical list of blacklisted domains ordered by their discovery date. These have been collected during at least the last three years. This is an essential dataset requirement in order to test the predictability of the approach.

- **PhishTank**[6]: PhishTank is a community website where anybody can submit a suspicious phishing URL that will be further checked and verified. The downloaded historical blacklist contains 3,738 phishing URLs.
- **DNS-BH**[7]: DNS-Black-Hole aims to maintain an up-to-date list of domains that spread malware and spyware. A list of 17,031 malicious domains is available.
- **MDL**[8]: Malware Domain List is another community project aimed at creating and maintaining a blacklist of domains involved in malware spreading. This list contains 80,828 URL entries.

DNS-BH and MDL are not only dedicated to phishing, but also to malware diffusion. These two lists have been chosen because as described in [1], diffusion of malware designed for data-stealing and particularly crimeware is a big part of phishing activities. This various dataset allows also to strengthen the validation of our approach (introduced in section 4). Following the extraction of the distinct domain names from the 101,597 URLs and the deletion of duplicated entries between the three lists, the final dataset contains 51,322 different *two-level-domains*. Out of these 51,322 domain names, 39,980 have their *main domain* divisible in at least two parts.

3.2 Legitimate Dataset

The objective is to faithfully represent realistic normal domain names. This dataset is selected to show that even if malicious domains use some brands

[6] http://www.phishtank.com, accessed on 15/03/12.

[7] http://www.malwaredomains.com, accessed on 15/03/12.

[8] http://www.malwaredomainlist.com, accessed on 15/03/12.

included in the URLs of famous websites in order to mimic them, they still disclose differences. Two sources are chosen to compose this “legitimate” dataset.

- **Alexa**[9]: Alexa is a company that collects browsing behavior information in order to report statistics about web traffic and websites ranking. From Alexa’s “top 1000000 sites” list, 40,000 domain names are randomly picked in the top 200,000 domains.
- **Passive DNS**: To diversify this dataset and in order to have the same amount of domain names in each dataset, we had it completed with 11,322 domain names extracted from DNS responses. DNS responses were passively gathered from DNS recursive servers of some Luxembourg ISPs. We ensure that these domain names are not present in the initial dataset from Alexa.

The normal dataset contains 51,322 entries. 38,712 names have their *main domain* divisible in at least two parts. Hence, we have two datasets: a *legitimate* one and a *malicious* one of equivalent size.

4 Experiments

4.1 Datasets Analysis

In this section metrics and statistical parameters extracted from each dataset are compared to demonstrate that features described before are able to distinguish malicious from legitimate domains. A first proposition is to analyze the number of words that composes the *main domain* name $\#len_{2,n}$. *Main domains* that can be split in at least two parts are considered. The *malicious* dataset contains 39,980 such domain names and the *legitimate* dataset 38,712. Figure 3 shows the distribution of the ratio of *main domains* that are composed from 2 to 10 words ($distlen_{2,n} \mid n \in \{2;10\}$) in the *legitimate* dataset and in the *malicious* dataset.

69% of legitimate *main domains* are composed of two words whereas only 50% of malicious are. For all upper values, the ratio for malicious domains is higher than for legitimate ones. This shows that malicious *main domains* tend to be composed of more words than legitimate *main domains.*

The following analysis studies the composition similarity between the domain names of the different datasets. Two probabilistic distributions are extracted from the domain names:

- the different labels of the TLDs: $\forall w \in W, P_1(w) = distword_{1,w}$
- the different words that compose the *main domains*: $\forall w \in W, P_2(w) = distword_{2,w}$

We used the Hellinger Distance to evaluate the similarity in each dataset and dissimilarity between datasets. The Hellinger Distance is a metric used to quantify

[9] http://www.alexa.com, accessed on 15/03/12.

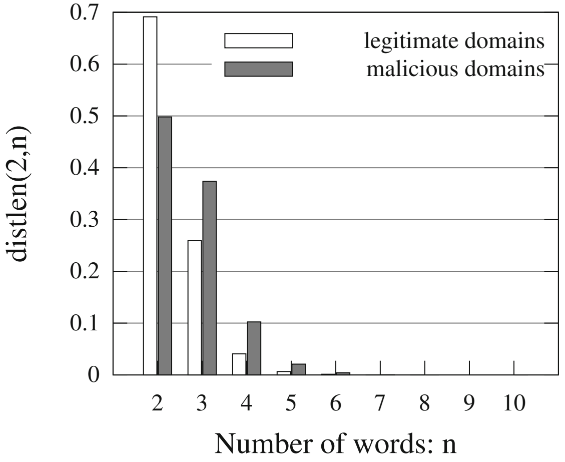


Fig. 3. $distlen_{2,n} \mid n \in \{2;10\}$ for malicious and legitimate dataset

the similarity (or dissimilarity) between two probabilistic distributions P and Q. In continuous space, the definition is:

$$H^2(P,Q) = \frac{1}{2}\int\left(\sqrt{\frac{dP}{d\lambda}} - \sqrt{\frac{dQ}{d\lambda}}\right)^2 d\lambda \tag{8}$$

The equivalent function in discrete space distribution is given by:

$$H^2(P,Q) = \frac{1}{2}\sum_{x\in P\cup Q}\left(\sqrt{P(x)} - \sqrt{Q(x)}\right)^2 \tag{9}$$

It's an instance of f-divergence as well as KL-divergence metric. Hellinger Distance is symmetric and bounded on $[0;1]$ where 1 is a total dissimilarity ($P \cap Q = \emptyset$) and where 0 means that P and Q have the same probabilistic distribution.

This metric is preferred rather than more usual metric such as Jaccard Index or KL-divergence. Jaccard Index only considers the presence or not of an element in two datasets but never considers the probability associated to an element. KL-divergence metric is an non-symmetric measure as well as unbounded function ($[0;+\infty]$). Finally, KL-divergence requires that Q includes at least the same elements of P: $\forall i P(i) > 0 \Rightarrow Q(i) > 0$. This constraint may not be satisfied with our datasets.

The *malicious* dataset and *legitimate* datasets are randomly split in five smaller subsets, respectively mal-x and leg-$x \mid x \in \{1;5\}$, of equivalent size ($\sim$ 10000 domains). Table 3 shows the Hellinger Distance for TLDs distribution between all the subset $P_1(w)$. Globally all the TLDs are quite the same in all subsets ($0 < H(P,Q) < 0.15$), a clear difference is although present in $H(P,Q)$ when P and Q are picked from the same dataset (leg/leg or mal/mal, $H(P,Q) \sim 0.015$) or from two different datasets (leg/mal, $H(P,Q) \sim 0.130$).

Table 3. Hellinger Distance for TLDs (leg=legitimate, mal=malicious)

	leg-5	leg-4	leg-3	leg-2	leg-1	mal-5	mal-4	mal-3	mal-2
mal-1	0.133	0.136	0.133	0.129	0.134	0.014	0.012	0.013	0.014
mal-2	0.134	0.140	0.135	0.131	0.135	0.014	0.012	0.013	
mal-3	0.135	0.139	0.134	0.131	0.136	0.013	0.013		
mal-4	0.130	0.136	0.131	0.127	0.132	0.013			
mal-5	0.134	0.138	0.132	0.129	0.134				
leg-1	0.017	0.017	0.018	0.019					
leg-2	0.018	0.020	0.018						
leg-3	0.016	0.019							
leg-4	0.017								

0.0 0.04 0.08 0.12 0.16 0.20

Table 4 considers the words-in-main-domain distribution $P_2(w)$. Here, the distributions are more scattered ($0.4 < H(P,Q) < 0.6$); however, difference is higher between subsets created from distinct datasets ($H(P,Q) \sim 0.56$) and subsets of the same dataset. Moreover, we can see that malicious *main domains* show more similarity between them ($H(P,Q) \sim 0.44$) than legitimate *main domains* between them ($H(P,Q) \sim 0.50$).

Table 4. Hellinger Distance for words (leg=legitimate, mal=malicious)

	leg-5	leg-4	leg-3	leg-2	leg-1	mal-5	mal-4	mal-3	mal-2
mal-1	0.564	0.571	0.561	0.566	0.565	0.446	0.439	0.443	0.438
mal-2	0.565	0.569	0.566	0.571	0.565	0.445	0.447	0.446	
mal-3	0.561	0.566	0.563	0.569	0.564	0.448	0.444		
mal-4	0.563	0.567	0.558	0.564	0.561	0.447			
mal-5	0.564	0.565	0.554	0.555	0.558				
leg-1	0.501	0.494	0.490	0.493					
leg-2	0.493	0.497	0.496						
leg-3	0.490	0.491							
leg-4	0.489								

0.4 0.44 0.48 0.52 0.56 0.6

Table 5 provides the statistics of the Markov chains for each dataset for the *main domain* level. The number of initial states is given by $Card(V) \mid \forall w \in V, \#fisrtword_{l_2,w} > 0$, the number of states corresponds to $Card(W) \mid \forall w \in W, \#words_{l_2,w} > 0$ and the number of transitions before implementation of Laplace smoothing is $Card(U^2) \mid \forall (w_1, w_2) \in U^2, \#biwords_{l_2,w_1,w_2} > 0$.

This table strengthens the assertion that words present in malicious *main domains* are more related together than those present in legitimate *main domains*, because Hellinger Distance is lower between malicious subsets compared to legitimate subsets despite the higher number of words (states) in the Markov chain created from the malicious dataset.

These experiments show that our model built on top of blacklist will be able to generate proactively maliciously registered domains with a limited impact regarding legitimate ones.

Table 5. Markov Chain statistics for *main domain*

Metrics	*Legitimate*	*Malicious*
# initial states	14079	14234
# states	23257	26987
# transitions	48609	56286

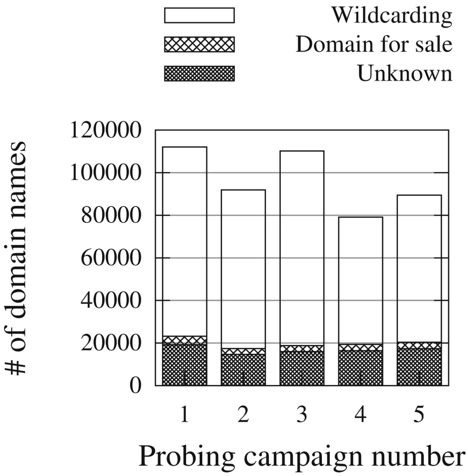
4.2 Types of Generated Domains

The dataset chosen for the rest of the experiments is the whole *malicious dataset* introduced in section 3.1. This dataset is split in two subsets and depending on the experiment performed, the domains selection technique to compose the subsets and the number of domains in each subset vary. One of these subsets is called the *training set*, from which the features described in section 2.2 are extracted in order to build the word generation system depicted in Figure 2, section 2.1. Based on it, new domain names are generated and their maliciousness is confirmed only if they belong to the second subset called the *testing set*.

The term *probing campaign* is defined as the generation of one million of unique *two-level-domains* that are checked in term of existence and maliciousness. A domain name is considered as existing if it is actually reachable over the Internet, *i.e.* it is mapped to an IP address. For each generated domain, a DNS A request is performed and according to the DNS response status, the domain name is considered as existing (status = `NOERROR`) or non-existing (status = `NXDOMAIN`). For more information about DNS and its operation, the reader must refer to [17–19].

The first step of the experiments aims at analyzing the existence and the type of generated domains. Figure 4 is an histogram depicting the run of five probing campaigns using a generation model trained on 10% of the *malicious dataset*, each of the five complete rectangle represents the number of existing domains generated. We can see that over one million unique domains probed, between 80,000 and 110,000 so around 10% are potentially reachable over the Internet. These existing domains are divided in three categories represented distinct filling pattern in the histograms.

The white one represents the number of wildcarded domains. Domain wildcarding is a technique that consists in associating an IP address to all possible subdomains of a domain by registering a domain name such as `*.yahoo.com`. As a result all DNS queries sent for a domain containing the suffix `yahoo.com` will be



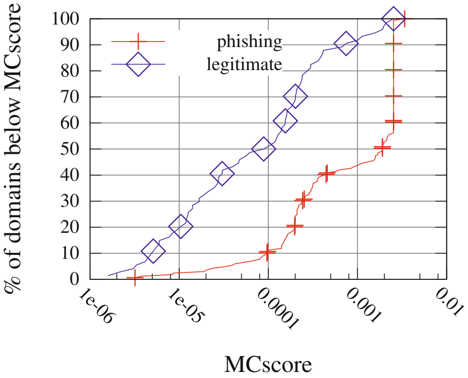


Fig. 4. Distribution of domains discovered regarding their types

Fig. 5. Cumulative ratio of domains regarding their score

answered with a NOERROR status DNS response containing always the same IP address. This technique is useful to tolerate Internet users typing mistakes, or misspelling of subdomain without any consequence. For instance, DNS requests for domain such as `wwe.yahoo.com`, `snt.yahoo.com` or `anyotherlabel.yahoo.com` will return the same IP address. However some TLDs such as `.ws`, `.tk`, `.us.com`, etc. apply also wildcarding. As a result these TLDs have been identified in order to discard all generated *two-level-domains* that contain one of them. We can see that these domains represent between 75% and 85% (from 60,000 to 90,000) of the domains discovered.

The remaining part is composed of two other categories. First some domains are registered but lead to websites of domain name resellers such as GoDaddy or Future Media Architect. A lot of meaningful domain names belong to this category, around 4,000 per campaign. Some examples of such domains are `freecolours.com` or `westeurope.com`. Regarding a probing campaign, the IP addresses obtained through DNS responses are stored and sorted by their number of occurrences. The IP addresses having more than fifty occurrences are manually checked to see if they are either related to real hosting or domain selling. Around fifty IP addresses and ranges have been identified as leading to domain name resellers. These domains are also discarded in our study, as they are not likely to be malicious domains. Finally the black part represents the domains that are unknown and have to be checked to confirm if they are related to phishing or not. As highlighted in Figure 4, the remaining potential malicious domains represent only between 15,000 and 20,000 domains out from one million of generated ones. This reduction is automated and allows discarding a lot of domain names, which will reduce the overhead of the checking process.

For domains of the unknown part that are known to be really legitimate or phishing, a score, $MCscore$, is calculated. This latter measures the similitude with the underlying training dataset, which have been used for building the model. Assuming a *two-level-domain* $w_1w_2 \ldots w_n.tld$ where w_i is the i^{th} word composing the name, w_i may have been generated using DISCO from an original word observed w'_i. $MCscore$ is computed as follows:

$$MCscore = distfirstword_{2,w'_1} \times sim_{w_1,w'_1} \times \prod_{i=1}^{n} distbiwords_{2,w_i,w'_{i+1}} \times sim_{w_{i+1},w'_{i+1}} \tag{10}$$

The first word probability is multiplied by each probability of crossed transition in the Markov chain. If some parts are found using DISCO, the similarity score given in equation (7) is used ($sim_{w_i,w_i} = 1$ else).

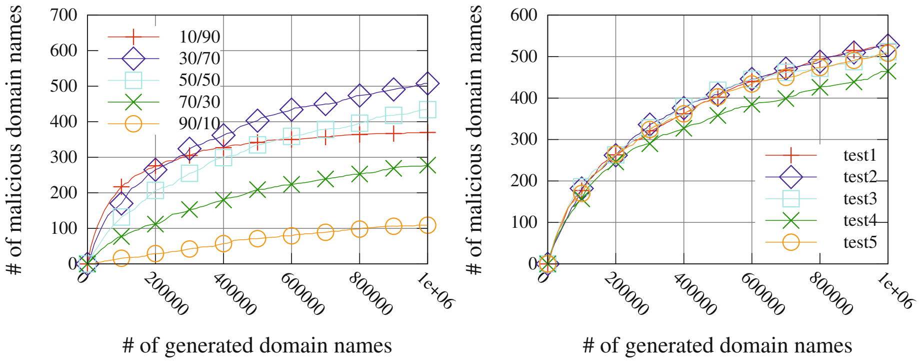
Figure 5 represents the cumulative sums of the ratio of domains (in %) that have a score lower than x for each kind of label. These curves show that globally phishing related domain names have a higher $MCscore$ than legitimate ones, around ten times higher. Even if a high number of domains are labeled as unknown and some of them are legitimate, it is easy to discard a lot of them in order to keep a set containing a main part of malicious domains. If we consider as malicious only the generated domains having a $MCscore$ higher than 0.001, then 93% of the legitimate domains will be discarded while 57% of the malicious domains will be kept. This technique can be used to avoid the use of a domain checking technique, as introduced in 2.1, or to reduce its workload.

4.3 Efficiency and Steadiness of Generation

This section assesses the variation of the efficiency of the malicious domain discovery regarding the ratio of domains in the *testing set* and in the *training set*. Five probing campaigns are performed with a ratio that varies from 10% training/90% testing (10/90) to 90% training/10% testing (90/10), the subsets are randomly made up. Figure 6(a) shows the number of malicious domains generated regarding the total number of probed names.

On one hand the best result is given by 30% training/70% testing with a total number of 508 phishing domains discovered. When the *testing set* size decreases, there are less domain names candidates that can be found, which implies that more domains are discovered with 30/70 than 90/10. On the other hand, the curve representing 10% training/90% testing grows faster, and after only 100,000 probes more than the half (217 domains) of the total number of phishing domains generated are found. Following the curve's trend, if more probes are performed, a reasonable assumption is that more malicious domain names can be discovered.

Figure 6(b) depicts the steadiness of the discovery results. Five probing campaigns are performed for the ratio that yields the best result: 30% training/70% testing. The training and testing sets are randomly made up and are different for each campaign. Observations are similar for every tests which lead to discover



(a) Variation of testing/training set ratio

(b) 5 probing campaigns with 30% training & 70% testing

Fig. 6. Number of domains discovered regarding the number of probes

around 500 phishing domains. Moreover, half of the discovered phishing domains are generated during the first 200,000 generations, highlighting the ability of our system to generate the most likely malicious domains in priority before being discarded for next probes.

4.4 Predictability

This experiment evaluates the time between the date when a malicious domain name can be generated using the generator and the date it is actually blacklisted. The training set is composed of the 10% oldest blacklisted domains and the remaining 90% belong to the *testing set*. The *testing set* represent 34 months of blacklisted domains and the *training set* 4 months. Figure 7 depicts the number of malicious domain names generated regarding the number of months they are actually blacklisted after their generation (m+x). A large quantity of generated malicious domains appears in first four months after their generation, 14 in the two following months and 23 more in the next two months. This shows that domain name composition follows fashion schemes because more malicious domains that are discovered appear just after the ones that are used to train the model. However, it is worth noting that such domains continue to be discovered in the present showing that even old datasets can be useful to generate relevant malicious domains

4.5 Strategy Evaluation

We have described in section 2 the two core building blocks for generating domain names: the Markov chain model and the semantic exploration module. The impact of each module is assessed with respect to four strategies:

- MC: the Markov chain model alone.
- MC + 5 DISCO: the Markov chain model and for each selected state of the Markov chain the five most related words, regarding DISCO, are tested.

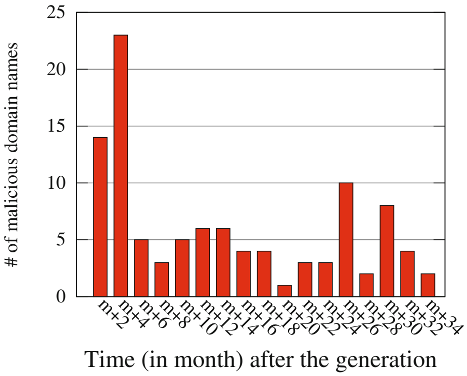


Fig. 7. Distribution of domain discovered regarding the time they are blacklisted

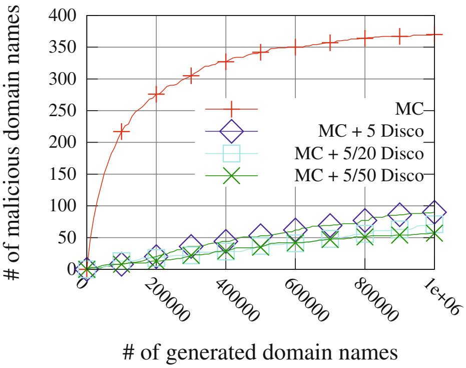


Fig. 8. Number of domain names discovered regarding the method

- MC + 5/20 DISCO: the Markov chain model and for each selected state of the Markov chain five words randomly picked from the twenty most related words, regarding Disco, are tested.
- MC + 5/50 DISCO: the Markov chain model and for each selected state of the Markov chain five words randomly picked from the fifty most related words, regarding Disco, are tested.

The objective of this assessment is to identify the best tradeoff between the success rates in discovering rogue domain names with respect to the computational effort.

Figure 8 shows the number of actual generated malicious domain names with respect to the number of probes performed over a probing campaign. The same training set is used to build the generation model for the four different probing campaigns. It clearly comes out that the Markov chain model alone yields the best results in term of number of malicious domain names discovered with a total of 370. However even if DISCO strategies are able to generate only between 57 and 90 malicious domain names over these campaigns depending on the technique, between 79% and 85% of these generated domains are unique, *i.e.* none of the other strategies are able to find them. If a global probing is targeted all the part of the generation module must be used in order to discover the maximum of phishing related domain names. However, the Markov chain model is sufficient to find out domains over a short period of time.

5 Related Work

Because of their essential role, anti-phishing and identification of malicious domain are increasingly popular and addressed in several previous works. Two major approaches exist: methods based on blacklists and heuristics-based methods.

Heuristic-based approaches rely on classification algorithms to identify whether a domain is malicious or not, based on features extracted from different sources. The leveraging of machine learning techniques to classify on the fly, domains as malicious or benign is widely used, with either batch methods such as SVM, Naive Bayes, Logistic Regression(like in [2–4]), or on-line classification algorithms such as Confidence Weighted (CW), Adaptive Regularization of Weight (AROW), Passive-Aggressive (PA) (see [5, 9, 15]).

Their differences are mainly related to the feature set. In [2–4], the building of classification models relies on passively gathered DNS queries to figure out predominantly malware domains involved in botnet communications. However for phishing detection purposes, it can be either host-based features (WHOIS info, IP prefix, AS number) in [15] or web page content-based features in [25].

The majority of features are however extracted directly from URLs. These can be for instance the type of protocol, hostname, TLD, domain length, length of URL, etc. In [5, 9, 11], the authors particularly focus on the tokens that compose a complete URL, which includes the domain as well as the path and the query. In these studies, the classification is based on the relative position of these tokens (domain or path level for instance) or the combination of these tokens (*token1.token2/token3/token4?token5=token6*). The conclusion is that tokens that occur in phishing URLs belong to a limited dictionary and tend to get reused in other URLs. Moreover, Garera *et al.*, in [8], are the first to use the occurrences of manually defined words (secure, banking, login, etc.) in URL as features. In [13], Le *et al.* use both batch and on-line classification techniques to show that lexical features extracted from URLs are sufficient to detect phishing domains. Even also based on lexical features, our work is different as we consider meaningful words that compose the same label of a domain name. Moreover, our work consists in a predictive and active discovery rather than classification of domain names observed on network traffic.

There have been other works taken advantage of URL based lexical analysis for different purposes. In [27], statistical measures are applied to alphanumeric characters distribution and bigrams distribution in URLs in order to detect algorithmically generated fluxing domains. The same technique is used in [6] to detect DNS tunnels and, in [26], Xie *et al.* generate signature for spamming URLs using Regular Expressions. URLs related to the same spam campaign are grouped for creating a signature based on regular expression.

This work is close to our approach but only lead to disclose domain names related to a specific spam campaign from which some domain names have been already observed. Our approach is more general and allows discovering new phishing domains that have no apparent relations with previous ones.

Blacklisting approaches consist in the partially manual construction of a list of malicious URLs that will be used by web browser or e-mail client in order to prevent the users to access them. Due to their short lifetime, the early identification of phishing websites is paramount, as a result several methods have been proposed to avoid reactive blacklisting and develop more proactive methods. In [10], Hao *et al.* analyze early DNS behavior of newly registered domains. It is demonstrated that they are characterized by DNS infrastructure pattern and DNS lookup patterns monitored as soon as they are registered, such as either a wide scattering of resource records across the IP address space in only few regions, or resource records that are often hosted in tainted autonomous systems. A very close approach is used in [7] to build proactive domain blacklists. Assuming a known malicious domain, zone information is mined to check if other domains are registered at the same time, on the same name server and on the same registrar. However, this approach cannot be widely applied because domain zone information is not always available and needs a prior knowledge.

Predictive blacklisting is also addressed in [28] where it's assumed that a host should be able to predict future attackers from the mining of its logs. Hence, a host can create its own customized blacklist well fitted to its own threats. The composition of this blacklist relies also on other machines' logs that are considered as similar. This similarity score is calculated using the number of common attackers between two victims and stored into a graph explored using a PageRank algorithm to estimate whether a domain is likely to conduct an attack against a particular victim or not. The similarity calculation is refined in[23].

Another proactive blacklisting approach to detect phishing domains is addressed through Phishnet in [20]. This work is the closest to ours, the idea is to discover new phishing URLs based on blacklist of existing phishing URLs. As in [26], URLs are clustered based on their shared common domain names, IP addresses or directory structures, then a regular expression is extracted from each cluster. The variable part of regular expression is exploited to generate new URLs instead of only compare existing URLs to extracted patterns like in [26]. Though this method is more proactive than the previous ones, it is only able to disclose URLs related to already blacklisted URLs that are likely to belong to the same phishing attack. Whereas these URLs are part of a very small pool of domains, our approach is capable to extend the knowledge about distinct phishing attacks. Therefore these approaches are quite complementary.

Finally we have already treated and proved the efficacy of algorithmic domain generation based on Markov chain in [24]. We apply this technique on bigrams in order to perform a discovery of all the subdomains (`www`, `mail`, `ftp`) of a given domain (`example.com`). We extend this approach in [16], based on an existing list of subdomains, we leverage semantic tools and incremental techniques to discover more subdomains.

6 Conclusion and Future Work

This paper introduces an efficient monitoring scheme for detecting phishing sites. The main idea consists in generating a list of potential domain names that might be used in the future by an attacker. This list can be checked on a daily basis to detect the apparition of a new phishing site. The list is generated using language models applied to known ground truth data. We have proposed a novel technique to generate domain names following a given pattern that can be learned from existing domain names. This domain generation leverages a Markov chain model and relevant lexical features extracted from a semantic splitter. Domain specific knowledge is added from semantic tools. The efficiency of this generation tool is tested on the real world datasets of phishing domain names. We proved that our method is able to generate hundreds of new domain names that are actually related to phishing and appear to be in use in the period following their generation. To the best of our knowledge, our approach is the only one to propose proactive generation and discovery of malicious domains, which is complementary to state of the art approaches that addressed proactive blacklisting of URLs.

In future works, the remaining part of the architecture, the domain checker, will be implemented shortly. Furthermore, the feedback from this checker will be used to adapt the Markov chain transition probability through reinforcement learning in order to strengthen the generation model. The code is available on request.

Acknowledgements. This work is partly funded by OUTSMART, a European FP7 project under the Future Internet Private Public Partnership programme. It is also supported by MOVE, a CORE project funded by FNR in Luxembourg. The authors would like to thank P. Bedaride for discussions and advice on natural language processing tools.

References

1. Anti-Phishing Working Group and others: Phishing Activity Trends Report - 1H2011. Anti-Phishing Working Group (2011)
2. Antonakakis, M., Perdisci, R., Dagon, D., Lee, W., Feamster, N.: Building a dynamic reputation system for dns. In: Proceedings of the 19th USENIX Conference on Security, USENIX Security 2010, p. 18. USENIX Association, Berkeley (2010)
3. Antonakakis, M., Perdisci, R., Lee, W., Vasiloglou II, N., Dagon, D.: Detecting malware domains at the upper dns hierarchy. In: Proceedings of the 20th USENIX Conference on Security, SEC 2011, p. 27. USENIX Association, Berkeley (2011)
4. Bilge, L., Kirda, E., Kruegel, C., Balduzz, M.: EXPOSURE: Finding Malicious Domains Using Passive DNS Analysis. In: NDSS 2011. Internet Society (February 2011)
5. Blum, A., Wardman, B., Solorio, T., Warner, G.: Lexical feature based phishing url detection using online learning. In: Proceedings of the 3rd ACM Workshop on Artificial Intelligence and Security, pp. 54–60. ACM (2010)
6. Born, K., Gustafson, D.: Detecting dns tunnels using character frequency analysis. Arxiv preprint arXiv:1004.4358 (2010)
7. Felegyhazi, M., Kreibich, C., Paxson, V.: On the potential of proactive domain blacklisting. In: Proceedings of the 3rd USENIX Conference on Large-Scale Exploits and Emergent Threats: Botnets, Spyware, Worms, and More, p. 6. USENIX Association (2010)
8. Garera, S., Provos, N., Chew, M., Rubin, A.D.: A framework for detection and measurement of phishing attacks. In: Proceedings of the 2007 ACM Workshop on Recurring Malcode, pp. 1–8. ACM (2007)
9. Gyawali, B., Solorio, T., Wardman, B., Warner, G., et al.: Evaluating a semisupervised approach to phishing url identification in a realistic scenario. In: Proceedings of the 8th Annual Collaboration, Electronic Messaging, Anti-Abuse and Spam Conference, pp. 176–183. ACM (2011)

10. Hao, S., Feamster, N., Pandrangi, R.: Monitoring the initial DNS behavior of malicious domains. In: Proceedings of the ACM SIGCOMM Internet Measurement Conference, IMC 2011, pp. 269–278. ACM, New York (2011)
11. Khonji, M., Iraqi, Y., Jones, A.: Lexical url analysis for discriminating phishing and legitimate websites. In: Proceedings of the 8th Annual Collaboration, Electronic messaging, Anti-Abuse and Spam Conference, pp. 109–115. ACM (2011)
12. Kolb, P.: DISCO: A Multilingual Database of Distributionally Similar Words. In: Storrer, A., Geyken, A., Siebert, A., Würzner, K.-M. (eds.) KONVENS 2008 – Ergänzungsband: Textressourcen und Lexikalisches Wissen, pp. 37–44 (2008)
13. Le, A., Markopoulou, A., Faloutsos, M.: Phishdef: Url names say it all. In: INFOCOM, 2011 Proceedings IEEE, pp. 191–195. IEEE (2011)
14. Ludl, C., Mcallister, S., Kirda, E., Kruegel, C.: On the Effectiveness of Techniques to Detect Phishing Sites. In: Hämmerli, B.M., Sommer, R. (eds.) DIMVA 2007. LNCS, vol. 4579, pp. 20–39. Springer, Heidelberg (2007)
15. Ma, J., Saul, L., Savage, S., Voelker, G.: Identifying suspicious urls: an application of large-scale online learning. In: Proceedings of the 26th Annual International Conference on Machine Learning, pp. 681–688. ACM (2009)
16. Marchal, S., François, J., Wagner, C., Engel, T.: Semantic Exploration of DNS. In: Bestak, R., Kencl, L., Li, L.E., Widmer, J., Yin, H. (eds.) NETWORKING 2012, Part I. LNCS, vol. 7289, pp. 370–384. Springer, Heidelberg (2012)
17. Mockapetris, P.: Rfc 1035: Domain names - implementation and specification
18. Mockapetris, P.: Rfc 1034: Domain names - concepts and facilities (1987)
19. Mockapetris, P., Dunlap, K.: Development of the domain name system. In: Proceedings of the 1988 ACM SIGCOMM, pp. 123–133. IEEE Computer Society, Stanford (1988)
20. Prakash, P., Kumar, M., Kompella, R., Gupta, M.: Phishnet: predictive blacklisting to detect phishing attacks. In: INFOCOM, 2010 Proceedings IEEE, pp. 1–5. IEEE (2010)
21. Rasmussen, R., Aaron, G.: Global phishing survey: trends and domain name use in 1h2011. Anti-Phishing Working Group (2011)
22. Segaran, T., Hammerbacher, J.: Beautiful Data: The Stories Behind Elegant Data Solutions, ch. 14. O'Reilly Media (2009)
23. Soldo, F., Le, A., Markopoulou, A.: Predictive blacklisting as an implicit recommendation system. In: INFOCOM, 2010 Proceedings IEEE, pp. 1–9. IEEE (2010)
24. Wagner, C., François, J., State, R., Engel, T., Dulaunoy, A., Wagener, G.: SDBF: Smart DNS Brute-Forcer. In: Proceedings of IEEE/IFIP Network Operations and Management Symposium - NOMS. IEEE Computer Society (2012)
25. Xiang, G., Hong, J.: A hybrid phish detection approach by identity discovery and keywords retrieval. In: Proceedings of the 18th International Conference on World Wide Web, pp. 571–580. ACM (2009)
26. Xie, Y., Yu, F., Achan, K., Panigrahy, R., Hulten, G., Osipkov, I.: Spamming botnets: signatures and characteristics. In: ACM SIGCOMM Computer Communication Review, vol. 38, pp. 171–182. ACM (2008)
27. Yadav, S., Reddy, A.K.K., Reddy, AL, Ranjan, S.: Detecting algorithmically generated malicious domain names. In: Proceedings of the 10th Annual Conference on Internet Measurement, pp. 48–61. ACM (2010)
28. Zhang, J., Porras, P., Ullrich, J.: Highly predictive blacklisting. In: Proceedings of the 17th Conference on Security Symposium, pp. 107–122. USENIX Association (2008)

Evaluating Electricity Theft Detectors in Smart Grid Networks

Daisuke Mashima[1] and Alvaro A. Cárdenas[2]

[1] Georgia Institute of Technology
mashima@cc.gatech.edu
[2] Fujitsu Laboratories of America
alvaro.cardenas-mora@us.fujitsu.com

Abstract. Electricity theft is estimated to cost billions of dollars per year in many countries. To reduce electricity theft, electric utilities are leveraging data collected by the new Advanced Metering Infrastructure (AMI) and using data analytics to identify abnormal consumption trends and possible fraud. In this paper, we propose the first threat model for the use of data analytics in detecting electricity theft, and a new metric that leverages this threat model in order to evaluate and compare anomaly detectors. We use real data from an AMI system to validate our approach.

1 Introduction

The smart grid refers to the modernization of the power grid infrastructure with new technologies, enabling a more intelligently networked automated system with the goal of improving efficiency, reliability, and security, while providing more transparency and choices to electricity consumers. One of the key technologies being deployed currently around the globe is the Advanced Metering Infrastructure (AMI).

AMI refers to the modernization of the electricity metering system by replacing old mechanical meters by *smart meters*. Smart meters are new embedded devices that provide two-way communications between utilities and consumers, thus eliminating the need to send personnel to read the meters on site, and providing a range of new capabilities, such as, the ability to monitor electricity consumption throughout the network with finer granularity, faster diagnosis of outage—with analog meters, utilities learned of outages primarily by consumer call complaints—automated power restoration, remote disconnect, and the ability to send information such as dynamic pricing or the source of electricity (renewable or not) to consumers, giving consumers more—and easier to access—information about their energy use.

Smart meters are, by necessity, billions of low-cost commodity devices, with an operational lifetime of several decades and operating in physically insecure locations [16]. Hardening these devices by adding hardware co-processors and tamper resilient memory might increase the price of smart meters by a few dollars, and because utilities have to deploy millions of devices, the reality of the

D. Balzarotti, S. Stolfo, and M. Cova (Eds.): RAID 2012, LNCS 7462, pp. 210–229, 2012.

market is that these additions are not considered cost-effective in practice, and are not even recommended as a priority [21].

Therefore, while some basic protective measures have been developed (tamper-evident seals, secure link communications), they are not enough to prevent successful attacks during the meter lifespan. In addition to vulnerabilities identified by security researchers [17,9]—some of them allowing rogue remote firmware updates [20]—hacked smart meters have been used to steal electricity, costing a single U.S. electric utility hundreds of millions of dollars annually, as reported by a cyber-intelligence bulletin issued by the FBI [14]. The FBI report warns that insiders and individuals with only a moderate level of computer knowledge are likely able to compromise and reprogram meters with low-cost tools and software readily available on the Internet. The FBI report also assesses with medium confidence that as smart grid use continues to spread throughout the country, this type of fraud will also spread because of the ease of intrusion and the economic benefit to both the hacker and the electric customer.

Detecting electricity theft has traditionally been addressed by physical checks of tamper-evident seals by field personnel and by using balance meters [10]. While valuable, these techniques alone are not enough. Tamper evident seals can be easily defeated [5] and balance meters can detect that some of the customers connected to it are misbehaving, but cannot identify exactly who they are. Despite the vulnerabilities of smart meters, the high-resolution data they collect is seen as a promising technology to improve electricity-theft detection. In general, utilities are gathering more data from many devices and they are leveraging *big data analytics* [15] to obtain better situational awareness of the health of their system. One of the key services offered by Meter Data Management (MDM) vendors for turning big data into actionable information is called *revenue assurance*, where data analytics software is used by the utility on the collected meter data to identify possible electricity theft situations and abnormal consumption trends [13]. Big data analytics is thus a new cost-effective way to complement the use of balance meters (which are still necessary to detect when electricity thieves connect directly to the power distribution lines instead of tampering with the meter) and physical personnel checking for tamper-evident seals.

In this paper we focus on the problem of data analytics in MDM systems for detecting electricity theft. While some MDM vendors are already offering this functionality, their methods and algorithms are not publicly available, so it is impossible to evaluate the effectiveness of these tests. In addition, the few papers available on the topic have limitations [18,19,11,6]: (1) They do not consider a threat model, and therefore, it is not clear how the detection algorithm will work against sophisticated attackers, (2) they have lower resolution data, and therefore they tend to focus on nonparametric statistics, instead of leveraging advanced signal processing algorithms, and (3) they assume a dataset of attack examples to test the accuracy of the classifiers, and therefore the evaluation will be biased depending on how easy it is to detect attacks available in the database, and the effectiveness of the classifier will be unknown to unseen attacks.

In this paper we make the following contributions: (1) We introduce an attacker model for anomaly detectors in MDM systems. Previous work never assumed an intelligent attacker and therefore might have easily been evaded by an advanced attacker. This threat model is particularly important in digital meters, as an attacker with access to a tampered meter can send an arbitrary fine-grained attack signal with a precision that was not previously available with mechanical attacks to meters (such as using powerful magnets to affect the metrology unit). (2) We introduce a new metric for evaluating the classification accuracy of anomaly detectors. This new metric takes into consideration some of the fundamental problems in anomaly detection when applied to security problems: (a) the fact that attack examples in a dataset might not be representative of future attacks (and thus a classifier trained with such attack data might not be able to detect new *smart* attacks), and (b) in many cases it is hard to get *attack* data for academic studies—this is particularly true for SCADA data and data from sensor and actuators in industrial or power grid systems—therefore we argue that we have to avoid training and evaluating classifiers with imbalanced and unrepresentative datasets. (3) Using real AMI data (6 months of 15 minute reading-interval for 108 consumers) provided by an utility, we evaluate the performance of electricity-theft detection algorithms, including a novel ARMA-GLR detector designed with the goal of capturing an attack invariant (reducing electricity bill) in the formal model of composite hypothesis testing.

2 Evaluation of Classifiers in Adversarial Environments

In this section we describe a new general way of evaluating classifiers in adversarial environments. Because this framework can be used for other problems, we introduce the model in a general classification setting. We focus on two topics: (1) **adversarial classification**, or *how to evaluate the effectiveness of a classifier when the attacker can create undetected attacks*, and (2) **adversarial learning**, or *how to prevent an attacker from providing false data to our learning algorithm.*

2.1 Adversarial Classification

In machine learning, classifiers are traditionally evaluated based on a testing dataset containing examples of the negative (normal) class and the positive (attack) class. However, in adversarial environments there are many practical situations where we cannot obtain examples of the attack class a priori. There are two main reasons for this: (1) by definition, we cannot obtain examples of zero-day attacks, and (2) using attack examples which are generated independently of the classifier implicitly assumes that the attacker is not adaptive and will not try to evade our detection mechanism.

In this paper we argue that instead of using a set of attack samples for evaluating classifiers, we need to find the worst possible attack for each classifier and evaluate the classifier by considering the costs of this worst-case attack.

Model and Assumptions: We model the problem of evaluating classifiers by generating worst-case attack patterns as follows:

1. A random process generates observations $x \in \mathcal{X}$. These observations are the realization of a random vector X with distribution P_0.
2. We assume x is only observed by a *sensor* (e.g., a smart meter), and the sensor sends y to a classifier. Thus while P_0 is known to the world, the specific sample x is only known to the sensor.
3. The sensor can be in one of two states (1) honest, or (2) compromised. If the sensor is honest, then $y = x$. If the sensor is dishonest, then $y = h(x)$, where $h : \mathcal{X} \rightarrow \mathcal{X}$ is a function such that the inferred probability distribution P_1 for Y satisfies a *Relation* (the attacker intent): $g(X)\ R\ g(Y)$ (e.g., $\mathbb{E}[Y] < \mathbb{E}[X]$ where $\mathbb{E}[X]$ is the expectation of the random variable X).
4. The classifier $f : \mathcal{X} \rightarrow \{n, p\}$ outputs a decision: A negative n for concluding that y is a sample of P_0 and a positive p to decide that y is a sample of P_1.

A Metric for Evaluating Classifiers in Adversarial Environments: In order to generate attacks we propose a cost function $C(x_i, y_i)$ to generate attack vectors y_i by modifying the original value x_i such that y_i is the attack that maximizes $C(x_i, y_i)$ while being undetected. In particular, we assume we are given:

1. A set $\mathcal{N} = \{x_1, \ldots, x_m\} \in \mathcal{X}^m$ where each x_i is assumed to be a sample from P_0. Note that $x_i \in \mathcal{X}$. A common example is $\mathcal{X} = \mathbb{R}^d$, i.e., each observation x_i is a vector of real values with dimension d. In a smart-metering application this can mean that x_i corresponds to the meter readings collected over a 24-hour period.
2. A value $\alpha \in [0, 1]$ representing an upper bound on the tolerable false alarm probability estimate in the set $\mathcal{N}$.
3. A cost function $C : \mathcal{X} \times \mathcal{X} \rightarrow \mathbb{R}$ representing the cost of a false negative.
4. A set of candidate classifiers $\mathcal{F} = \{f_0, \ldots, f_q\}$, where each classifier is parameterized by a threshold τ used to make a decision. If we want to make explicit the threshold used by a particular classifier we use the notation f_{i,τ_i}.

Calculating the Adversarial Classification Metric

1. $\forall f_{i,\tau_i} \in \mathcal{F}$ find the threshold that configures the classifier to allow a false alarm rate as close as possible to the upper bound α. If no such threshold exists, then discard the classifier (since it will always raise more false alarms than desired).

$$\tau_i* = \arg\max_{\tau_i \in \mathbb{R}} \Pr[f_{i,\tau_i}(X) = p | X \sim P_0] \tag{1}$$

$$\text{subject to:} \qquad \Pr[f_{i,\tau_i}(X) = p | X \sim P_0] \leq \alpha \tag{2}$$

This formal definition can be empirically estimated by the following equations:

$$\tau_i* = \arg\max_{\tau_i \in \mathbb{R}} \frac{|\{x \in \mathcal{N} : f_{i,\tau_i}(x) = p\}|}{|\mathcal{N}|} \tag{3}$$

$$\text{subject to: } \frac{|\{x \in \mathcal{N} : f_{i,\tau_i}(x) = p\}|}{|\mathcal{N}|} \leq \alpha \tag{4}$$

2. Among all classifiers that satisfy the false alarm constraint, find the worst possible *undetected* attacks for each of them. Let $y = h(x)$ denote the attack strategy based on observation x, then the optimal attack strategy requires an optimization over the functional space h:

$$\begin{aligned} C(f_i) = \quad & \max_h \mathbb{E}[C(X, h(X))] \\ \text{subject to: } & f_{i,\tau_i *}(h(x)) = n \text{ if } f_{i,\tau_i *}(x) = n \;\text{ and } h(x) = x \text{ if } f_{i,\tau_i *}(x) = p \end{aligned}$$

Notice that if the negative sample raises a false alarm then the attacker just forwards this value. While in practice an attacker might want to stay undetected at all times ($\forall x, f(h(x)) = n$), this would lower the attacker's gain (e.g., the amount of electricity the attacker can steal). We are therefore, considering the most conservative case where we allow the attacker to steal more without being detected (we count alarms generated by $h(x) = x$ as false alarms, and therefore allow the attacker to remain undetected with this aggressive attack). Given a dataset of negative examples, we can empirically estimate this functional optimization problem by the following equations:

$$\begin{aligned} C(f_i) = & \max_{[y_1,\ldots,y_m] \in \mathcal{X}^m} \sum_{x_i \in \mathcal{N}} C(y_i, x_i) \\ \text{Subject to: } & f_{i,\tau_i *}(y_i) = n \text{ if } f_{i,\tau_i *}(x_i) = n \quad \text{ and } y_i = x_i \text{ if } f_{i,\tau_i *}(y_i) = p \end{aligned}$$

3. The best classifier f_i* is the one with the minimum cost for the most costly undetected attack:

$$f_i* = \arg \min_{f_i \in \mathcal{F}} C(f_i) \tag{5}$$

2.2 Adversarial Learning

Another fundamental evaluation criteria should be the resilience and countermeasures deployed for adversarial learning. In general, the idea of learning some basic properties of a random process and then using them to detect anomalies sounds intuitive; however, in several cases of interest the random process may be non-stationary, and therefore we might need to retrain the classifier periodically to capture this **concept drift**.

Retraining a classifier opens the vulnerability that a smart attacker might force us to learn false *normal* models by poisoning the dataset. For our smart meter example, the attacker can send fake sensor measurement readings that lower average consumption but that do not raise alarms (when classified) so they can be used as part of the new training set. Over a period of time, our new estimated probability models will be different from the real process generating this data. We refer to these attacks as *contamination attacks* because they inject malicious data used to train the classifiers.

To evaluate the susceptibility of classifiers to contamination attacks, we study how these attacks can be generated and discuss a countermeasure in Section 4.2.

3 Electricity-Theft Detectors and Attacks

AMI systems collect and send electricity consumption data to the utility company several times per day. Electricity consumption data for a consumer is a time series $Y_1, Y_2, \ldots$, where Y_i is the electricity consumption of the utility customer in Watt-hours [Wh] from time period between measurement Y_{i-1} to measurement Y_i. The time between recorded measurements can change between different AMI deployments, as there is no standard defining the granularity of these measurements; however, a common measurement frequency is to take a recording every 15 minutes.

If an attacker obtains access to the meter, or is able to impersonate it, the attacker can send any arbitrary time series $\hat{Y}_1, \hat{Y}_2, \ldots$ back to the utility. Depending of the goal of the attacker, this false time-series can have different properties. In this paper we focus on attacks that electric utilities are most interested in detecting: electricity theft.

The goal of an attacker who steals electricity is to create a time series $\hat{Y}_1, \hat{Y}_2, \ldots$ that will lower its energy bill. Assuming the billing period consists of N measurements, the false time series should satisfy the following **attack invariant** for periods of time where electricity is billed at the same rate:

$$\sum_{i=1}^{N} \hat{Y}_i < \sum_{i=1}^{N} Y_i. \tag{6}$$

While one of the goals of the smart grid is to provide more flexible tariffs, these demand-response systems are still experimental and are currently deployed in trail phases. In addition, while the electric utility we are working with has a Time Of Use (TOU) program, all the traces we received were of their flat rate program (most of their customers do not take advantage of TOU). Therefore, while in future work we might need to consider other utility functions for the attacker (e.g., $\min_{Y_t} \sum Cost_t Y_t$) for the current work we focus on an attacker who only wants to minimize $\sum Y_i$. The main goal of this paper is to establish a sound evaluation methodology that can be extended for different cost-models.

In this section we propose several electricity-theft detectors to capture this attack invariant. While these detection algorithms have been studied extensively in the statistics and machine learning literature, this is the first work that studies how to apply them for electricity-theft detection.

To use the concept of *worst possible undetected attack* as outlined in Section 2, we define the following objective for the attacker: the attacker wants to send to the utility a time-series $\hat{Y}_i$ that will minimize its electricity bill: $\min_{\hat{Y}_i} \sum \hat{Y}_i$, subject to the constraint that a detector will not raise an alarm with $\hat{Y}_i$. We assume a very powerful attacker who has complete knowledge about each detection algorithm, the parameters that a detector uses, and has a complete historical data recorded on his own smart meter. This is indeed a very strong adversary model and might not represent the average risk of a utility; however, we want to build a lower-bound on the operational performance of the classifiers. The evaluation of classifiers using machine-learning and statistics in adversarial conditions has been historically performed under fairly optimistic assumptions [22], therefore we would like to motivate future research for evaluating attack-detection algorithms in worst-possible scenarios so their performance is not overstated.

3.1 Average Detector

One of the most straightforward ways to construct an electricity-theft detector is to use an average of the historical electricity consumption under the same conditions. This is in fact the way utilities used to detect metering abnormalities pre-AMI systems [12]: given a gross measurement (e.g., average or total power consumption for a month) Y, determine if Y is significantly lower than the historical average.

With AMI systems utilities can now obtain a fine grained view of the consumption pattern, where $\sum_{i=1}^{N} Y_i = Y$ and N is the number of measurements to compute the average. To use this electricity-theft indicator in AMI systems we can calculate $\bar{Y} = \frac{1}{N}\sum_{i=1}^{N} Y_i$ and then raise an alarm if $\bar{Y} < \tau$, where τ is a variable threshold.

Our *average detector* implementation calculates the detection threshold τ as follows. We consider a detector to handle an average of daily record.

1. Given a training dataset, say T days in the most recent past, we can compute T daily averages, D_i $(i = 1, \ldots, T)$.
2. $\tau = \min_i(D_i)$

Determining the threshold in this way, we do not encounter any false positives within the training dataset.

An attacker equipped with knowledge of τ and our implementation can mount an optimal attack by simply sending τ as $\hat{Y}_i$ all the day. Even though this attack results in an entirely "flat" electricity usage pattern, the average detector cannot detect this anomaly.

3.2 ARMA-GLR

One of the advantages of fine-grained electricity consumption patterns produced by the smart grid is that we can leverage sophisticated signal processing algorithms to capture more properties of normal behavior. We selected Auto-Regressive Moving Average (ARMA) models to represent a normal electricity consumption probability distribution p_0 because ARMA processes can approximate a wide range of time-series behavior using only a small number of parameters. ARMA is a parametric approach, and has the potential to perform better than nonparametric statistics if we can model p_0 and the optimal attack appropriately.

We train from our dataset an ARMA probability distribution p_0 (we used the *auto.arima* function in the *forecast* library in R [2] to fit ARMA models of our data by using the Yule-Walker equations and the Akaike information criteria) defined by the following equation:

$$Y_k = \sum_{i=1}^{p} A_i Y_{k-i} + \sum_{j=0}^{q} B_j V_{k-j} \tag{7}$$

where V is white noise distributed as $\mathcal{N}(0, \sigma)$.

An attacker will choose a probability distribution that changes the mean value of the sequence of observations. Therefore the attack probability distribution (p_γ) is defined by

$$Y_k = \sum_{i=1}^{p} A_i Y_{k-i} + \sum_{j=0}^{q} B_j (V_{k-j} - \gamma) \tag{8}$$

where $\gamma > 0$ is an unknown value and quantifies how small will the attacker select $\mathbb{E}_\gamma[Y]$ (the expectation of Y under probability distribution p_γ).

Given $Y_1, \ldots, Y_n$, we need to determine what is more likely: is this time series distributed according to p_0, or p_γ? To address this problem we prove the following theorem.

Theorem 1. *Among all changes that lower the mean of an ARMA stochastic processes, the optimal classification algorithm in the Neyman-Pearson sense is to raise an alarm if $\bar{\epsilon}^2$ is greater than a threshold τ: where $\bar{\epsilon} = \frac{1}{n}\sum_{i=1}^{n} \epsilon_i$, ϵ_k is the innovation process*

$$Y_k - \mathbb{E}_0[Y_k | Y_1, \ldots, Y_{k-1}] \text{ where } \mathbb{E}_0 \text{ is the expectation under probability } p_0, \tag{9}$$

and where we assume $\bar{\epsilon}$ is smaller than zero. (If $\bar{\epsilon} \geq 0$ then we decide that there is no attack.)

Proof. An optimal classification algorithm in the Neyman-Pearson sense refers to a classifier that given a fixed false alarm rate, will maximize the probability of detection. Given two probability distributions p_0 and p_γ defining the distribution of Y_i under each class, the optimal classifier in the Neyman-Pearson sense is a likelihood ratio test:

$$\ln \frac{p_\gamma(Y_1, \ldots, Y_n)}{p_0(Y_1, \ldots, Y_n)} = -\frac{\gamma}{\sigma} \sum_{i=1}^{n} (\epsilon_i + \frac{\gamma}{2}) \tag{10}$$

However, we do not know the value of γ as an attacker can choose any arbitrary value. Therefore we need to use the Generalized Likelihood Ratio (GLR) test to find the maximum likelihood estimate of γ given the test observations $Y_1, \ldots, Y_n$.

$$\ln \frac{\sup_{\gamma>0} p_\gamma(Y_1, \ldots, Y_n)}{p_0(Y_1, \ldots, Y_n)} = \max_{\gamma>0} \sum_{i=1}^{n} (-\frac{\epsilon_i \gamma}{\sigma} - \frac{\gamma^2}{2\sigma}) \tag{11}$$

To find the maximum (assuming the constraint $\gamma > 0$ is not active):

$$\frac{\partial f}{\partial \gamma} = \sum_{i=1}^{n}(-\frac{\epsilon_i}{\sigma} - \frac{\gamma}{\sigma}) = 0 \text{ which implies } \gamma = -\sum_{i=1}^{n}\frac{\epsilon_i}{n} = -\bar{\epsilon} \qquad (12)$$

as long as $\gamma > 0$ (i.e., the optimization constraint is not active).

Therefore, the final GLR test (if $\bar{\epsilon} < 0$) is:

$$\frac{\bar{\epsilon}}{\sigma}\sum_{i=1}^{n}(\epsilon_i - \frac{\bar{\epsilon}}{2}) = \frac{\bar{\epsilon}^2}{\sigma}(n - \frac{1}{2}) \qquad (13)$$

and since $\frac{1}{\sigma}(n - \frac{1}{2})$ is constant for the test, we obtain our final result.

By using one-step-ahead forecast, we calculate the innovation process ϵ_i. The threshold τ is determined based on the maximum $\bar{\epsilon}^2$ observed in the training dataset. The optimal attack strategy is as follows:

1. Calculate $E = \sqrt{\tau}$
2. Send $\hat{Y}_i = \mathbb{E}_0[Y_i|\hat{Y}_1, \ldots, \hat{Y}_{i-1}] - E$

where $\mathbb{E}_0[Y_i|\hat{Y}_1, \ldots, \hat{Y}_{i-1}]$ is the predicted ith value based on the observed measurements (including crafted ones) by the ARMA model.

3.3 Nonparametric Statistics

A concern regarding the ARMA-GLR detector is that it is only guaranteed to be optimal if ARMA processes can be used to model accurately normal electricity consumption behavior and attack patterns. To address these concerns we evaluate two more algorithms: nonparametric statistics (in this section) and unsupervised learning (in the following section).

Nonparametric statistics are robust to modeling errors: they have better classification accuracy when our model assumptions for the time-series is not accurate enough. This is a particularly important property for security problems, as we generally do not have good knowledge about the probability distribution properties of attacks.

A number of nonparametric algorithms have been designed to detect changes in the mean of a random processes. In this work we consider EWMA (Exponentially-weighted Moving Average) control chart [1] and Non-parametric CUSUM [8]. Because of space constraints and the fact that nonparametric test did not perform well in our experimental results, we omit the implementation details in this section and just give a brief overview of each detector and our attack.

A detector based on EWMA chart can be defined as $EWMA_i = \lambda Y_i + (1 - \lambda)EWMA_{i-1}$ where λ is a weighting factor and $0 < \lambda \leq 1$ and Y_i is one of the time series measurements (i.e. meter readings). An alarm is raised if $EWMA_i < \tau$, where τ is a configurable parameter. An attacker with knowledge of τ can create an attack as follows: While $EWMA_{i-1} > \tau$, send $\hat{Y}_i =$

$MAX(0, \frac{\tau-(1-\lambda)EWMA_{i-1}}{\lambda})$. When $EWMA_{i-1} = \tau$, send $\hat{Y}_i = \tau$. The idea here is that, before the EWMA statistic hits the threshold, an attacker attempts to reduce the meter-reading value as much as possible, and once it reaches τ, the attacker sends τ.

On the other hand, the Non-parametric CUSUM statistic for detecting a change in the mean of a random process is defined by $S_i = MAX(0, S_{i-1} + (\mu - Y_i - b))$ $(i = 1, \ldots, N)$, where μ is the expected value of the time-series, and b is a "slack" constant defined so that $\mathbb{E}[|\mu - Y_i| - b] < 0$ under normal operation. An alarm is raised if $S_i > \tau$. Our attack against this CUSUM-based detector is as follows: Calculate $M = \frac{\tau+Nb}{N}$ and send $\hat{Y}_i = \mu - M$. Note that this attack can take advantage of the total margin calculated as $\tau + Nb$.

3.4 Unsupervised Learning

One of the most successful algorithms for anomaly detection is the Local Outlier Factor (LOF) [7]. In our experiments we used *RapidMiner* [3] to calculate LOF scores. A *LOF detector* is implemented as follows:

1. Create a vector containing all measurements of a day to be tested in order, $V_{test} = \{Y_1, \ldots, Y_N\}$ where N is the number of measurements per day.
2. For all days in a training dataset, create vectors in the same way, $V_i = \{X_{i1}, \ldots X_{iN}\}$ $(i = 1, \ldots, T)$.
3. Create a set containing V_{test} and all V_is, and apply LOF to this set.
4. If $LOF_{test} < \tau$ where LOF_{test} is a score corresponding to V_{test}, conclude V_{test} is normal and exit.
5. If $\bar{Y}(= \frac{1}{N}\sum_{i=1}^{N} Y_i) < \frac{1}{NT}\sum_{i=1}^{T}\sum_{j=1}^{N} X_{ij}$, raise an alarm.

Because a high LOF score just implies that the corresponding data point is considered an outlier, we can not immediately conclude that high LOF score is a potential energy theft. In order to focus on detecting energy theft we only consider outliers with lower than average energy consumption.

While we are not able to prove that the following attack against our LOF detector is optimal because of the complexity of LOF, in the experimental section we show how our undetected attack patterns for LOF were better than the optimal attacks against other algorithms.

1. Among daily records in the training dataset whose LOF scores are less than τ, pick the one with the minimum daily sum, which we denote $\{Y_1^*, \ldots, Y_N^*\}$.
2. Find the maximum constant B such that $\{\hat{Y}_1, \ldots, \hat{Y}_N\}$, where $\hat{Y}_i = Y_i^* - B$, does not raise an alarm.
3. Send $\hat{Y}_i$.

4 Experimental Results

We use real (anonymized) meter-reading data measured by an electric utility company during six months. The meter readings consisted of 108 customers with a mix of residential and commercial consumers. The meter readings were recorded every 15 minutes. Because our dataset contains measurements that were sent immediately after installation, we assume the meter readings during this period are legitimate.

4.1 Adversarial Evaluation: Cost of Undetected Attacks

To complete the evaluation proposed in Section 2, we now define the cost function C as follows:

$$C(Y, \hat{Y}) = MAX(\sum_{i=1}^{N} Y_i - \hat{Y}_i, 0)$$

where $Y = \{Y_1, \ldots, Y_N\}$ is the actual electricity usage and $\hat{Y} = \{\hat{Y}_1, \ldots, \hat{Y}_N\}$ is the fake meter reading crafted by an attacker.

Note that, if the actual usage is very small, the term $\sum_{i=1}^{N} Y_i - \hat{Y}_i$ can become negative, which means that an attacker will pay more money. We assume that a sophisticated attacker can turn the attack off and let real readings go unmodified when the actual electricity consumption is expected to be lower than the crafted meter readings. Under this strategy, the cost is always positive or equal to 0.

There are a number of ways to configure an electricity-theft detector. Ideally we would like to train anomaly detections with seasonal information, but given that our data only covers half a year, experiments in this section focus on a setting where electricity theft detectors are re-trained daily based on the last T-days data.

The experiments are conducted as follows. For each customer,

1. Set $i = 0$
2. Pick records for 4 weeks starting at the ith day as a training dataset (i.e. $T = 28$).
3. By using this training data set, compute parameters, including τ.
4. Pick a record of a day just after the training dataset as testing data.
5. Test the data under the detection model trained to evaluate false positive rate. If the result is negative (i.e. normal), attacks are mounted and the cost of the undetected attack is calculated.
6. Increment i and go back to Step 2.

Given the limited set of data we had, finding the optimal training length is outside the scope of this work. We chose a 4-week sliding window because we saw on preliminary results that it gave us a good trade-off between average loss and false alarms. As we obtain more data, we plan to consider in future work year-long datasets so we can fit seasonal models into our experiments and analyze in-depth the optimal training length.

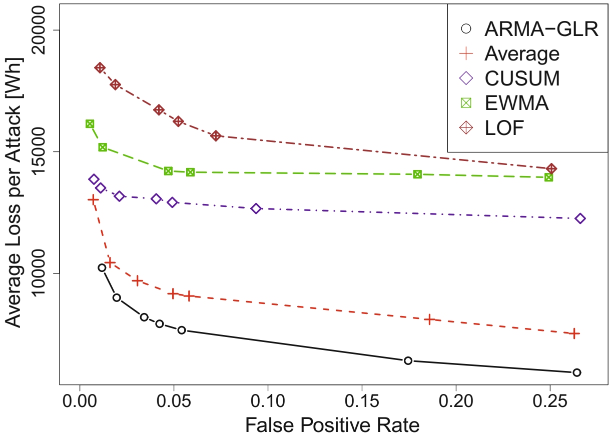


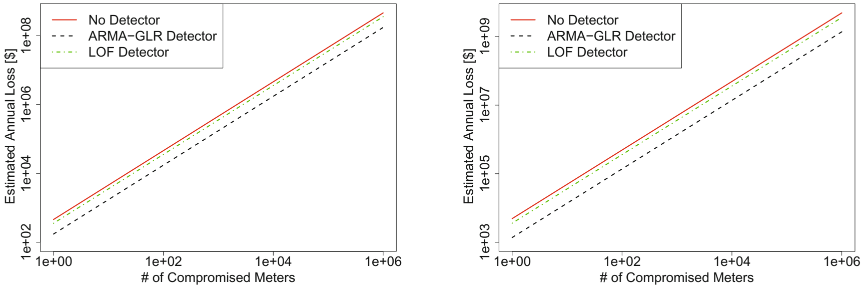
Fig. 1. Trade-off between false positive rate (the probability of false alarm) and average cost per attack

For each detector, we conducted 2,808 tests using the dataset of 108 customers, and for the cases that were claimed negative, we mounted attacks. The results are summarized in the trade-off curves in Fig. 1. The average cost per attack is calculated by dividing the total cost by the number of attacks performed.

As can be seen from the figure, the ARMA-GLR detector worked well. The average detector also is effective, but when the false positive rate is around 5%, its cost is higher than ARMA-GLR by approximately 1 KWh. It was somewhat surprising that the average detector outperformed the two online detectors: CUSUM and EWMA. One of the problems of these detectors is that they are designed to detect changes in a random process as quick as possible, and while this might have advantages for quick detection, it forced us to set very high thresholds τ to prevent false alarms in the 4-week-long training dataset. Detectors like ARMA-GLR and the average detectors on the other hand, smooth out sudden changes and are not susceptible to short-term changes of the random process. The cost of the LOF detector is the largest for all false positive rates evaluated.

Monetary Loss Caused by Undetected Electricity Theft. While assigning monetary losses to computer attacks is a difficult problem, one of the advantages of the dataset we have is that our data is directly related to the source of revenue of the utility company, and thus, it is easier to quantify the costs of potential attacks.

Using the electricity consumption rate charged by the utility company during the period of time we have available (while the utility company offers time-of-use prices, the tested customers belong to the flat rate program) we calculated that the (lower-bound) average revenue per-customer per-day is $1.256 dollars.



(a) Average customer

(b) Customer with largest consumption

Fig. 2. Estimated annual loss over # of compromised meters with 5% false positive rate. Notice that the x and y axes are in log-scales.

From the experimental results, we picked one result whose false positive rate in the testing dataset is nearly 5% for each detector (we cannot achieve exact 5% in all detectors), and then, we calculated the average monetary loss for optimal attacks per-customer per-day. The result is summarized in Table 1. 5% false positive rate may seem too high, but utilities do not have investigate them equally. Instead, they could choose to focus on large-consumption customers to protect major portion of their revenue.

Table 1 shows that (at 4.2% false alarm rate), ARMA-GLR can protect 62% of the revenue of the utility against optimal attacks, while the remaining detectors fair much worse, most of them even protecting less than 50% of the revenue at higher false alarm rates.

While in practice detecting electricity theft is a much more complex problem (as mentioned in the introduction it involves the use of balance meters and personnel inspections), and the anomaly detection tests considered in this paper should only be considered as *indicators* of theft, and not complete proof of theft, we believe these numbers are helpful when utility companies create a business case for investments in security and revenue protection. For example, we can study the average losses as the number of compromised meters increases (Fig. 2(a)). In this example we notice that the losses reported in studies about electricity theft [14,4] would require about 10,000 randomly compromised meters. However, if we look at the losses caused by the top electricity consumers (commercial meters) (Fig. 2(b)), the same amount of losses can be achieved by about 100 compromised meters (or close to 10,000 compromised meters if we use ARMA-GLR detectors). While prices of electricity vary globally, we can infer that to achieve the losses previously reported, a large portion of hacked meters must correspond to large commercial consumers.

Table 1. Monetary loss caused by undetected electricity theft (5% false positive rate)

Detector	FP Rate	Average Loss	Revenue Lost
Average	0.0495	$0.55	43%
EWMA	0.0470	$0.852	68%
CUSUM	0.0491	$0.775	62%
LOF	0.0524	$0.975	77%
ARMA-GLR	0.0423	$0.475	38%

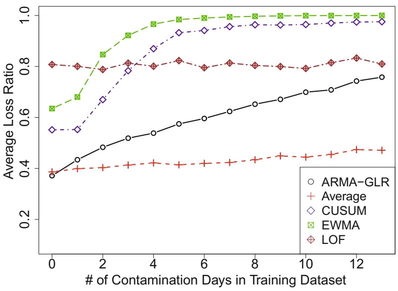


Fig. 3. Average loss ratio under contamination attack

4.2 Adversarial Learning: Detecting Contaminated Datasets

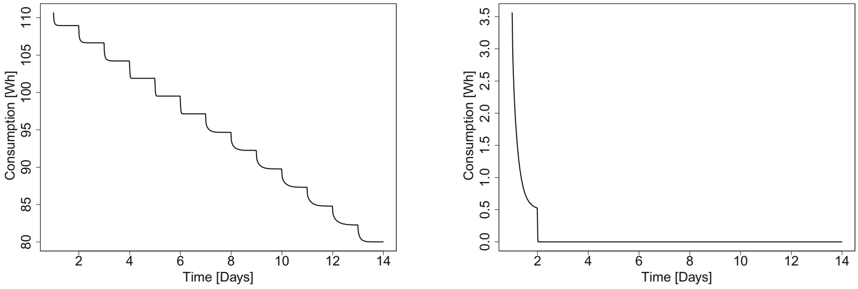
To evaluate the impact of contamination attacks discussed in Section 2.2, we show experiments using the same configuration as the ones used in the previous section. In this experiment, the optimal undetected attack is fed back into future training datasets. Namely, the training dataset of the second day includes an attack generated in the previous day, the training dataset of the third day contains two attack patterns generated for the first and the second day, and so forth.

We ran experiments for three disjoint time periods and calculated their average. The results are shown in Fig. 3. As can be seen from the plot, we can see increasing trends for all detectors except the LOF detector, which implies that LOF is more resilient to contamination attacks. In addition, the impact on the ARMA-GLR detector is much more significant than for the average detector. An intuitive explanation for this result is that ARMA models capture trends (unlike the average detector) therefore if we continue training ARMA models with a trend towards lower electricity consumption provided by an attacker, then the ARMA-GLR test will assume that future downward trends are normal.

Possible Countermeasures. A typical contamination attack pattern for the ARMA-GLR detector has the shape like the one shown in Fig. 4(a), in which we can see "roughly" a linear decreasing trend. A similar trend can be found in the case of other detectors. A straightforward way to identify such a pattern is fitting a linear model for the entire (or part of) a training dataset. We can expect that the resulting model would have negative slope significantly larger than other non-hacked customers. We applied linear regression for the contamination attack pattern of each customer. We also did the same for non-hacked meters for comparison. The results are summarized in Fig. 5(a). Though all of the attack patterns have negative slope, Fig. 5(a) shows this alone is not discriminative enough. Fortunately, we can find a clear difference in determination coefficients

(R^2) shown in Fig. 5(b)—determination coefficients are a measure of how well can a regression predict future values. High R^2, say $R^2 > 0.6$, with negative slope effectively indicates the existence of attacks. We manually investigated the attack patterns with low R^2 (those lower than 0.6) and found that all of them hit zero in the middle. For instance, the attack pattern shown in Fig. 4(b) gets to zero very quickly and remains at zero afterwards. Consecutive zeros is an indication of an anomaly and many utilities flag these events already, so the only attacks that will not be detected by the determination coefficients will be discovered by traditional rules.

The approach using linear regression also worked for other detectors since optimal attacks against them result in the similar, monotonically decreasing trends. While a motivated attacker can try to contaminate the training dataset at a slower pace so it is not detected, this will severely increase its effort and lower the effectiveness of its attacks.



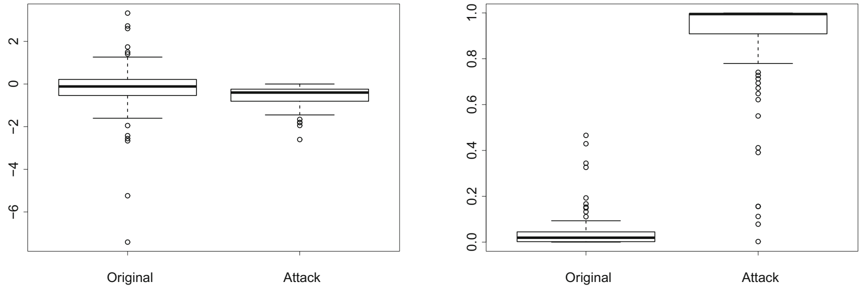
(a) Contamination attack with high R^2 (b) Contamination attack with low R^2

Fig. 4. Attack patterns under 14-day contamination attack experiment

5 Discussion

5.1 Cross-Correlation among Customers to Improve Detectors

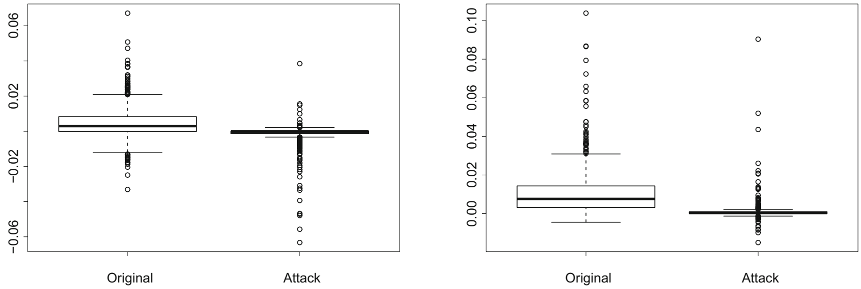
One possible way to identify attacks is to use cross-correlation among customers in nearby areas, assuming that honest customers exhibit similar trends while malicious customers have trends different from theirs. To evaluate this strategy, assuming that all 108 customers in the dataset are in the same neighborhood, we picked 7 daily consumption patterns from each of 108 customers and calculate cross covariance with the remaining 107 consumption patterns of the same day. Then, the average and quantile of these 107 cross covariances is calculated. Similarly, we calculated cross covariance between an attack pattern against the ARMA-GLR detector (Section 3.2) and original consumption patterns of the other 107 customers.



(a) Distribution of slopes of fitted linear Models

(b) Distribution of determination Coefficients of fitted linear models

Fig. 5. Distribution of slopes and determination coefficients of contamination attack patterns under linear model



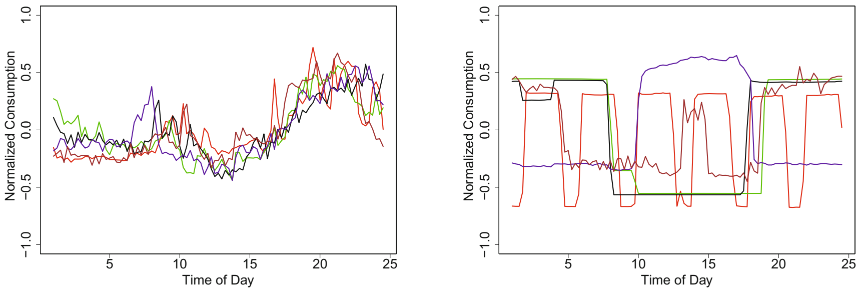
(a) Average

(b) 75 Percentile

Fig. 6. Distribution of average and 75 percentile of cross covariances

While we did not see significant difference in terms of 25 percentile and median of the 107 cross covariance values, their average and 75 percentile could be useful. Fig. 6 implies that a crafted attack pattern tends to exhibit a trend different from many of other customers' consumption patterns. Even though this alone can not be considered as definitive indication of attack, we could use it as an additional factor for electricity theft detection leveraging alarm fusion technologies.

In addition to cross-correlation, we can use outlier detection algorithms such as LOF [7], to identify outliers, exhibiting different trends in their electricity consumption patterns when compared to other similar consumers. In this direction, we have conducted some preliminary analyses. We smoothed daily electricity consumption patterns of a certain day in our dataset by using a low-pass filter. Then we normalized them since our focus here is anomaly in terms of shape and trends, not necessarily high or low consumption anomalies. Fig. 7 shows some samples of consumption patterns with top-5 (greater than 2.4) and low (less than 1.0) LOF scores. While inliers with low LOF scores are categorized into a couple of "typical" usage patterns, like the one shown in Fig. 7(a), we can identify unique patterns, including "artificially-looking" ones (Fig. 7(b)). We will continue this area of research in future work.



(a) Typical patterns identified by LOF

(b) Outliers identified by LOF

Fig. 7. LOF can Find Unusual Activity Patterns

5.2 Use of Auto-correlation of Residuals in ARMA-GLR Detector

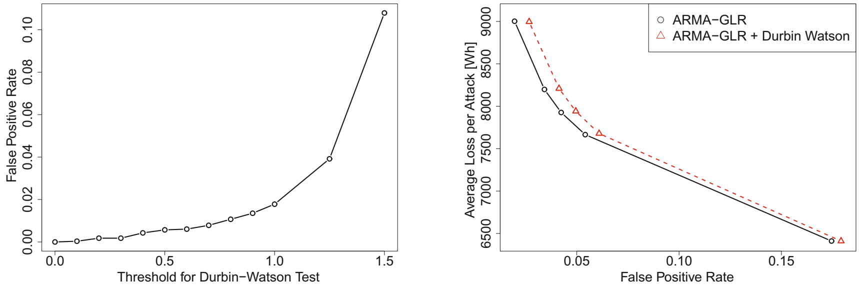
Based on the definition of attack strategy we tested in Section 3.2, we can expect that the sequence of residuals of generated attack patters have high auto-correlation, which can be a possible indication of attack. We have also explored this direction.

One of the possible metrics to quantify such auto-correlation is the Durbin-Watson statistic, defined as $d = \frac{\sum_{i=2}^{N}(e_i - e_{i-1})^2}{\sum_{i=1}^{N} e_i^2}$, where e_i denotes the ith residual and N is the number of measurements in the series. In general, we can infer that there exists auto-correlation when $d < 1.0$. Following this idea, we added the test of auto-correlation in residuals for the ARMA-GLR detector. Namely, for time-series patterns that passed the GLR test, we apply the test based on Durbin-Watson statistics. Using this approach we found that by setting the threshold for d around 1.0, it can detect all of the attacks mounted against ARMA-GLR as discussed in 3.2. The empirical relation between threshold values and the false positive rate, where false positives are counted only based on Durbin-Watson test (i.e. regardless of the result of ARMA-GLR test), is shown in Fig. 8(a).

Although we found that the use of Durbin-Watson statistics is effective to detect attacks against the ARMA-GLR detector, unfortunately it is not difficult to create attacks to defeat this other measure. For instance, a slightly modified attack strategy shown below would give attackers almost the same gain as the one he could do in case of the ARMA-GLR detector. When τ is the threshold used for the ARMA-GLR test,

1. Calculate $E = \sqrt{\tau}$
2. When i is an even number, send $\hat{Y}_i = \mathbb{E}_0[Y_i|\hat{Y}_1, \ldots, \hat{Y}_i] - 2E$. Otherwise, just send $\hat{Y}_i = \mathbb{E}_0[Y_i|\hat{Y}_1, \ldots, \hat{Y}_i]$.

This attack generates a sequence of residuals where 0 and $2E$ appear alternatively and results in having d approximately 2.0, which implies that attack can not be detected based on the threshold that is usually set around 1.0, while the total gain of an attacker is almost equal. As can be seen in Fig. 8(b), the trade-off curves are very similar. The weakness of the Durbin-Watson statistic is that it only considers first-order auto-correlation, so using higher-order correlation, such as Breusch-Godfrey Test or Ljung-Box Test, would make attacks harder. We will continue exploring ways to improve our detectors against sophisticated attackers in future work.



(a) False positive rates for Durbin-Watson statitics.

(b) Trade-off curves of ARMA-GLR and ARMA-GLR + Durbin Watson.

Fig. 8. Plots related to Durbin-Watson tests

5.3 Energy Efficiency

One of the goals of the smart grid is to give incentives for users to reduce their electricity consumption. In some cases (such as the installation of solar panels), the electric utility will know the consumer has installed these systems because the utility has to send personnel to approve the installation and allow them to sell electricity back to the grid. However, in some other cases, the incorporation of other green-energy technology might be unknown to the utility. In this case any anomaly detection algorithm will raise a false alarm. The best we can do is complement anomaly detection mechanisms with other information (e.g., balance meters) and in the case of false alarms, retrain new models with the new equipment in place. These changes are part of the non-stationarity of the random process we considered in this work.

6 Conclusions

In this paper we introduced the first rigorous study of electricity-theft detection from a computer-security perspective. While previous work has introduced other methods for electricity-theft detection, we argue that the incorporation of a new adversarial classification metric, and new mechanisms that consider adversarial learning are fundamental contributions to this growing area.

While all of the results in this paper consider pessimistic scenarios (the most-powerful attacker), we anticipate that these algorithms will perform much better under average cases where the attacker does not know the algorithms or time-intervals we use for anomaly detection and where it may not be able to compute optimal attack strategies. In addition, it is important to point out that the proposed anomaly detectors will only output *indicators* of an attack: a utility company will not only look at time-series anomalies as sources of attacks, but also at balance meters, smart meter tampering alarms, and might send personnel for periodic field monitoring reports. Combining all this information will give the utility good situational awareness of their network and accurate electricity-theft reports.

We plan to continue extending our work in multiple directions. For instance, optimal attacks are often artificial: e.g., the attacks against our average detector are constant values, therefore, adding additional mechanism that take advantage of the “shape” of the signal would be effective. We also plan to study more in-depth cross-correlation among nearby customers as an indicator of anomalies. Another approach to design classifiers resilient to attackers include the addition of randomness so the attacker cannot know at any time the state of the classifier. One example can be to leverage randomness in the use of training data, so an attacker would not know the exact configuration of the classifier. Finally, as we obtain datasets containing longer-periods of time, we plan to leverage accurate seasonal models and correlation with other factors, such as weather and temperature.

Acknowledgements. We would like to thank the reviewers and our shepherd, Guillaume Hiet, for insightful comments to improve this manuscript.

References

1. EWMA Control Charts, http://itl.nist.gov/div898/handbook/pmc/section3/pmc324.html
2. forecast package for R, http://robjhyndman.com/software/forecast/
3. RapidMiner, http://rapid-i.com/
4. Antmann, P.: Reducing technical and non-technical losses in the power sector. Technical report, World Bank (July 2009)
5. Appel, A.: Security seals on voting machines: A case study. ACM Transactions on Information and Systems Security 14, 1–29 (2011)
6. Bandim, C., Alves Jr., J., Pinto Jr., A., Souza, F., Loureiro, M., Magalhaes, C., Galvez-Durand, F.: Identification of energy theft and tampered meters using a central observer meter: a mathematical approach. In: 2003 IEEE PES Transmission and Distribution Conference and Exposition, vol. 1, pp. 163–168. IEEE (2003)

7. Breunig, M., Kriegel, H.-P., Ng, R.T., Sander, J.: Lof: Identifying density-based local outliers. In: Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data, pp. 93–104. ACM (2000)
8. Brodsky, B., Darkhovsky, B.: Non-Parametric Methods in Change-Point Problems. Kluwer Academic Publishers (1993)
9. Davis, M.: Smartgrid device security. adventures in a new medium (July 2009), http://www.blackhat.com/presentations/bh-usa-09/MDAVIS/BHUSA09-Davis-AMI-SLIDES.pdf
10. De Buda, E.: System for accurately detecting electricity theft. US Patent Application 12/351978 (January 2010)
11. Depuru, S., Wang, L., Devabhaktuni, V.: Support vector machine based data classification for detection of electricity theft. In: Power Systems Conference and Exposition (PSCE), 2011 IEEE/PES, pp. 1–8 (March 2011)
12. ECI Telecom. Fighting Electricity Theft with Advanced Metering Infrastructure (March 2011)
13. Geschickter, C.: The Emergence of Meter Data Management (MDM): A Smart Grid Information Strategy Report. GTM Research (2010)
14. Krebs, B.: FBI: smart meter hacks likely to spread (April 2012), http://krebsonsecurity.com/2012/04/fbi-smart-meter-hacks-likely-to-spread/
15. Lesser, A.: When big IT goes after big data on the smart grid (March 2012), http://gigaom.com/cleantech/when-big-it-goes-after-big-data-on-the-smart-grid-2/
16. McLaughlin, S., Podkuiko, D., McDaniel, P.: Energy Theft in the Advanced Metering Infrastructure. In: Rome, E., Bloomfield, R. (eds.) CRITIS 2009. LNCS, vol. 6027, pp. 176–187. Springer, Heidelberg (2010)
17. McLaughlin, S., Podkuiko, D., Miadzvezhanka, S., Delozier, A., McDaniel, P.: Multi-vendor penetration testing in the advanced metering infrastructure. In: Proceedings of the Annual Computer Security Applications Conference (ACSAC) (December 2010)
18. Nagi, J., Yap, K.S., Tiong, S.K., Ahmed, S.K., Mohamad, M.: Nontechnical loss detection for metered customers in power utility using support vector machines. IEEE Transactions on Power Delivery Systems 25(2), 1162–1171 (2010)
19. Nizar, A., Dong, Z.: Identification and detection of electricity customer behaviour irregularities. In: Power Systems Conference and Exposition (PSCE), pp. 1–10 (March 2009)
20. Peterson, D.: AppSecDC in review: Real-world backdoors on industrial devices (April 2012), http://www.digitalbond.com/2012/04/11/appsecdc-in-review/
21. Smart Grid Interoperability Panel, editor. NISTIR 7628. Guidelines for Smart Grid Cyber Security. NIST (August 2010)
22. Sommer, R., Paxson, V.: Outside the closed world: On using machine learning for network intrusion detection. In: IEEE Symposium on Security and Privacy (2010)

PoisonAmplifier: A Guided Approach of Discovering Compromised Websites through Reversing Search Poisoning Attacks

Jialong Zhang, Chao Yang, Zhaoyan Xu, and Guofei Gu

SUCCESS Lab, Texas A&M University
{jialong,yangchao,z0x0427,guofei}@cse.tamu.edu

Abstract. Through injecting dynamic script codes into compromised websites, attackers have widely launched search poisoning attacks to achieve their malicious goals, such as spreading spam or scams, distributing malware and launching drive-by download attacks. While most current related work focuses on measuring or detecting specific search poisoning attacks in the crawled dataset, it is also meaningful to design an effective approach to find more compromised websites on the Internet that have been utilized by attackers to launch search poisoning attacks, because those compromised websites essentially become an important component in the search poisoning attack chain.

In this paper, we present an active and efficient approach, named PoisonAmplifier, to find compromised websites through tracking down search poisoning attacks. Particularly, starting from a small seed set of known compromised websites that are utilized to launch search poisoning attacks, PoisonAmplifier can recursively find more compromised websites by analyzing poisoned webpages' special terms and links, and exploring compromised web sites' vulnerabilities. Through our 1 month evaluation, PoisonAmplifier can quickly collect around 75K unique compromised websites by starting from 252 verified compromised websites within first 7 days and continue to find 827 new compromised websites on a daily basis thereafter.

1 Introduction

Search Engine Optimization (SEO) manipulation, also known as "black hat" SEO, has been widely used by spammers, scammers and other types of attackers to make their spam/malicious websites come up in top search results of popular search engines. Search poisoning attacks, as one particular type of "black hat" SEO, inject malicious scripts into compromised web sites and mislead victims to malicious websites by taking advantages of users' trust on search results from popular search engines. By launching search poisoning attacks, attackers can achieve their malicious goals such as spreading spam, distributing malware (e.g., fake AntiVirus tools), and selling illicit pharmacy [14]. For example, in April 2011, many search terms (e.g., those related to the royal wedding between Britain Prince William and Catherine Middleton) are poisoned with Fake

D. Balzarotti, S. Stolfo, and M. Cova (Eds.): RAID 2012, LNCS 7462, pp. 230–253, 2012.

AntiVirus links [15]. These links mislead victims to install fake Security Shield AntiVirus software. In 2011, one research group from Carnegie Mellon University also reported substantial manipulation of search results to promote unauthorized pharmacies by attackers through launching search poisoning attacks [3].

Essentially, search poisoning attacks compromise benign websites by injecting malicious scripts either into existing benign webpages or into newly created malicious pages. Then, these scripts usually make the compromised websites respond with different web content to users that visit via or not via particular search engines. Specifically, once the compromised websites recognize that the requests are referred from specific search engines, the compromised websites may lead the users to malicious websites through multiple additional redirection hops. However, if the compromised websites recognize that the requests are directly from users, the compromised websites will return normal content rather than malicious content. Thus, this kind of cloaking makes the attack very stealthy and difficult to be noticed. In addition, the good reputation of these (compromised) websites (e.g., many are reputable .edu domains) essentially help boost the search engine ranks and access opportunities of malicious webpages. In this case, it is meaningful to discover those compromised websites as many as possible to stop such search poisoning attacks.

Most current state-of-the-art approaches to find such compromised websites merely utilize pre-selected key terms such as "Google Trends [6]", "Twitter Trending Topics [16]" or specific "spam words" to search on popular search engines. However, the number of newly discovered compromised websites by using this kind of approaches is highly restricted to those pre-selected key terms. First, the limited number of the pre-selected terms will restrict the number of compromised websites that could be found. Second, since these terms usually belong to some specific semantic topics, it will be hard to find more compromised websites in different categories. In addition, since many pre-selected key terms (e.g., Google Trends) are also widely used in benign websites, such approaches will also search out many benign websites leading to low efficiency.

In this paper, we propose a novel and efficient approach, PoisonAmplifier, to find compromised websites on the Internet that are utilized by attackers to launch search poisoning attacks. Specifically, PoisonAmplifier consists of five major components: Seed Collector, Promote Content Extractor, Term Amplifier, Link Amplifier, and Vulnerability Amplifier. Seed Collector initially collects a small seed set of compromised websites by searching a small number of terms on popular search engines. Then, for each known compromised website, Promote Content Extractor will extract "promoted web content", which is promoted by compromised website *exclusively* to search engine bots, but not seen by normal users. This web content is essentially promoted by attackers and usually has close semantic meaning with final malicious website (e.g, illicit pharmacy content). Through extracting specific query terms from "promoted web content", Term Amplifier will find more compromised websites by searching those query terms instead of simply using pre-selected key terms. The intuition behind designing this component is that attackers tend to provide similar key terms for search engine bots to index the webpages. For each compromised website, Link Amplifier first extracts two types of links: inner-links and outer-links. Inner-links refer to those links/URLs in the promoted web content of the compromised website. Outer-links refer to those links/URLs in the web content of other websites, which also have link of known compromised website. Then, Link Amplifier finds more compromised web sites by searching those inner-links and outer-links. The intuition is that the links in the promoted content tend to link to other compromised websites. Also, the websites linking to known compromised websites may also link to other (unknown) compromised websites. Vulnerability Amplifier will find more compromised websites, which have similar system or software vulnerabilities to existing known compromised websites. The intuition is that attackers tend to exploit similar vulnerabilities to compromise websites to launch search poisoning attacks. Through implementing a prototype system, PoisonAmplifier, our approach can find around 75,000 compromised web sites by starting from 252 known comprised websites within first 7 days and continue to find 827 new compromised websites everyday on average thereafter. In addition, our approach can achieve a high Amplifying Rate[1], much higher than existing work [22,23].

The major contributions of this paper can be summarized as follows.

- We propose PoisonAmplifier, a new, active, and efficient approach to find more compromised websites by analyzing attackers' promoted content and exploiting common vulnerabilities utilized by the attackers. Rather than simply using pre-selected (static) keywords to search on popular search engines, PoisonAmplifier is more effective and efficient to discover more compromised websites.
- We implement a prototype system and evaluate it on real-world data. Through our evaluation, PoisonAmplifier can find around 75,000 compromised websites by starting from only 252 verified compromised websites within first 7 days.[2] As a comparison with two recent studies using pre-selected terms, it takes 9 months to collect 63K compromised websites in [22] and 1 month to collect 1K compromised websites in [23]. Furthermore, PoisonAmplifier can discover around 4 times and 34 times compromised websites by analyzing the same number of websites, compared with [22] and [23].

The rest of the paper is structured as follows. We describe the background and our targeted search poisoning attacks in Section 2. We present the whole system design of PoisonAmplifier in Section 3 and the evaluation in Section 4. We discuss our limitations in Section 5 and current related work in Section 6. Finally, we conclude our work in Section 7.

[1] It is the ratio of the number of newly discovered compromised websites to the number of seed compromised websites.

[2] This speed is limited by the search rate constraint imposed by the Google search engine.

2 Background

In this Section, we first provide a brief overview on how our targeted search poisoning attacks work. Then, we present two typical methods utilized by attackers to promote malicious content in search engines to launch such search poisoning attacks.

2.1 Threat Model

Search Engine Optimization (SEO) is the process of optimizing websites to have higher search engine ranks. It includs white hat SEO and black hat SEO. Unlike white hat SEO, which increases search ranks by constructing websites to be more easily crawled by search engines, black hat SEO techniques attempt to obtain high rankings by violating search engines' policies such as keyword stuffing [12], hiding texts [9], and cloaking [2].

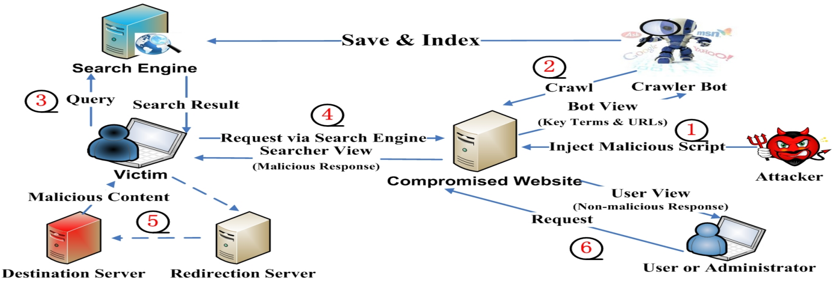


Fig. 1. The work flow of search poisoning attacks

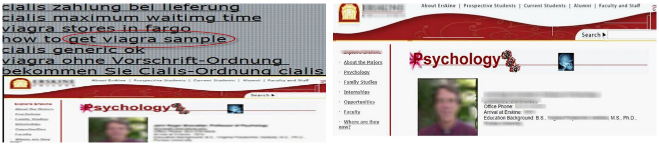
In our work, we focus on one specific type of black hat SEO techniques, named Search Poisoning Attack, which usually responds with malicious content to the users referred via search engines, while responds with non-malicious content to the direct visiting users. Next, we will describe how this search poisoning attack works.

As illustrated in Figure 1, to launch such a search poisoning attack, an attacker typically needs to first compromise a website by exploiting the website's system/software vulnerabilities, and then injects malicious scripts (PHP or Javascript) into the compromised website (labeled as ① in Figure 1). The core of such search poisoning attack is the ability for the compromised website to utilize injected malicious scripts to recognize different origins of the requests. Specifically, once the compromised website finds that the requests originate from crawler bots such as Google Bots, the website responds with web content containing special keywords and URLs injected by attackers. These special keywords and URLs are essentially what attackers desire to promote exclusively to the search engine crawler bots and hope to be indexed by the search engines(②). Then, if a user queries those keywords on search engines (③) and sends requests to the compromised website by clicking on the search results, the user will be a desired target victim because he shows interest in finding this website. In this case, the compromised server will provide malicious content to the user (④). The malicious response could directly be malicious web content or it redirects the user to malicious websites through multiple redirection hops (⑤). However, if the request originates from direct users (not via specific search engines), the attackers will not intent to expose the malicious content. This cloaking technique can make the attack very stealthy. In this case, the website will return non-malicious content (⑥).

In our work, we define the web content responded by the compromised website (after redirection if it has) to the crawler bot as "Bot View", to users via the search engine as "Searcher View", and to users *not* via the search engine as "User View". We apply a similar technique used in [22,25] to collect the ground truth on whether a website is compromised by search poisoning attack or not, i.e., whether its Searcher View and User View are different. More precisely, we *conservatively* consider the two views (Searcher/User) are different only when the final domain names (after redirection if there is any) are different [22]. In this way we can reduce false positives (due to dynamics in normal websites) and increase our confidence.[3]

2.2 Methods of Responding Malicious Content

As described in Section 2.1, in such search poisoning attacks, the compromised websites need to recognize the crawler bot to promote malicious content in search engines. Next, we describe two typical methods utilized by attackers to promote malicious content: tampering normal web pages, and creating new malicious web pages.



cialis zahlung bei lieferung
cialis maximum waitimg time
viagra stores in fargo
how to get viagra sample
cialis generic ok
viagra ohne Vorschrift-Ordnung
bekommen Sie Cialis-Ordnung cialis

(a) Bot View (b) User View

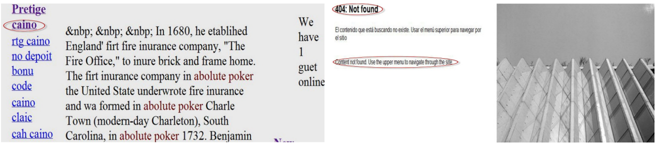
Fig. 2. A case study of tampering normal web pages

Tampering Normal Web Pages. In this way, when attackers compromise the website, they will inject malicious content into normal web pages. Once the compromised website recognizes that a request is from a search crawler bot, it will reply with both injected malicious content and normal web page content. Once the compromised website recognizes that a request is from a user's direct visit (not referred from search engines), it will reply with the normal webpage. As a case study illustrated in Figure 2, an attacker compromised a professor's

[3] Note that we may have very few false negatives using this conservative comparison. However that is not a problem for us because our goal is not on the precise detection but on the high efficiency in finding more compromised websites.

personal homepage to launch search poisoning attack. Under such an attack, the Bot View contains both illicit pharmacy content such as "get Viagra sample" (as seen in the upper part of Figure 2(a)) and the professor's personal information (as seen in the lower part of Figure 2(a)), and the User View only contains correct personal information (as seen in Figure 2(b)).

Creating Malicious Web Pages. In this way, unlike tampering existing normal web pages, attackers will upload or create totally new malicious web pages and only provide malicious content as Bot View to the search crawler bot. This content may be totally irrelevant to the themes of the whole website. As a case study illustrated in Figure 3, the attacker compromised a furniture company's website, which is implemented using a vulnerable version of WordPress [17]. Through exploiting the vulnerabilities of the WordPress, the attacker promoted casino content in Bot View to the search engine through creating a new malicious webpage hosted in the compromised website (as seen in Figure 3(a)). However, the attacker will provide a web page displaying "Not Founded" to users, who visit the same URL without using the search engine (as seen in Figure 3(b)).



Pretige
caino
rtg caino
no depoit
bonu
code
caino
claic
cah caino

&nbp; &nbp; &nbp; In 1680, he etablihed England' firt fire inurance company, "The Fire Office," to inure brick and frame home. The firt inurance company in abolute poker the United State underwrote fire inurance and wa formed in abolute poker Charle Town (modern-day Charleton), South Carolina, in abolute poker 1732. Benjamin

We have 1 guet online

404: Not found

El contenido que está buscando no existe. Usar el menú superior para navegar por el sitio

Content not found. Use the upper menu to navigate through the site

(a) Bot View (b) User View

Fig. 3. A case study of creating malicious Web pages

3 System Design

3.1 Intuition

Our design of PoisonAmplifier is based on the following three major intuitions:

Intuition 1: Attackers tend to use a similar set of keywords in multiple compromised websites (in the Bot View) to increase the visibility to desired users through search engines. Attackers usually artificially construct the content of Bot View, which will be indexed by search engines, to increase the chance of making compromised websites be searched through search engines. More specifically, similar to keyword stuffing [12], a common way of achieving this goal is to put popular keywords (those words are frequently searched by users on search engines such as Google Trends) into the Bot View. In this way, different compromised websites may share the similar popular keywords to draw attentions from victims. However, since many popular benign websites may also use these popular keywords and thus occupy high search ranks, it is difficult to guarantee high search ranks for those compromised websites that may be not very popular. As a supplement, another way is to buy some "distinguishable keywords" from specific websites [11]. These keywords may be not so popular as those popular terms. However, they tend to have low competition in search engines, i.e., they are not widely contained in the websites and can be effectively used to search out target websites. Thus, through promoting these words in the Bot View, the compromised websites could occupy high search ranks when users query these keywords in search engines. Thus, attackers may use these "distinguishable keywords" in multiple compromised websites to increase their search ranks.

In addition, since some attackers desire to reply malicious content to their target victims rather than arbitrary users, they tend to put specific keywords into the Bot View of compromised websites, which have close semantic meanings to the promoted websites. For example, some attackers tend to post pharmacy words into the Bot View, because they will finally mislead victims who are interested in buying pharmacy to malicious websites selling illicit pharmacy. In this way, different attackers who promote similar malicious content may spontaneously use similar keywords in the Bot View.

Based on this intuition, once we obtain those specific keywords injected by attackers into the Bot View of known compromised websites, we can search these keywords in search engines to find more compromised websites.

Intuition 2: Attackers tend to insert links of compromised websites in the Bot View to promote other compromised websites; and the websites containing URLs linking to known compromised websites may also contain URLs linking to other unknown compromised websites. To increase the chance of leading victims to malicious websites, attackers usually use multiple compromised websites to deliver malicious content. Thus, to increase the page ranks of those compromised websites to search engines or to help newly created webpages on compromised websites to be indexed by search engines, attackers tend to link these compromised websites with each other by inserting links of compromised websites into the Bot View. In addition, attackers with different malicious goals may spontaneously promote links of compromised websites into the same popular third-party websites such as forums and online social network websites, either because these third-party websites are easy to be indexed by search engines or they do not have sanitation mechanisms. *Based on this intuition, we can find more compromised websites by searching the URLs in the Bot View linking to known compromised websites, and by searching the URLs in the web content of other websites, which have already been exploited to post URLs linking to known compromised websites.*

Intuition 3: Attackers tend to compromise multiple websites by exploiting similar vulnerabilities. Once attackers compromise some specific websites by exploiting their system/software vulnerabilities to launch search poisoning attacks, they tend to use similar tricks or tools to compromise other websites with similar vulnerabilities to launch search poisoning attacks. *Based on this intuition, once we know the vulnerabilities exploited by attackers to some compromised websites, we can find more compromised websites by searching websites with similar vulnerabilities.*

3.2 System Overview

We next introduce the system overview of PoisonAmplifier, based on the three intuitions described in Section 3.1. As illustrated in Figure 4, PoisonAmplifier mainly contains five components: Seed Collector, Promoted Content Extractor, Term Amplifier, Link Amplifier, and Vulnerability Amplifier.

- Similar to other existing work [27,23], the goal of **Seed Collector** is to collect a seed set of compromised websites by searching initial key terms (e.g., Google Trends) in popular search engines.
- For each compromised website, **Promoted Content Extractor** will first work as a search engine bot to crawl the website's Bot View, and then work as a real user to obtain the websites' User View. Then, Promoted Content Extractor will extract those content that exists in the website's Bot View but does *not* exist in the website's User View. This content, defined as "promoted content", is essentially what attackers desire to promote into search engines.
- After extracting the promoted content, **Term Amplifier** extracts special key terms by analyzing the promoted content and querying these key terms in search engines to find more compromised websites.
- **Link Amplifier** extracts URLs in the promoted content. Link Amplifier will also extract URLs contained in the web content of third-party websites, which have already been posted links to known compromised websites. Then, Link Amplifier will analyze these URLs to find more compromised websites.
- By analyzing system/software vulnerabilities of those seed compromised websites and newly found compromised websites through using Term Amplifier and Link Amplifier, **Vulnerability Amplifier** finds more compromised websites by searching other websites with similar vulnerabilities.

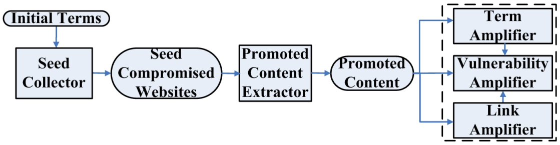


Fig. 4. The system architecture of PoisonAmplifier

3.3 Seed Collector

As illustrated in Figure 5, Seed Collector mainly uses the following four steps to collect seed compromised websites: (1) it first uses Google Trends [6], Twitter trends[16], and our customized key terms as initial key terms to search on search engines. (2) For each term, it will extract the links of the top M search results showed in the search engine[4]. (3) For each link, Seed Collector crawls its Searcher

[4] In our experiment, we choose $M = 200$.

View and User View through utilizing HttpClient-3.x package[10][5] to set different HTTP header parameters and values. Specifically, to crawl the Searcher View of the website linked by each search result, we send HTTP requests with customized Http Referrer (http://www.google.com/?q="term") to simulate a user to visit the website through searching Google. To crawl the User View, we send HTTP requests with customized values of UserAgent in the HTTP header (e.g., UserAgent: Mozilla/5.0 (Windows NT 6.1), AppleWebKit/535.2 (KHTML, like Gecko), Chrome/15.0.874.121, Safari/535.2) to simulate a user to directly visit the website. For both User View and Searcher View, the seed collector follows their redirection chains and gets their final destination URL. (4) For each link, if its final destination domains between User View and Searcher View are different, we consider that this linking website is compromised and output it as a compromised website.

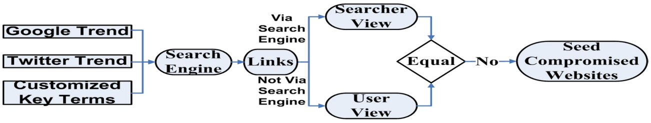


Fig. 5. Flow of Seed Collector

3.4 Promoted Content Extractor

As described in Section 2.1, the essence of the search poisoning attack is to recognize different request origins and provide different web content to crawler bots (Bot View), to users via search engines (Searcher View), and to users not via search engines (User View). And attackers tend to inject specific content into the Bot View to increase the chances of their compromised websites to be searched out in search engines. They may also tend to inject malicious content that is related to the final promoted destination malicious websites. This content is usually different from normal web content, and can not be seen by users without using search engines.

The goal of the Promoted Content Extractor is to extract that injected content in the Bot View of known compromised webpages, which may also be contained in other compromised websites. Note that the Bot View may also contain normal content that is not injected by attackers and will be displayed in the User View. To be more effective, PoisonAmplifier only extracts and analyzes the content that is in the Bot View but is *not* in the User View, i.e., the content will be indexed by crawler bots, but not be seen by users directly visiting the websites. As illustrated in Figure 6, for each compromised website, Promoted Content Extractor crawls its Bot View and User View through sending crafted requests from crawler bots and users without using search engines, respectively. Specifically, to crawl the Bot View, we send request with customized value of UserAgent

[5] This package can handle HTTP 3xx redirection and provide flexible HTTP header configuration functions.

in the HTTP header (e.g., UserAgent: Mozilla/5.0 (compatible; Googlebot/2.1; +http://www.google.com/bot.html)) to mimic a Google bot visit. Promoted Content Extractor crawls the User View in the same way as Seed Collector. Then, Promoted Content Extractor extracts HTML content that appears in the Bot View but not in the User View. Then, it will further filter web content that is used for displaying in the web browsers such as HTML Tags and CSS codes, and also remove dynamic web function related codes such as Javascripts, which are not unique enough to help further amplification. Finally, it outputs this extracted "Promoted Content" after filtering.

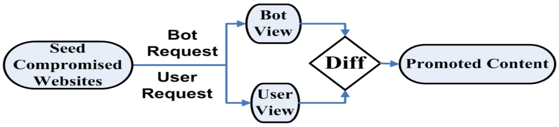


Fig. 6. Flow of Promoted Content Extractor

It is worth noting that some legitimate websites with dynamic server-side codes can also return different content or even redirect to different websites for every request no matter where the visit is from (User View or Bot View), which may lead to false positives in our extracted promoted content. To decrease this kind of false positives, we crawl the User View twice within a short time period. In this case, if the two User Views are different, we will conservatively consider that this website is *not* compromised (even its User View and Searcher View may be different) and discard it for promoted content extraction.

3.5 Term Amplifier

Based on Intuition 1 in Section 3.1, the goal of Term Amplifier is to find more compromised websites through searching specific query terms extracted from promoted content.

It is worth noting that if we use less distinguishable content as query terms to search, we can obtain a higher recall number (more compromised websites could be returned) but a lower accuracy (top search results are less likely to be compromised websites), and vice versa. In addition, in order to obtain a higher accuracy, it is practical to focus on analyzing replied search results with top search ranks rather than analyzing all search results. Thus, the essential part of Term Amplifier is how to extract effective query terms from promoted content, through searching which we can obtain as many compromised websites as possible with a high accuracy. One option is to use each word/phrase in the content as one query term. However, in this way, some terms may be so general that most returned websites are benign, leading to a low accuracy. Another option is to use the "n-gram" algorithm [13] ($n \geq 2$). In this way, some terms may be so distinguishable that many compromised websites will be missed, leading to a low recall number.

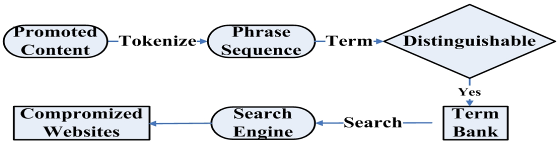


Fig. 7. Flow of Term Amplifier

In our work, we design an algorithm, named "distinguishable n-gram", to extract query terms. As illustrated in Figure 7, Term Amplifier first tokenizes the promoted content into a sequence of phrases $\{P_i|i = 1, 2, \ldots, N\}$ by using the tokenizer of any non-Alphanumeric character except "blank", such as "comma", "semicolon", and "question mark". Then, for each phrase P_i, Term Amplifier will *exactly search* it on the search engine. If the number of returned search results SN_i is lower than a threshold T_D[6], we consider P_i as a "distinguishable" term and directly add it into a term set, named TermBank. Otherwise, Term Amplifier combines the phrases of P_i and P_{i+1} as a new query term to search. If this new term is "distinguishable", we add it into TermBank; otherwise, Term Amplifier combines the phrases of P_i, P_{i+1} and P_{i+2} as a new term to search. This process will continue until the number of phrase in the new term is equal to n. If the new term with n phrases is still not "distinguishable", the algorithm will discard the phrase P_i. In this way, TermBank comprises all the distinguishable terms extracted from the promoted content. The detailed description of "distinguishable n-gram" algorithm is shown in Algorithm 1.

Algorithm 1. Distinguishable n-gram Algorithm

```
Tokenize promoted content into phrases $\{P_i|i = 1, 2, \ldots, N\}$
for $i := 1$ to $N$ do
    for $j := 0$ to $n - 1$ do
        Search "$P_i$ $P_{i+1}$ ... $P_{i+j}$" on the search engine to get $SN_i$
        if $SN_i \leq T_D$ then
            Add '$P_i$ $P_{i+1}$ ... $P_{i+j}$" into TermBank
            CONTINUE
        end if
    end for
end for
Return TermBank
```

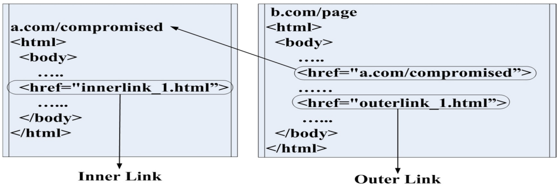
After building TermBank, similar to Seed Collector, Term Amplifier uses each query term in TermBank to search in the search engine and identifies compromised webpages through comparing their Searcher Views and User Views.

3.6 Link Amplifier

Based on Intuition 2 in Section 3.1, Link Amplifier first extracts two types of links: inner-links and outer-links. Inner-links refer to those links/URLs in the

[6] T_D can be tuned with the consideration of the tradeoff between the accuracy and the recall number. In our preliminary experiment, we choose $T_D = 1,000,000$.

promoted web content of the compromised websites (as illustrated in the left part of Figure 8). Outer-links refer to those links/URLs in the web content of third-party websites, which have been posted with URLs linking to known compromised websites (as illustrated in the right part of Figure 8). We utilize Google dork [7] to locate the outer-links. For example, if one compromised website "seed.com" is obtained through searching one seed term "seedTerm", then we obtain those websites through searching "*intext:seed.com intext:seedTerm*" on Google. Then we crawl all the websites in search results, which usually are blogs or comments that contain "seed.com" and other scam links. Then, similar to Term Amplifier, for each inner-link and outer-link, Link Amplifier crawls its Searcher View and User View, and considers the linking website as compromised website if the Searcher View and User View are different.



```
a.com/compromised
<html>
  <body>
    .....
  <href="innerlink_1.html">
    ......
  </body>
</html>
```

```
b.com/page
<html>
  <body>
    .....
    <href="a.com/compromised">
    ......
    <href="outerlink_1.html">
    ......
  </body>
</html>
```

Fig. 8. The illustration of inner-links and outer-links

We acknowledge that since those third-party websites may also post many benign links, many of outer-links will not link to compromised websites, leading to a relatively low accuracy. However, one of the benefits is that, through analyzing those outer-links, we can find more categories of compromised websites. For example, through analyzing outer-links from compromised websites selling illicit pharmacy, we could find compromised websites that promote other topics such as "adult/sexy content". One case study of a forum webpage posting outer-links to both "adult" and "pharmacy" websites can be seen in Appendix A. Furthermore, we can still somehow increase the accuracy through focusing on only those third-party websites that have posted scam terms. This is because that this kind of websites are more likely to be used to promote malicious content by attackers than other websites. Thus, the links posted in such websites are more suspicious.

3.7 Vulnerability Amplifier

Once an attacker compromises a website to launch search poisoning attack by exploiting specific system/software vulnerabilities of the websites, it is very likely that he uses the same vulnerability to compromise more websites. For example, once some attackers know about the vulnerabilities of some specific version of "WordPress" [17] and successfully use some existing tools to compromise some websites that are implemented through using that specific version of WordPress, they may try to find other vulnerable websites that are also implemented with that version of WordPress. One possible simple way of finding those vulnerable websites could be to search keywords such as "powered by WordPress" on search engines.

Based on Intuition 3 in Section 3.1, Vulnerability Amplifier essentially mimics the way of attackers to find those compromised websites. Specifically, Vulnerability Amplifier first collects compromised websites by using Term Amplifier and Link Amplifier. Then, it will analyze possible system/software vulnerabilities of those compromised websites and extract the web content signature of the websites that utilize those vulnerable software. In our preliminary work, we only focus on analyzing the vulnerabilities of one specific software WordPress[7], which is a very popular target for attackers[1]. For example, one vulnerability of "Timthumb.php" in the WordPress themes allows attackers to upload and execute arbitrary php scripts. Vulnerability Amplifier will find compromised websites through searching those websites that use WordPress and contain at least one scam word. Since the URLs of the websites developed using WordPress typically contain a string of "wp-content", we can find those websites through searching Google Dork "*inurl:wp-content intext:scamWord*". After visiting each of such websites, Vulnerability Amplifier examines whether it is compromised or not by comparing its Searcher View and User View.

Currently, Vulnerability Amplifier still requires some manual work to extract search signatures. In the future, we plan to incorporate some techniques similar to existing automatic signature generation studies (e.g., AutoRE [30]) to automate some of the tasks.

4 Evaluation

In this section, we evaluate PosionAmplifier in two stages. For the first stage, we evaluate PoisonAmplifier regarding its effectiveness, efficiency, and accuracy with first 7 days' data. We also check the "discovery diversity" among different components in terms of finding exclusive compromised websites, i.e, how different the discovered compromised websites by different components are. In addition, we examine how existing Google security diagnostic tools in labeling malicious/compromised websites work on our found compromised websites. In the second stage, we extend the time to 1 month to verify if the PoisonAmplifier can constantly find new compromised websites.

4.1 Evaluation Dataset

As mentioned in Section 3, the seed term set consists of three categories: Google Trends, Twitter Trends and our customized keywords. For the Google Trend Topics, we crawled 20 Google Trend keywords each day for a week. In this way, we collected 103 unique Google Trends topics. For the Twitter Trends, we collected top 10 hottest Twitter trends each day for a week. In this way, we

[7] Even though we only analyze the vulnerabilities of one specific software in this work, our approach can easily include other types of system/software vulnerabilities, which is our future work.

collected 64 unique Twitter Trends topics. For the customized key terms, we chose one specific category of scam words – pharmacy words[8]. Specifically, we chose 5 pharmacy words from one existing work [26], which provides several categories of scam words. We also manually selected another 13 pharmacy words from several pharmacy websites. Table 1 lists all 18 pharmacy words used in our study.

Table 1. 18 seed pharmacy words

kamagra	diflucan	levitra	phentermine	propecia	lasix
viagra	amoxil	xanax	cialis	flagyl	propeciatramadol
zithromax	clomid	Viagra super active	cialis super active	cipro	pharmacy without prescription

Then, for each of 18 pharmacy words, we obtained another 9 Google Suggest words through Google Suggest API [8]. In this way, we finally collected 165 unique pharmacy words. Table 2 summarizes the number and unique number of seed terms for each category.

Table 2. The number of seed terms for three different categories

Category	# of terms	# of unique terms
Google Trend	140	103
Twitter Trend	70	64
Pharmacy	180	165
Total	390	332

Then, for each of these 332 unique seed terms, we searched it on "Google.com" and collected the top 200 search results[9]. Then, for each search result, we use the similar strategy as in [22] to determine whether a website is compromised by examining whether the domain of its Searcher View and User View are different. In this way, we finally obtained 252 unique seed compromised websites through using those 332 seed terms. We denote this dataset as S_I, which is used in Stage I. After one week's amplification process, we denote the amplified terms and compromised websites from Stage I as S_{II}, which is the input for Stage II to recursively run PoisonAmplifier for 1 month.

[8] In our preliminary experiment, we only use pharmacy words. However, our approach is also applicable to other categories of words such as "adult words" or "casino words".

[9] In our experiment, we only focus on the search poisoning attacks on Google. However, our approach can be similarly extended to other search engines such as "yahoo.com" and "baidu.com". Also, the number of 200 can be tuned according to different experiment settings.

4.2 Evaluation Results

Effectiveness. To evaluate the effectiveness of our approach, we essentially check how many new compromised websites can be found through amplifying dataset S_I. To measure the effectiveness, we use a metric, named "Amplifying Rate (AR)", which is the ratio of the number of newly found compromised websites to the number of seed compromised websites. Thus, a higher value of AR implies that the approach is more effective in finding compromised websites based on the seed compromised websites.

Table 3 shows the number of newly found compromised websites for each component. We can see that Term Amplifier has the highest AR of 323, which confirms that Term Amplifier can be very effective in discovering compromised websites. Even though Inner-link Amplifier and Outer-link Amplifier have relatively lower ARs than Term Amplifier, they can still discover over 10 times more compromised websites from the seeds. Actually, the reason why Term Amplifier can obtain a higher AR is mainly because we can extract much more query terms than inner-links and outer-links from the promoted content. In this way, we can essentially search out much more websites that contain the query terms from the search engine. In addition, even though we only focus on analyzing one specific software in our Vulnerability Amplifier, we can still discover over 4 times more compromised websites from the seeds. Overall, starting from only 252 seed compromised websites, these four strategies can totally discover around 75,000 unique compromised websites, and achieve a overall high amplifying rate of 296. The distribution information of these compromised websites in terms of their Top Level Domain(TLD) is show in Figure 10(a).

Table 3. The effectivenss of PoisonAmplifier

Component	# seed compromised website	# unique compromised websites	Amplifying Rate
TermAmplifier	252	69,684	323.03
Inner-linkAmplifier	252	2,468	10.63
Outer-linkAmplifier	252	2,401	10.34
VulnerabilityAmplifier	252	482	4.49
Total (Unique)	252	74,671	296.31

Efficiency. To evaluate the efficiency of our approach, we essentially examine whether the websites visited by PoisonAmplifier are more likely to be compromised websites or not. To measure the efficiency, we use another metric, named "Hit Rate (HR)", which is the number of newly found compromised websites to the total number of websites visited by PoisonAmplifier. Thus, a higher Hit Rate implies that our approach is more efficient, because it means our approach can find more compromised websites by visiting fewer websites. Next, we evaluate the efficiency of individual amplification component, as well as the efficiency of different types of query keywords.

Component Efficiency. Table 4 shows the number of visited websites, the number of newly found compromised websites, and the values of hit rate for each component.

Table 4. The efficiency of different components

Component	# Visited Websites	# Compromised Websites	Hit Rate
TermAmplifier	684,540	69,684	10.18%
Inner-linkAmplifier	3,097	2,468	79.69%
Outer-linkAmplifier	353,475	2,401	0.68%
VulnerabilityAmplifier	45,348	482	1.06%
Total	1,086,496	74,671	6.87%

From this table, we can see that Inner-link Amplifier can achieve the highest hit rate of 79.69%. This confirms that attackers tend to promote links of compromised websites to the search engine bots. The hit rate of Term Amplifier is around 10%, which is lower than that of Inner-link Amplifier. However, Term Amplifier can discover much more compromised websites than that of Inner-link Amplifier in terms of overall quantity, because we essentially extract significantly more terms than inner-links to search on the search engine. The hit rate of Outer-link Amplifier is relatively low, which is mainly because most of those outer-links are benign or do not have redirections. However, through using Outer-link Amplifier, we can find new types of scam terms promoted by different attackers. This is very useful to increase the diversity of the seed terms and to find more types of compromised websites. Vulnerability Amplifier also has a relatively low hit rate, because most top ranked websites with "WordPress" are benign. However, similar to Outer-link Amplifier, Vulnerability Amplifier also provides a method to find more (new) types of scam words and compromised websites.

Term Efficiency. We also analyze the term efficiency in finding compromised websites, i.e., which kinds of terms can be used to efficiently search out compromised (rather than normal) websites. Specifically, we compare three types of terms: seed terms (those 332 seed terms used in the Seed Collector), promoted phrases (the sequence of phrases obtained through tokenizing promoted content), and distinguishable terms (all the terms in TermBank obtained by utilizing "Distinguishable n-gram Algorithm"). Essentially, we use these three types of terms to search on Google to find compromised websites by utilizing Term Amplifier.

As seen in Figure 9, among these three types of terms, our extracted distinguishable terms can achieve the highest hit rate. Specifically, around 60% of distinguishable terms' hit rates are less than 0.2, while around 80% of promote phrases and 90% of seed terms have such values. This implies that using distinguishable terms is more effective to find compromised websites. In addition, over 60% of seed terms' hit rates are nearly zero, which shows that the current pre-selected terms are not very efficient compared to our new terms extracted from promoted content.

To find what specific terms are most efficient, we further analyze the terms with the top five hit rates in TermBank. As seen in Table 5, we can see that all these five terms' hit rates are higher than 79%. In addition, we also find that three of these five terms have the same semantic meanings of sub-phrase

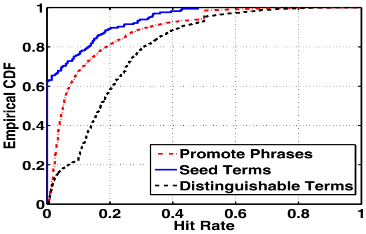


Fig. 9. Hit rate distribution

as “No Prescription Needed”. That may be due to the reason that attackers frequently use such phrases to allure victims, because many kinds of pharmacy drugs need prescription to buy in reality. The other two terms contain the names of two popular drugs: “Diflucan” (an anti-fungal medicine) and “Nimetazepam” (working specifically on the central nervous system).

Table 5. Terms with Top Five Hit Rates

Term	Hit Rate
Order Diflucan from United States pharmacy	90%=180/200
Buy Online No Prescription Needed	87%=174/200
Buy Cheap Lexapro Online No Prescription	85%=170/200
Online buy Atenolol without a prescription	83%=166/200
Nimetazepam interactions	79.5%=159/200

Diversity among Different Components. In this section, we analyze the diversity of newly found compromised websites among different components, i.e., we examine how many compromised websites of each component are exclusive, which can not be found by other components. The intuition is that if one component can find the compromised websites that can not be found by another component, then these components are very complementary and they can be combined together to be effective in discovering compromised websites. To measure the diversity, we use a metric, named “Exclusive Ratio (ER)”, which is the ratio of the number of compromised websites that are only found by this component to the total number of compromised websites found by this component.

As seen in Table 4, we can find that all four components can obtain high exclusive ratios, higher than 88%. This observation shows that all these four components are complementary and it makes perfect sense to combine them together to achieve high effectiveness in discovering new compromised websites. Also, we can find that Term Amplifier’s exclusive ratio is over 99%. That is mainly because Term Amplifier can find more compromised websites through visiting more websites.

Comparison with Existing Work. In this experiment, we first compare the performance of PoisonAmplifier with two existing work: Leontiadis et al. [22]

Table 6. Exclusive ratio of different components

Component	TermAmplifier	Inner-linkAmplifier	Outer-linkAmplifier	VulnerabilityAmplifier
Exclusive Ratio	99.56%	96.11%	89.09%	88.77%

and Lu et al. [23] (both of them use pre-selected terms). To further verify the performance of the pre-selected term method used in above two work, we tested this method with our dataset. Table 7 shows the comparison result.

Table 7. The comparison of effectiveness with existing work

Research Work	**# Seed Terms**	**# Visited Websites**	**# Compromised Websites**	**Hit Rate**	**Time**
Leontiadis et al. [22]	218	3,767,040	63,000	1.67%	9 months
Lu et al. [23]	140	500,000	1,084	0.2%	1 months
Pre-selected terms	322	64,400	252	0.39%	7 days
PoisonAmplifier	332	1,086,496	74,671	6.87%	7 days

From this table, we can see that compared with [22], based on a similar number of seed terms, our work can find more compromised websites with a higher hit rate within a significantly shorter period. Also, compared with [23], our approach uses much fewer seed terms, but discovers much more compromised websites with a much higher hit rate within a significantly shorter period. Compared with pre-selected terms method, with the same number of seed terms and same evaluation time, our approach can find much more compromised websites. Also, the hit rate of our approach is the highest, which is around 4 times and 34 times as that of [22] and [23], respectively. This observation shows that our approach is more efficient and effective in discovering/collecting compromised websites, since our approach does not highly rely on the pre-selective keywords (pre-selective keywords typically lead to a low hit rate, which has also been verified by [19]), which are used by both existing approaches.

Comparison with Google Security Diagnostic Tools. We conducted another experiment to further evaluate the effectiveness of our approach. We want to test whether our newly found compromised websites are also detected in a timely fashion by Google's two state-of-the-art security diagnostic tools: Google Safe Browsing (GSB) [5] and Google Security Alert [4]. GSB is a widely used URL blacklist to label phishing websites and malware infected websites. Google Security Alert is another security tool, which labels compromised websites through showing the message "This site maybe compromised" within Google search results.

We first check how many new compromised websites found by each component are labelled as "phishing" or "malware infected". As seen in Table 8, we found that GSB labels only 547 websites as "malware infected" and zero as "phishing" through examining all $74,671$ newly found compromised websites. We next check how Google Security Alert works on our newly found compromised websites. Specifically, we sampled 500 websites (which were randomly selected from those

Table 8. Labeling results by using GSB

Component	# Compromised Websites	# Phishing	# Malware Infected
TermAmplifier	69,684	0	536
Inner-linkAmplifier	2,468	0	2
Outer-linkAmplifier	2,401	0	3
VulnerabilityAmplifier	482	0	6
Total (Unique)	74,671	0	547

$74,671$ compromised websites) and finally found none of them were labelled as compromised.

Through the above experiments, we can find that most of our newly found compromised websites have not been detected by current Google security diagnostic tools. Although we do not argue that our approach is more effective than those two Google security tools, this observation shows that our approach can be effectively utilized to discover many new compromised websites that Google has not yet detected.

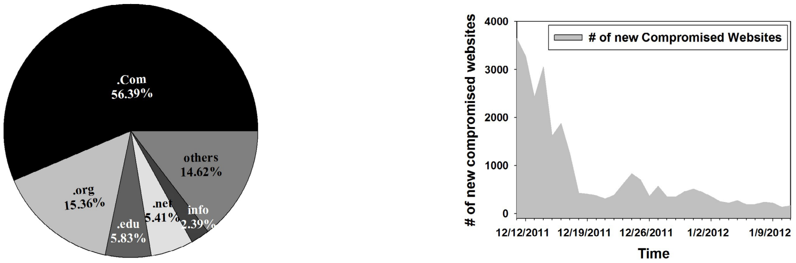
Accuracy. In this paper, we collect the ground truth through comparing the difference between Searcher View and User View, which proves to be a conservative and effective approach to identify search poisoning attacks [22,25]. To further gain more confidence, we have conducted a manual verification on 600 randomly sampled URLs from all labelled compromised websites, and all of these sample websites were manually verified as indeed compromised websites.

Constancy. To evaluate the constancy of our approach, we essentially examine whether PoisonAmplifier can continue to find new compromised websites over time. Figure 10(b) is the distribution of new crawled compromised websites in Stage II. We can see that during the first several days, our system can find more new compromised websites because Term Amplifier inherits a large number of terms from data S_I. With these terms, our system can efficiently find other compromised websites sharing similar terms. After that, the daily newly found compromised websites decrease quickly due to the exhaustion of terms. However, Link Amplifier and Vulnerability Amplifier can keep finding new terms and compromised websites everyday because the attackers keep promoting and attacking everyday. In this case, our system can still constantly find new compromised websites everyday leading to 26,483 new found compromised websites during 1 month's recursive amplification process.

5 Limitations

In this section, we will discuss the limitations of our work.

We first acknowledge that since we mainly utilize pharmacy keywords as initial terms in our evaluation, this method may generate some bias. We use illicit pharmacy as a specific case study to evaluate our approach mainly because it is a typical target of search poisoning attack. However, our approach can be



(a) Distribution based on Top Level Domain (b) Daily new found compromised websites.

Fig. 10. Statistics of found compromised websites

easily applied to other scenarios such as fake AntiVirus or web scams through changing customized keywords. In addition, through our evaluation, we can also observe that even though we use pharmacy keywords as initial customized keywords, those newly found compromised websites could be injected with content related to other scenarios. Thus, PoisonAmplifier can discover a broader range of compromised websites, instead of being restricted to only those used to promote illicit pharmacy by attackers.

We also acknowledge that since it is still difficult for Promoted Content Extractor to accurately filter all dynamic content, this may decrease the performance (in terms of hit rate) of our approach. However, visiting websites multiple times can somehow relieve this kind of problem. In addition, we indeed manually checked several hundred randomly sampled compromised websites and we have not found such kind of false positives so far. Also, our Distinguishable n-gram Algorithm can filter some general terms (generate by dynamic content) and reduce their impact.

In addition, we realize that once attackers know about the principle of our approach, they may try to evade our approach through providing non-malicious content to our Bot View with the utilization of IP-based cloaking techniques. For example, they may refuse to deliver malicious content if they find the IP address from our crafted Google bot crawler does not match known addresses of Google. However, as an alternative technique of Bot View by manipulating Http Referer, we can use the cache results of search engines such as Google cache as Bot View. In such way, we can obtain the Bot View of those compromised websites, as long as attackers want to make their content crawled and indexed by popular search engines to launch search poisoning attacks. Besides, attackers may also try to decrease the effectiveness of our approach through inserting noisy content into their injected content. However, if the noisy content is general, our system will drop them based on our "Distinguishable n-gram Algorithm". Otherwise we can still consider these noisy data as "real" promoted content as long as they are shared in multiple compromised websites.

6 Related Work

Measurement and Understanding of Search Poisoning Attacks. Cloaking techniques are commonly used in search poisoning attacks, which have been analyzed and measured in many existing work [27] [28] [29]. Wu et al. present the earliest measurement study of cloaking techniques [27]. This work provides a way of identifying cloaking techniques through crawling webpages multiple times (using both user and crawler identifiers) and comparing the replied content.

In terms of search poisoning attacks, Wang et al. [25] investigate the dynamic property and show that the majority of compromised websites remain high in search results. Leontiadis et al. present a detailed measurement study of search poisoning attacks in the scenario of illicit online pharmacy [22]. Moore et al. provide an in-depth study on Google Trending Terms and Twitter Trending Topics that are abused to attract victims [24]. Through analyzing 60 million search results and tweets, this work characterizes how trending terms are used to perform search poisoning attacks and to spread social network spam.

Detection of Search Poisoning Attacks. Besides understanding search poisoning attacks, several approaches have been proposed to detect such attacks. John et al. [21] analyze a specific case study of search poisoning attacks and propose an automatic detection method based on detection insights obtained through the observation that the new created page(named SEO page in the paper) always contains trending terms and exhibit patterns not previously seen by search engines on the same domain. Lu et al. [23] present an in-depth study on analyzing search poisoning attacks and redirection chains, then build a detection system based on several detection features extracted from browser behaviors, network traffic, and search results.

Unlike most existing studies that try to understand or detect search poisoning attacks, our work focuses more on efficiently and effectively identifying(amplifying) more websites compromised by the search poisoning attacks, given a small seed set. We think this is an important problem not addressed so far. Our work is essentially motivated by these existing studies and is complementary to them.

In addition, the intuition behind our work is that we try to use attackers' tricks against them. Specifically, our work tries to find compromised websites through exploiting attackers' promoted content, which are injected by the attackers to attract the search engine bot and search traffic. In such case, John et al. [20] have similar ideas but target on a different problem, in which the authors propose a framework to find more malicious queries by generating regular expressions from a small set of malicious queries. In a recent concurrent study, EvilSeed[19] also shares similar inspiration but with different target and techniques. It searches the web for pages that are likely malicious by starting from a small set of malicious pages. To locate the other nearby malicious pages, they design several gadgets to automatically generate search queries. However, with the three oracles used in their work, Google's Safe Browing blacklist[5], Wepawet[18], and a custom-built tool to detect sites that host fake AV tools, EvilSeed cannot handle more stealthy attacks such as Search Poisoning Attacks discussed in this paper. That is, EvilSeed can only find a small subset of these cloaking attacks that PoisonAmplifier can find. In addition, since PoisonAmplifier extracts the content that the attackers intend to promote while EvilSeed uses much more generic signatures in its SEO gadget, PoisonAmpifier can find more search poisoning compromised websites more *efficiently* and *effectively* than EvilSeed, e.g., the hit rate of EvilSeed is 0.93% in its SEO gadget compared with 6.87% hit rate in PoisonAmplifier. We consider PoisonAmplifier as a good complement to EvilSeed.

7 Conclusion

In this work, we have designed and implemented a novel system, PoisonAmplifier, to discover compromised websites that are utilized by attackers to launch search poisoning attack. Based on intrinsic properties of search poisoning attack, PoisonAmplifier first extracts attackers' promotion content in a small seed set of known compromised websites. Then, PoisonAmplifier utilizes Term Amplifier and Link Amplifier to find more compromised websites through searching specific terms and links in those promotion content on search engines. PoisonAmplifier also utilizes Vulnerability Amplifier to find more compromised websites, which have similar system/software vulnerabilities to existing known compromised websites. Our evaluation shows that PoisonAmplifier can find nearly 75,000 compromised websites by starting from 252 verified compromised websites within first 7 days. Also, compared with two related work, PoisonAmplifier can find around 4 times and 34 times compromised websites by analyzing the same number of websites.

Acknowledgment. This material is based upon work supported in part by the National Science Foundation under Grant CNS-0954096 and the Texas Higher Education Coordinating Board under NHARP Grant no. 01909. 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 National Science Foundation and the Texas Higher Education Coordinating Board.

References

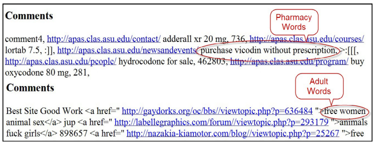
1. 50,000 websites infected with spam from wplinksforwork, http://news.softpedia.com/news/50-000-Websites-Infected-with-Spam-From-Wplinksforwork-223004.shtml/
2. Cloaking, http://en.wikipedia.org/wiki/Cloaking
3. Cmu researcher finds web hackers profiting from illegal online pharmacies, http://www.darkreading.com/insider-threat/167801100/security/client-security/231400204/cmu-researcher-finds-web-hackers-profiting-from-illegal-online-pharmacies.html

4. Google fights poisoned search results, http://www.securitynewsdaily.com/google-poisoned-search-results-0603/
5. Google safe browsing, http://code.google.com/apis/safebrowsing/
6. Google trend, http://www.google.com/trends
7. Googledork, http://googledork.com/
8. Googlesuggest, http://code.google.com/p/google-refine/wiki/SuggestApi
9. Hiding text with css for seo, http://www.seostandards.org/seo-best-practices/hiding-text-with-css-for-seo.html
10. Httpclient, http://hc.apache.org/httpclient-3.x/
11. The keyword shop, http://www.blackhatworld.com/blackhat-seo/buy-sell-trade/
12. Keyword stuffing, http://www.seo.com/blog/keyword-stuffing/
13. N-gram algorithm, http://en.wikipedia.org/wiki/N-gram
14. The pharmacy example, http://www.cmu.edu/news/stories/archives/2011/august/aug11_onlinepharmacyhackers.html
15. Royal wedding, obama birth certificate search poisoned with fake av links, http://www.eweek.com/c/a/Security/Royal-Wedding-Obama-Birth-Certificate-Search-Poisoned-with-Fake-AV-Links-489242/
16. Trending topics, http://support.twitter.com/entries/101125-about-trending-topics
17. Word press, http://wordpress.com/
18. Cova, M., Kruegel, C., Vigna, G.: Detection and Analysis of Drive-by-Download Attacks and Malicious JavaScript Code. In: International World Wide Web Conference, WWW 2010 (2010)
19. Invernizzi, L., Comparetti, P., Benvenuti, S., Kruegel, C., Cova, M., Vigna, G.: EVILSEED: A Guided Approach to Finding Malicious Web Pages. In: IEEE Symposium on Security and Privacy, Oakland (2012)
20. John, J., Yu, F., Xie, Y., Abadi, M., Krishnamurthy, A.: Searching the Searchers with SearchAudit. In: Proceedings of the 19th USENIX Security (2010)
21. John, J., Yu, F., Xie, Y., Abadi, M., Krishnamurthy, A.: deSEO: Combating search-result poisoning. In: Proceedings of the 20th USENIX Security (2011)
22. Leontiadis, N., Moore, T., Christin, N.: Measuring and analyzing search-redirection attacks in the illicit online prescription drug trade. In: Proceedings of the 20th USENIX Security (2011)
23. Lu, L., Perdisci, R., Lee, W.: SURF: Detecting and Measuring Search Poisoning. In: Proceedings of ACM Conference on Computer and Communications Security, CCS 2011 (2011)
24. Moore, T., Leontiadis, N., Christin, N.: Fashion Crimes: Trending-Term Exploitation on the Web. In: Proceedings of ACM Conference on Computer and Communications Security, CCS 2011 (2011)
25. Wang, D., Savage, S., Voelker, G.: Cloak and Dagger: Dynamics of Web Search Cloaking. In: Proceedings of ACM Conference on Computer and Communications Security, CCS 2011 (2011)
26. Wang, Y., Ma, M., Niu, Y., Chen, H.: Double-Funnel: Connecting Web Spammers with Advertisers. In: Proceedings of the 16th International Conference on World Wide Web, pp. 291–300 (2007)
27. Wu, B., Davison, B.: Cloaking and redirection: A preliminary study. In: Adversarial Information Retrieval on the Web(AIRWeb) (2005)

28. Wu, B., Davison, B.: Identifying link farm spam pages. In: Special Interest Tracks and Posters of the International Conference on World Wide Web (2005)
29. Wu, B., Davison, B.: Detecting semantic cloaking on the Web. In: Proceedings of International Conference on World Wide Web, WWW 2006 (2006)
30. Xie, Y., Yu, F., Achan, K., Panigrahy, R., Hulten, G., Osipkov, I.: Spamming Botnet: Signatures and Characteristics. In: Proceedings of ACM SIGCOMM 2008 (2008)

A Case Study of Outer-Link

As seen in Figure 11, we first find the forum webpages through searching the websites that are known compromised websites - here is a pharmacy target compromised website. Then, through analyzing the forum webpage's content, we can also find other compromised websites with "Adult" content.



Comments

comment4, http://apas.clas.asu.edu/contact/ adderall xr 20 mg, 736, http://apas.clas.asu.edu/courses/ lortab 7.5, :]], http://apas.clas.asu.edu/newsandevents/ purchase vicodin without prescription, >:[[[, http://apas.clas.asu.edu/people/ hydrocodone for sale, 462803, http://apas.clas.asu.edu/program/ buy oxycodone 80 mg, 281,

Comments

Best Site Good Work <a href=" http://gaydorks.org/oc/bbs//viewtopic.php?p=636484 ">free women animal sex</a> jup <a href=" http://labellegraphics.com/forum//viewtopic.php?p=293179 ">animals fuck girls</a> 898657 <a href=" http://nazakia-kiamotor.com/blog//viewtopic.php?p=25267 ">free

Fig. 11. Example of outer-link

DEMACRO: Defense against Malicious Cross-Domain Requests

Sebastian Lekies[1], Nick Nikiforakis[2], Walter Tighzert[1],
Frank Piessens[2], and Martin Johns[1]

[1] SAP Research, Germany
[2] IBBT-DistriNet, KU Leuven, 3001 Leuven, Belgium
{sebastian.lekies,walter.tighzert,martin.johns}@sap.com,
{nick.nikiforakis,frank.piessens}@cs.kuleuven.be

Abstract. In the constant evolution of the Web, the simple always gives way to the more complex. Static webpages with click-through dialogues are becoming more and more obsolete and in their place, asynchronous JavaScript requests, Web mash-ups and proprietary plug-ins with the ability to conduct cross-domain requests shape the modern user experience. Three recent studies showed that a significant number of Web applications implement poor cross-domain policies allowing malicious domains to embed Flash and Silverlight applets which can conduct arbitrary requests to these Web applications under the identity of the visiting user. In this paper, we confirm the findings of the aforementioned studies and we design *DEMACRO*, a client-side defense mechanism which detects potentially malicious cross-domain requests and de-authenticates them by removing existing session credentials. Our system requires no training or user interaction and imposes minimal performance overhead on the user's browser.

1 Introduction

Since the release of the World Wide Web by CERN, the online world has dramatically changed. In this ever-expanding and ever-changing Web, old technologies give way to new ones with more features enabling developers to constantly enrich their Web applications and provide more content to users. This evolution of the Web is one of the main reasons that the Internet, once accessible by an elite few, is now populated by almost 2 billion users[1].

Two of the most popular platforms for providing enriched Web content are Adobe Flash and Microsoft Silverlight.Through their APIs, developers can serve data (e.g. music, video and online games) in ways that couldn't be traditionally achieved through open standards, such as HTML. The latest statistics show a 95% and 61% market penetration of Flash and Silverlight respectively, attesting towards the platforms' popularity and longevity [17].

Unfortunately, history and experience have shown that functional expansion and attack-surface expansion go hand in hand. Flash, due to its high market

[1] http://www.internetworldstats.com

D. Balzarotti, S. Stolfo, and M. Cova (Eds.): RAID 2012, LNCS 7462, pp. 254–273, 2012.

penetration, is a common target for attackers. The last few years have been a showcase of "zero-day" Flash vulnerabilities where attackers used memory corruption bugs to eventually execute arbitrary code on a victim's machine [1].

Apart from direct attacks against these platforms, attackers have devised ways of using legitimate Flash and Silverlight functionality to conduct attacks against Web applications that were previously impossible. One of the features shared by these two platforms is their ability to generate client-side cross-domain requests and fetch content from many remote locations. In general, this is an opt-in feature which requires the presence of a policy configuration. However, in case that a site deploys an insecure wildcard policy, this policy allows adversaries to conduct a range of attacks, such as leakage of sensitive user information, circumvention of CSRF countermeasures and session hijacking. Already, in 2007 a practical attack against Google users surfaced, where the attacker could upload an insecure cross-domain policy file to Google Docs and use it to obtain cross-domain permissions in the rest of Google's services [18]. Even though the security implications of cross-domain configurations are considered to be well understood, three recent studies [13,14,9] showed that a significant percentage of websites still utilize highly insecure policies, thus, exposing their user base to potential client-side cross-domain attacks.

To mitigate this threat, we present *DEMACRO*, a client-side defense mechanism which can protect users against malicious cross-domain requests. Our system automatically identifies insecure configurations and reliably disarms potentially harmful HTTP requests through removing existing authentication information. Our system requires no training, is transparent to both the Web server and the user and operates solely on the client-side without any reliance to trusted third-parties.

The key contributions of this paper are as follows:

- To demonstrate the significance of the topic matter, we provide a practical confirmation of this class of Web application attacks through the analysis of two vulnerable high-profile websites.
- We introduce a novel client-side protection approach that reliably protects end-users against misconfigured cross-domain policies/applets by removing authentication information from potentially malicious situations.
- We report on an implementation of our approach in the form of a Firefox extension called *DEMACRO*. In a practical evaluation we show that *DEMACRO* reliably protects against the outlined attacks while only implying a negligible performance overhead.

The rest of this paper is structured as follows: Section 2 provides a brief overview of cross-domain requests and their specific implementations. Section 3 discusses the security implications of misconfigured cross-domain policies, followed by two novel real-world use cases in Section 4. Section 5 presents in detail the design and implementation of *DEMACRO*. Section 6 presents an evaluation of our defense mechanism, Section 7 discusses related work and we finally conclude in Section 8.

2 Technical Background

In this section we will give a brief overview of client-side cross-domain requests.

2.1 The Same-Origin Policy

The Same-Origin Policy (SOP) [19] is the main client-side security policy of the Web. In essence, the SOP enforces that JavaScript running in the Web browser is only allowed access to resources that share the same origin as the script itself. In this context, the origin of a resource is defined by the characteristics of the URL (namely: protocol, domain, and port) it is associated with, hence, confining the capabilities of the script to its *own* application. The SOP governs the access both to local, i.e., within the browser, as well as remote resources, i.e., network locations. In consequence, a JavaScript script can only directly create HTTP requests to URLs that satisfy the policy's same-origin requirements. Lastly, note that the SOP is not restricted to JavaScript since other browser technologies, such as Flash and Silverlight, enforce the same policy.

2.2 Client-Side Cross-Domain Requests

Despite its usefulness, SOP places limits on modern Web 2.0 functionality, e.g., in the case of Web mash-ups which dynamically aggregate content using cross-domain sources. While in some scenarios the aggregation of content can happen on the server-side, the lack of client-side credentials and potential network restrictions could result in a less-functional and less-personalized mash-up. In order to accommodate this need of fetching resources from multiple sources at the client-side, Flash introduced the necessary functionality to make controlled client-side cross-domain requests. Following Flash's example, Silverlight and newer versions of JavaScript (using CORS [25]) added similar functionality to their APIs. For the remainder of this paper we will focus on Flash's approach as it is currently the most wide spread technique [14]. Furthermore, the different techniques are very similar so that the described approach can easily be transferred to these technologies.

2.3 An Opt-In Relaxation of the SOP

As we will illustrate in Section 3, a general permission of cross-domain requests would result in a plethora of dangerous scenarios. To prevent these scenarios, Adobe designed cross-domain requests as a server-side opt-in feature. A website that desires its content to be fetched by remote Flash applets has to implement and deploy a cross-domain policy which states who is allowed to fetch content from it in a white-list fashion. This policy comes in form of an XML file (`crossdomain.xml`) which must be placed at the root folder of the server (see Listing 1 for an example). The policy language allows the website to be very explicit as to the allowed domains (e.g. `www.a.net`) as well as less explicit through the use of wildcards (e.g. `*.a.net`). Unfortunately the wildcard can be used by itself, in which case all domains are allowed to initiate cross-domain requests and fetch content from the server deploying this policy. While this can be useful in case of well-defined public content and APIs, in many cases it can be misused by attackers to perform a range of attacks against users.

Listing 1. Exemplary crossdomain.xml file

```
<cross-domain-policy>
  <site-control
    permitted-cross-domain-policies="master-only" />
  <allow-access-from domain="a.net"/>
</cross-domain-policy>
```

2.4 Client-Side Cross-Domain Requests with Flash

Figure 1 gives an overview of how Flash conducts client-side cross-domain requests in a legitimate case. (The general scenario is equivalent for Silverlight and only differs in the name and the structure of its policy file). If the domain `a.net` would like to fetch data from the domain `b.net` in the user's authentication context, it has to include an applet file that implements cross-domain capabilities. This file can either present the fetched data directly or pass it on to JavaScript served by `a.net` for further processing. As already explained earlier, `b.net` has to white-list all domains that are allowed to conduct cross-domain requests. Therefore, `b.net` hosts a cross-domain policy named `crossdomain.xml` in it's root folder. (So the url for the policy-file is `http://b.net/crossdomain.xml`). If the Flash applet now tries to conduct a requests towards `b.net`, the Flash Player downloads the cross-domain policy from `b.net` and checks whether `a.net` is white-listed or not. If so, the request is granted and available cookies are attached to the request. If `a.net` is not white-listed the request is blocked by the Flash Player in the running browser.

3 Security Implications of Client-Side Cross-Domain Requests

In this section we present two classes of attacks that can be leveraged by an adversary to steal private data or to circumvent CSRF protection mechanisms.

3.1 Vulnerable Scenario 1: Insecure Policy

For this section we consider the same general setup as presented in section 2.4. This time however, `b.net` hosts personalized, access-controlled data on its domain and at the same time allows cross-domain requests from any other domain by white-listing a wildcard ("*") in its cross-domain policy. As a result any Flash/Silverlight applet is allowed to conduct arbitrary requests towards `b.net`

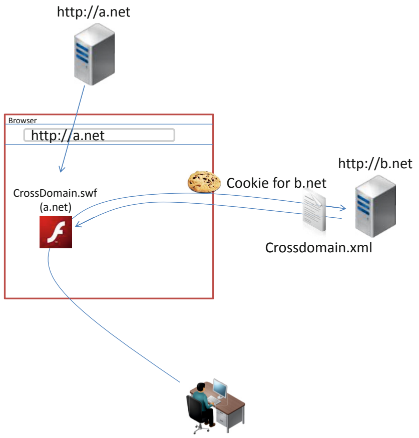


Fig. 1. General Use Case

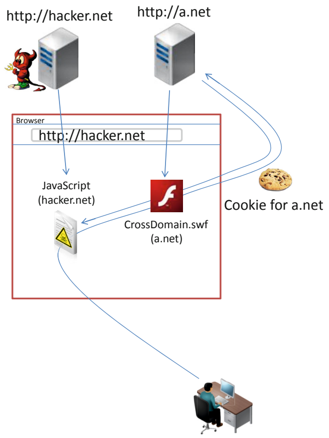


Fig. 2. Vulnerable Flash Proxy

with the user's session cookie attached to it. Thus, an adversary is able to craft an applet file that can access the personalized, access-controlled data on `b.net`. The last step for the attacker is to lure users into visiting a website that embeds the malicious file. This could either be achieved through social networks, social engineering or through the abuse of other vulnerabilities such as cross-site scripting on vulnerable sites. The more popular the website hosting the insecure policy is, the more chances the attacker has that the users who end up visiting the malicious domain will provide him with authenticated sessions.

3.2 Vulnerable Scenario 2: Insecure Flash Proxies

As we have recently shown [10], an insecure cross-domain policy is not the only condition which enables adversaries to conduct the attacks outlined in Section 3.1: The second misuse case results from improper use of Flash or Silverlight applets. As stated in Section 2.4, an applet is able to exchange data with JavaScript for further processing. For security reasons, communication between JavaScript and Flash/Silverlight applets is also restricted to the same domain. The reason for this is that, as opposed to other embedded content such as JavaScript, embedded Flash files keep their origin. Consequently, JavaScript located on `a.net` cannot communicate with an applet served by `b.net` even if that is embedded in `a.net`. But, as cross-domain communication is also sometimes desirable in this setup, an applet file can explicitly offer communication capabilities to JavaScript served by a remote domain. Therefore, Flash utilizes a white-listing approach by offering the ActionScript directive `System.security.allowDomain(domain)`. With this directive, an applet file can explicitly allow cross-domain communication from a certain domain or white-list all domains by using a wildcard.

We have shown that these wildcards are also misused in practice: Several popular off-the-shelf cross-domain Flash proxies include such wildcard directives, and thus, allow uncontrolled cross-domain JavaScript-to-Flash communication. If such a Flash applet offers cross-domain network capabilities and at the same time grants control over these capabilities to cross-domain JavaScript, an attacker can conduct requests in the name of the website serving the applet file.

Figure 2 shows the general setup for this attack. An adversary sets-up a website `hacker.net` that includes JavaScript capable of communicating with a Flash applet served by `a.net`. This applet includes a directive that allows communication from JavaScript served by any other domain. Thus, the attacker is able to instruct the Flash applet to conduct arbitrary requests in the name of `a.net`. If JavaScript from `hacker.net` now conducts a request towards `a.net` via the vulnerable applet, the request itself is not happening cross-domain as `a.net` is the sender as well as the receiver. Therefore, the Flash Player will grant any request without even checking if there is a cross-domain policy in place at `a.net`. Consequently, the attacker can conduct cross-domain requests and read the response as if `a.net` would host a wildcard cross-domain policy. Furthermore, the adversary is also able to misuse existing trust relationships of other domains towards `a.net`. So, if other domains white-list `a.net` in their cross-domain policy, the attacker can also conduct arbitrary cross-domain requests towards those websites by tunneling them through the vulnerable proxy located on `a.net` (please refer to [10] for details concerning this class of attacks).

3.3 Resulting Malicious Capabilities

Based on the presented use and misuse cases we can deduce the following malicious capabilities that an attacker is able to gain.

1. *Leakage of Sensitive Information:* As an adversary is able to conduct arbitrary requests towards a vulnerable website and read the corresponding responses, he is able to leak any information that is accessible via the HTML of that site including information that is bound to the user's session id. Thus, an attacker is able to steal sensitive and private information [8].
2. *Circumvention of Cross-Site Request Forgery Protection:* In order to protect Web applications from cross-site request forgery attacks, many websites utilize a nonce-based approach [4] in which a random and unguessable nonce is included into every form of a Web page. A state changing request towards a website is only granted if a user has requested the form before and included the nonce into the state changing request. The main security assumption of such an approach is that no one else other than the user is able to access the nonce and thus, nobody else is able to conduct state changing requests. As client-side cross-domain requests allow an adversary to read the response of a request, an attacker is able to extract the secret nonce and thus bypass CSRF protections.
3. *Session Hijacking:* Given the fact that an adversary is able to initiate HTTP requests carrying the victim's authentication credentials, he is essentially

able to conduct a session hijacking attack (similar to the one performed through XSS vulnerabilities). As long as the victim remains on the Web page embedding the malicious applet, the attacker can chain a series of HTTP requests to execute complex actions on any vulnerable Web application under the victim's identity. The credentials can be used by an attacker either in an automated fashion (e.g. a standard set of requests towards vulnerable targets) or interactively, by turning the victim in an unwilling proxy and browsing vulnerable Web applications under the victim's IP address and credentials (see Section 6.1).

3.4 General Risk Assessment

Since the first study on the usage of cross-domain policies conducted by Jeremiah Grossman in 2006 [7], the implementation of cross-domain policies for Flash and Silverlight applets is becoming more and more popular. While Grossman repeated his experiment in 2008 and detected cross-domain policies at 26% of the top 500 websites, the latest experiments show that the adoption of policies for the same set of websites has risen to 52% [14]. Furthermore, the amount of wildcard policies rose from 7% in 2008 up to 11% in 2011. Those figures clearly show that client-side cross-domain requests are of growing importance.

Three recent studies [9,13,14] investigated the security implications of cross-domain policies deployed in the wild and all came to the conclusion that cross-domain mechanisms are widely misused. Among the various experiments, one of the studies [14] investigated the Alexa top one million websites and found 82,052 Flash policies, from which 15,060 were found using wildcard policies in combination with authentication tracking and, thus, vulnerable to the range of attacks presented in Section 3.

4 Real-World Vulnerabilities

To provide a practical perspective on the topic matter, we present in this Section two previously undocumented, real-world cases that show the vulnerabilities and the corresponding malicious capabilities. These two websites are only two examples of thousands of vulnerable targets. However, the popularity and the large user base of these two websites show that even high profile sites are not always aware of the risks imposed by the insecure usage of client-side cross-domain functionality.

4.1 Deal-of-the-Day Website: Insecure Wildcard Policy[2]

The vulnerable website features daily deals to about 70 million users world-wide. At the time of this writing, it was ranked on position 309 of the Alexa Top Sites. When we started investigating cross-domain security issues on the website, a

[2] Anonymized for publication.

crossdomain.xml file was[3] present, which granted any site in the WWW arbitrary cross-domain communication privileges (see Listing 2). This policy can be seen as a worst case example as it renders void all restrictions implied by the Same-Origin Policy and any CSRF protection. On the same domain under which the policy was served, personal user profiles and deal registration mechanisms were available. Hence, an attacker was able to steal any information provided via the HTML user interface. As a proof-of-concept we implemented and tested an exploit which was able to extract any personal information[4]. Furthermore, it was possible to register a user for any deal on the website as CSRF tokens included into every form of the website could be extracted by a malicious Flash or Silverlight applet.

Listing 2. The website's crossdomain.xml file

```
<cross-domain-policy>
    <site-control permitted-cross-domain-policies="all" />
    <allow-access-from domain="*" />
    <allow-http-request-headers-from domain="*" headers="*" />
</cross-domain-policy>
```

4.2 Popular Sportswear Manufacturer: Vulnerable Flash Proxy[5]

As discussed in Section 3, even without a wildcard cross-domain policy, an attacker is able to conduct arbitrary cross-domain requests under certain circumstances. For this to be possible, a website needs to host a Flash or Silverlight file which is vulnerable to the second misuse case presented in Section 3.2.

We found such a vulnerable flash proxy on a Web site of a popular sportswear manufacturer that offers an online store for its products. Although the website's cross-domain policy only includes non-wildcard entries, it hosts a vulnerable Flash proxy which can be misused to circumvent the restrictions implied by the Same-Origin Policy.

Besides leaking private data and circumventing CSRF protections, the vulnerability can be exploited even further by an attacker to misuse existing trust relationships of the sportswear manufacturer with other websites. As the vulnerable Flash proxy enables an adversary to conduct client-side cross-domain requests in the name of the company, other websites which white-list the sportswear manufacturer's domain in their cross-domain policies are also exposed to attacks. During our tests, we found 8 other websites containing such a white-list entry.

[3] The vulnerability has been reported and fixed in the meantime.

[4] Notice: We only extracted our own personal information and, hence, did not attack any third person.

[5] Anonymized for Publication.

5 Client-Side Detection and Mitigation of Malicious Cross-Domain Requests

In Section 3.4 we showed that plug-in-based cross-domain techniques are widely used in an insecure fashion and thus users are constantly exposed to risks resulting from improper configuration of cross-domain mechanisms (see Section 3 for details). In order to safeguard end-users from these risks we propose *DEMACRO*, a client-side protection mechanism which is able to detect and mitigate malicious plug-in-based cross-domain requests.

5.1 High-Level Overview

The general mechanism of our approach functions as follows: The tool observes every request that is created within the user's browser. If a request targets a cross-domain resource and is caused by a plugin-based applet, the tool checks whether the request could potentially be insecure. This is done by examining the request's execution context to detect the two misuse cases presented in Section 3: For one, the corresponding cross-domain policy is retrieved and checked for insecure wildcards. Furthermore, the causing applet is examined, if it exposes client-side proxy functionality. If one of these conditions is met, the mechanism removes all authentication information contained in the request. This way, the tool robustly protects the user against insecurely configured cross-domain mechanisms. Furthermore, as the request itself is not blocked, there is only little risk of breaking legitimate functionality.

While our system can, in principle, be implemented in all modern browsers, we chose to implement our prototype as a Mozilla Firefox extension and thus the implementation details, wherever these are present, are specific to Firefox's APIs.

5.2 Disarming Potentially Malicious Cross-Domain Requests

A cross-domain request conducted by a plug-in is not necessarily malicious as there are a lot of legitimate use cases for client-side cross-domain requests. In order to avoid breaking the intended functionality but still protecting users from attacks, it is crucial to eliminate malicious requests while permitting legitimate ones. As described in Section 3.1 the most vulnerable websites are those that make use of a wildcard policy and host access-controlled, personalized data on the same domain; a practice that is strongly discouraged by Adobe [2]. Hence, we regard this practice as an anti-pattern that carelessly exposes users to high risks. Therefore, we define a potentially malicious request as one that carries access credentials in the form of session cookies or HTTP authentication headers towards a domain that serves a wildcard policy. When the extension detects such a request, it disarms it by stripping session cookies and authentication headers. As the actual request is not blocked, the extension does not break legitimate application but only avoids personalized data to appear in the response.

Furthermore, *DEMACRO* is able to detect attacks against vulnerable Flash proxies as presented in Section 3.2. If a page on `a.net` embeds an applet file served by `b.net` and conducts a same-domain request towards `b.net` user credentials are also stripped by our extension. The rationale here is that a Flash-proxy would be deployed on a website so that the website itself can use it rather than allowing any third party domain to embed it and use it.

5.3 Detailed Detection Algorithm

While *DEMACRO* is active within the browser it observes any request that occurs. Before applying actual detection and mitigation techniques, *DEMACRO* conducts pre-filtering to tell plugin- and non-plugin-based requests apart

. If a plugin-based request is observed, *DEMACRO* needs to check whether the request was caused by a Silverlight or a Flash Applet, in order to download the corresponding cross-domain policy file

. With the information in the policy file *DEMACRO* is now able to reveal the nature of a request by assessing the following values:

1. `Embedding Domain:` The domain that serves the HTML document which embeds the Flash or Silverlight file
2. `Origin Domain:` The domain that serves the Silverlight or Flash file and is thus used by the corresponding plug-in as the origin of the request
3. `Target Domain:` The domain that serves the cross-domain policy and is the target for the request
4. `Cross-domain whitelist:` The list of domains (including wildcard entries) that are allowed to send cross-domain requests to the target domain. This information is received either from the Silverlight or Flash cross-domain policy.

Depending on the scenario, the three domains (1,2,3) can either be totally distinct, identical or anything in between. By comparing these values *DEMACRO* is able to detect if a request was conducted across domain boundaries or if a vulnerable proxy situation is present. For the former, the extension additionally checks whether the policy includes a wildcard. If such a potentially malicious situation is detected the extension removes existing HTTP authentication headers or session cookies from the request. Figure 3 summarizes our detection algorithm.

In the remainder of this section, we provide technical details how *DEMACRO* handles the tasks of request interception, plugin identification, and session identifier detection.

Requests Interception and Redirect Tracing: In order to identify plug-in-based requests, *DEMACRO* has to examine each request at several points in time. Firefox offers several different possibilities to intercept HTTP requests, but none of them alone is sufficient for our purpose. Therefore, we leveraged the capabilities of the `nsIContentPolicy` and the `nsIObserver` interfaces.

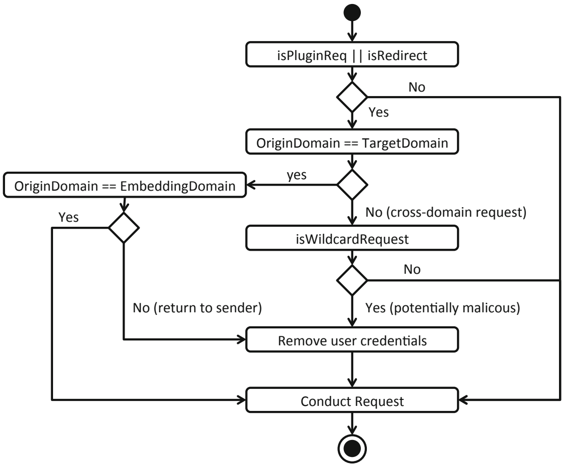

isPluginReq || isRedirect
No
Yes
OriginDomain == TargetDomain
yes
OriginDomain == EmbeddingDomain
Yes
No (cross-domain request)
isWildcardRequest
No
No (return to sender)
Yes (potentially malicous)
Remove user credentials
Conduct Request

Fig. 3. Detection and Mitigation Algorithm

The `nsIContentPolicy` interface offers a method called `shouldLoad` which is called each time an HTTP request is initiated and before the actual `HTTPChannel` object is created[6]. Thereby, the method returns a boolean value indicating whether a request should be conducted by Firefox or not. Since we do not want to block a request but only modify its header fields, this method cannot fully serve our purpose. But as it is the only method that receives the url of the webpage and the DOM object that caused the request, we need to intercept page requests here and detect the origin of a request. A request originating from either a `HTMLObjectElement` or from a `HTMLEmbedElement` is categorized as a plug-in-based request.

The `nsIObserver` interface offers the `observe` method which is called at three different points in time:

1. `http-on-modify-request`: Called each time before an HTTP request is sent.
2. `http-on-examine-response`: Called each time before the response is passed back to the caller.
3. `http-on-examine-cache-response`: Called instead of `http-on-examine-response` when the response is completely read from cache.

Thereby, the `observe` method receives an HTTPChannel object as a parameter which can be used to modify request as well as response header fields. If the extension detects a potentially malicious request, it can thus disarm it by stripping existing session information in `Cookie` fields and by removing `Authentication` header fields.

To prevent an attacker from hiding cross-domain requests behind local redirects, the extension also needs to keep track of any redirect resulting from a

[6] The HTTPChannel object is used to conduct the request and read the response.

plug-in-based request. This is also done in the `observe` method at the `http-on-examine-response` event. If a 3xx status code of a plug-in-based request is detected, the redirect location will be stored for examination of follow-up requests.

During experimentation with *DEMACRO* we noticed that Web applications tend to initiate a new session if an existing session identifier is not present in a user's cookie headers. More precisely, if a session identifier never reaches the application, the application emits a `Set-Cookie` header which includes a new session identifier. If this header reaches the user's browser it will override existing cookies with the same name for the corresponding domain and therefore the user's authenticated session is replaced by an unauthenticated one. As this can obviously lead to undesired side-effects and possible denial of service attacks, *DEMACRO* additionally examines each response of potentially malicious requests and removes `Set-Cookie` headers before allowing the response to be interpreted by the browser.

Plug-In Identification: In order for *DEMACRO* to investigate the correct cross-domain policy, our system must detect whether the underlying request was caused by a Silverlight or by a Flash applet. Since the HTTP request itself does not carry any information about its caller, we developed a mechanism for Firefox to distinguish between Flash and Silverlight requests.

As stated above, the only point in time where we have access to the request-causing DOM element is the call of the `shouldLoad` method in the `nsIContentPolicy` interface. But, due to the fact that Silverlight and Flash files can both be embedded into a page by using either an `HTMLObjectElement` or an `HTMLEmbedElement`, we need to examine the exact syntax used to embed those files for each element. By testing standard and less-standard ways of embedding an applet to a page, we resulted to the detection mechanism shown in Listing 3. In case the detection mechanism fails, the extension simply requests both policies, in order to prevent an attacker who is trying to circumvent our extension by using an obscure method to embed his malicious files.

Table 1. Default session naming for the most common Web frameworks

Web Framework	Name of Session variable
PHP	phpsessid
ASP/ASP.NET	asp.net_sessionid aspsessionid
JSP	x-jspsessionid jsessionid

Session-Cookie Detection: As described earlier, it is necessary to differentiate between session information and non-session information and strip the former while preserving the latter. The reasoning behind this decision is that while transmitting session identifiers over applet-originating cross-domain requests can

Listing 3. Object and Embed detection (pseudocode)

```
function detectPlugin(HTMLElement elem){

  var type = elem.getAttribute("type");
  var data = elem.getAttribute("data");
  var src  = elem.getAttribute("src");

  switch(type.startsWith){
    case "application/x-silverlight": return flash;
    case "application/x-shockwave-flash": return silverlight;
    default:
  }

  if(data=="data:application/x-silverlight")
    return silverlight;
  if(data.endsWith(".swf")) return flash;

  switch(src.endsWith){
    case ".swf": return flash;
    case ".xap": return silverlight;
    default:
  }

  return -1;
}
```

lead to attacks against users, non-session values should be allowed to be transmitted since they can be part of a legitimate Web application's logic.

DEMACRO utilizes a techniques initially described by Nikiforakis et al. [16] and Tang et al. [23] that attempts to identify session identifiers at the client-side. The approach consists of two pillars. The first one is based on a dictionary check and the second one on measuring the information entropy of a cookie value:

The dictionary check is founded on the observation that well known Web frameworks use well-defined names for session cookies - see Table 1. By recognizing these values we are able to unambiguously classify such cookies as session identifiers. Furthermore, in order to detect custom naming of session identifiers, we characterize a value as a session cookie if it's name contains the string "sess" and if the value itself includes letters as well as numbers and is more than ten characters long. We believe that this is a reasonable assumption since all the session identifiers generated by the aforementioned frameworks fall within this categorization.

The second pillar is based on the fact that session identifiers are long random strings. Thus their entropy, i.e., the number of bits necessary to represent them, is by nature higher than non-random strings. *DEMACRO* first compares a cookie variable's name with its dictionary and if the values are not located it then calculates the entropy of the variable's value. If the result exceeds a certain threshold (acquired by observing the resulting entropy of session identifiers generated by the PHP programming framework), the value is characterized as a session identifier and is removed from the outgoing request.

This session identifier technique works without the assistance of Web servers and offers excellent detection capabilities with a false negatives rate of ~3% and a false-positive ratio of ~0.8% [16].

6 Evaluation

6.1 Security

In this section we present a security evaluation of *DEMACRO*. In order to test its effectiveness, we used it against MalaRIA [15], a malicious Flash/Silverlight exploit tool that conducts Man-In-The-Middle attacks by having a user visit a malicious website. MalaRIA tunnels attacker's requests through the victim's browser thus making cross-domain requests through the victim's IP address and with the victim's cookies. We chose Joomla, a popular Content Management System, as our victim application mainly due to Joomla's high market penetration [5]. Joomla was installed on a host with a wild-card cross-domain policy, allowing all other domains to communicate with it through cross-domain requests.

Our victim logged in to Joomla and then visited the attacker's site which launched the malicious proxy. The attacker, situated at a different browser, could now initiate arbitrary cross-domain requests to our Joomla installation. Without our countermeasure, the victim's browser added the victim's session cookie to the outgoing requests, thus authenticating the attacker as the logged-in user. We repeated the experiment with *DEMACRO* activated. This time, the plug-in detected the cross-domain requests and since the Joomla-hosting domain implemented a weak cross-domain policy, it stripped the session-identifier before forwarding the requests. This means that while the attacker could still browse the Joomla website through the victim's browser, he was no longer identified as the logged-in user.

Apart from the security evaluation with existing and publicly available exploits, we also implemented several test-cases of our own. We implemented Flash and Silverlight applets to test our system against all possible ways of conducting cross-domain requests across the two platforms and we also implemented several vulnerable Flash and Silverlight applets to test for the second misuse case (Section 3.2). In all cases, *DEMACRO* detected the malicious cross-domain requests and removed the authentication information. Lastly we tested our system against the exploits we developed for the real-world use cases (See Section 4) and were able to successfully prevent the attacks in both cases.

6.2 Compatibility

In order to test *DEMACRO*'s practical ability to stop potentially malicious cross-domain requests while preserving normal functionality, we conducted a survey of

Table 2. Nature of requests observed by DEMACRO for Alexa Top 1k websites

Request Type	**#Requests**
Non-Cross-Domain	77,988 (98.6%)
Safe Cross-Domain	414 (0.52%)
Unsafe Cross-Domain	
Without Cookies	387 (0.49%)
With Session Cookies	275 (0.34%)
With Non-Session Cookies	29 (0.05%)
Total	79,093 (100%)

the Alexa top 1,000 websites. We used the Selenium IDE [7] to instrument Firefox to automatically visit these sites twice. The rationale behind the two runs is the following: In the first run, *DEMACRO* was deactivated and the sites and ad banners were populating cookies to our browser. In the second run, *DEMACRO* was enabled and reacting to all the insecure cross-domain requests by stripping-off their session cookies that were placed in the browser during the first run. The results of the second run are summarized in Table 2.

In total, we were able to observe 79,093 HTTP requests, of which 1,105 were conducted by plug-ins across domain boundaries. 691 of the requests were considered insecure by *DEMACRO* and thus our system deemed it necessary to remove any session cookies found in these requests. Of the 691, approximately half of them did not contain cookies thus these requests were not modified. For the rest, *DEMACRO* identified at least one session-like value in 275 requests which it removed before allowing the requests to proceed.

In order to find out more about the nature of the insecure requests that *DEMACRO* modified, we further investigated their intended usage: The 275 requests were conducted by a total of 68 domains. We inspected the domains manually and discovered that almost half of the requests where performed by Flash advertising banners and the rest by video players, image galleries and other generic flash applications. We viewed the websites first with *DEMACRO* de-activated and then activated and we noticed that in all but one cases, the applications were loading correctly and displaying the expected content. The one case that did not work, was a Flash advertisement that was no-longer functional when session cookies were stripped away from its requests.

One can make many observations based on the aforementioned results. First of all, we observe that the vast majority of requests do not originate from plugins which experimentally verifies the commonly-held belief that most of the Web's content is served over non-plugin technologies. Another interesting observation is that 50% of the cross-domain plugin-originating requests are towards hosts that implement, purposefully or accidentally, weak cross-domain policies. Finally, we believe that the experiments show that *DEMACRO* can protect against cross-domain attacks without negatively affecting, neither the user's browsing experience nor a website's legitimate content.

[7] http://seleniumhq.org/projects/ide/

Table 3. Best and worst-case microbenchmarks (in seconds) of cross-domain requests

1,500 C.D. requests	**Firefox**	**FF & DEMACRO**	**Overhead/req.**
JavaScript	27.107	28.335	0.00082
Flash	184	210	0.00173

6.3 Performance

Regardless of the benefits of a security solution, if the overhead that its use imposes is too large, many users will avoid deploying it. In order to evaluate the performance of our countermeasure we measured the time needed to perform a large number of cross-domain requests when a) issued by JavaScript and b) issued by a Flash applet.

JavaScript Cross-Domain Requests: This experiment presents the minimum overhead that our extension will add to a user's browser. It consists of an HTML page which includes JavaScript code to fetch 1,500 images from a different domain than the one the page is hosted on. Both domains as well as the browsing user are situated on the same local network to avoid unpredictable network inconsistencies. The requests originating from JavaScript, while cross-domain, are not part of the attack surface explored in this paper and are thus not inspected by our system. The experiment was repeated 5 times and the first row of Table 3 reports the time needed to fetch all 1,500 images with and without our protecting system. The overhead that our system imposes is 0.00082 seconds for each cross-domain request. While this represents the best-case scenario, since none of the requests need to be checked against weak cross-domain policies, we believe that this is very close the user's actual everyday experience where most of the content served is done so over non-plugins and without crossing domain boundaries, as shown in Section 6.2.

Flash Cross-Domain Requests: In this experiment we measure the worst-case scenario where all requests are cross-domain Flash-originating and thus need to be checked and processed by our system. We chose to measure "Flash-Gallery" [8], a Flash-based image gallery that constructs its albums either from images on the local disk of the webserver or using the public images of a given user on `Flickr.com`. Cross-domain accesses occur in the latter case in order for the applet to fetch the necessary information of each image's location and finally the image itself. A feature that made us choose this applet over other Flash-based image galleries is its pre-emptive loading of all available images before the user requests them. Thus, the applet will perform all cross-domain requests needed without any user interaction.

To avoid the network inconsistencies of actually fetching 500 images from Flickr, we implemented the necessary subset of Flickr's protocol to successfully provide a list of image URIs to the Flash applet, in our own Web application

[8] `http://www.flash-gallery.org/`

which we hosted on our local network. Using our own DNS server, we resolved `Flickr.com` to the host of our Web application instead of the actual Web service. This setup, allowed us to avoid unnecessary modifications on the client-side, i.e. the Flash platform, our plug-in and the Flash applet, and to accurately measure the imposed worst-case overhead of our solution. According to the current protocol of `Flickr.com`, an application first receives a large list of image identifiers. For each identifier, the applet needs to perform 3 cross-domain requests. One to receive information about the image, one to fetch the URIs of different image sizes and finally one to fetch the image itself. We configured our Web service to return 500 image identifiers which in total correspond to 1,500 cross-domain requests. Each transfered image had an average size of 128 Kilobytes. Each experiment was repeated 5 times and we report the average timings.

The second row of Table 3 reports the results of our experiment. Without any protection mechanisms, the browser fetched and rendered all images in 184 seconds. The reason that made these requests so much slower than the non-protected JavaScript requests of Section 6.3 is that this time, the images are loaded into the Flash-plugin and rendered, as part of a functioning interactive image gallery on the user's screen. With our system activated, the same task was accomplished in 210 seconds, adding a 0.00173 seconds overhead to each plugin-based cross-domain request in order to inspect its origin, the policy of the remote-server and finally perform any necessary stripping of credentials.

It is necessary to point out that this overhead represents the upper-bound of overhead that a user will witness in his every-day browsing. In normal circumstances, the majority of requests are not initiated by Flash or Silverlight and thus we believe that the actual overhead will be closer to the one reported in the previous section. Additionally, since our experiments were conducted on the local network, any delay that *DEMACRO* imposes affects the total operation time much more than requests towards remote Web servers where the round-trip time of each request will be significantly larger.

7 Related Work

One of the first studies that gave attention to insecure cross-domain policies for Flash, was conducted by Grossman in 2006 [7]. At the time, 36% of the Alexa top 100 websites had a cross-domain policy and 6% of them were using insecure wildcards. Kontaxis et al. [13] recently reported that now more than 60% of the same set of websites implement a cross-domain policy and the percentage of insecure wildcard policies has increased to 21%. While we [14] used a more conservative definition of insecure policies, we also came to the conclusion that the cross-domain traffic through Flash and Silverlight is a real problem.

To the best of our knowledge this paper presents the first countermeasure towards this increasingly popular problem. The nature of the problem, i.e. server-side misconfigurations resulting to poor security, allows for two categories of approaches. The first approach is at the server-side, where the administrator of a domain configures the cross-domain policy correctly and thus eliminates the problem all together. While this is the best solution, it a) depends on an administrator to realize the problem and implement a secure policy and b) needs to be repeated by all administrators in all the domains that use cross-domain policies. Practice has shown that adoption of server-side countermeasures can be a lengthy and often incomplete process [27]. For these reasons we decided to focus our attention on the client-side where our system will protect the user regardless of the security provisions of any given site.

Pure client-side security countermeasures against popular Web application attacks have in general received much attention due to their attractive "install once, secure all" nature. Kirda et al. [12] attempt to stop session hijacking attacks conducted through cross-site scripting (XSS) [26] at the client side using a proxy which blocks requests towards dynamically generated URIs leading to third-party domains. Nikiforakis et al. [16] and Tang et al. [23] tackle the same problem through the identification of session identifiers at the client-side and their subsequent separation from the scripts running in the browser. Vogt et al. [24] also attempt to prevent the leakage of session identifiers through the use of static analysis and dynamic data tainting, however Russo et al. [20] have shown that the identifiers can still be leaked through the use of side channels.

Moving on to Cross-Site Request Forgeries, Johns and Winter [11] propose a solution where a client-side proxy adds tokens in incoming URLs (based on their domains) that bind each URL with their originating domain. At each outgoing request, the domain of the request is checked against the originating domain and if they don't match, the requests are stripped from their credentials. De Ryck et al. [21] extend this system, by moving it into the browser where more context-information is available. Shahriar and Zulkernine [22] propose a detection technique where each cross-domain request is checked against the visibility of the code that originated it in the user's browser. According to the authors, legitimate requests will have originated from visible blocks of code (such as a visible HTML form) instead of hidden code (an invisible auto-submitting form or JavaScript code). None of the above authors consider cross-domain requests generated by Flash and Silverlight.

Client-side defense mechanisms have also been used to protect a user's online privacy. Egele et al. [6] designed a client-side proxy which allows users to make explicit decisions as to which personal information gets transmitted to third-party social network applications. Beato et al. propose a client-side access-control system for social networks, where the publishing user can select who will get access to the published information [3].

8 Conclusion

In this paper we have shown that the increasingly popular problem of insecure Flash/Silverlight cross-domain policies is not just an academic problem, but a real one. Even high profile sites carelessly expose their users to unnecessary risks by relying on misconfigured policies and plugin applets. In order to protect security aware users from malicious cross-domain requests we propose a client-side detection and prevention mechanism, *DEMACRO*. *DEMACRO* observes all requests that occur within the user's web browser and checks for potential malicious intent. In this context, we consider a request to be potentially harmful, if it targets a cross-domain resource on a Web server that deploys an insecure wildcard policy. In such a case, *DEMACRO* disarms potentially insecure cross-domain requests by stripping existing authentication credentials. Furthermore, *DEMACRO* is able to prevent the vulnerable proxy attack in which a vulnerable Flash application is misused to conduct cross-domain requests under a foreign identity. We examine the practicality of our approach, by implementing and evaluating *DEMACRO* as a Firefox extension. The results of our evaluation suggest that our system is able to protect against malicious cross-domain requests with a negligible performance overhead while preserving legitimate functionality.

Acknowledgments. This research was done with the financial support from the Prevention against Crime Programme of the European Union, the IBBT, the Research Fund KU Leuven, and the EU-funded FP7 projects NESSoS and WebSand.

References

1. Adobe. Adobe - security bulletins and advisories
2. Adobe Systems Inc. Cross-domain policy file specification (January 2010), http://www.adobe.com/devnet/articles/crossdomain_policy_file_spec.html
3. Beato, F., Kohlweiss, M., Wouters, K.: Scramble! Your Social Network Data. In: Fischer-Hübner, S., Hopper, N. (eds.) PETS 2011. LNCS, vol. 6794, pp. 211–225. Springer, Heidelberg (2011)
4. Burns, J.: Cross Site Request Forgery - An introduction to a common web application weakness. Whitepaper (2005), https://www.isecpartners.com/documents/XSRF_Paper.pdf
5. Water and Stone: Open Source CMS Market Share Report (2010)
6. Egele, M., Moser, A., Kruegel, C., Kirda, E.: Pox: Protecting users from malicious facebook applications. In: Proceedings of the 3rd IEEE International Workshop on Security in Social Networks (SESOC), pp. 288–294 (2011)
7. Grossman, J.: crossdomain.xml statistics, http://jeremiahgrossman.blogspot.com/2006/10/crossdomainxml-statistics.html
8. Grossman, J.: I used to know what you watched, on YouTube (September 2008), http://jeremiahgrossman.blogspot.com/2008/09/i-used-to-know-what-you-watched-on.html (accessed in January 2011)
9. Jang, D., Venkataraman, A., Swaka, G.M., Shacham, H.: Analyzing the Cross-domain Policies of Flash Applications. In: Proceedings of the 5th Workshop on Web 2.0 Security and Privacy, W2SP (2011)
10. Johns, M., Lekies, S.: Biting the Hand That Serves You: A Closer Look at Client-Side Flash Proxies for Cross-Domain Requests. In: Holz, T., Bos, H. (eds.) DIMVA 2011. LNCS, vol. 6739, pp. 85–103. Springer, Heidelberg (2011)
11. Johns, M., Winter, J.: RequestRodeo: Client Side Protection against Session Riding. In: Proceedings of the OWASP Europe 2006 Conference (2006)
12. Kirda, E., Kruegel, C., Vigna, G., Jovanovic, N.: Noxes: A Client-Side Solution for Mitigating Cross Site Scripting Attacks. In: Security Track of the 21st ACM Symposium on Applied Computing (SAC) (April 2006)

13. Kontaxis, G., Antoniades, D., Polakis, I., Markatos, E.P.: An empirical study on the security of cross-domain policies in rich internet applications. In: Proceedings of the 4th European Workshop on Systems Security, EUROSEC (2011)
14. Lekies, S., Johns, M., Tighzert, W.: The state of the cross-domain nation. In: Proceedings of the 5th Workshop on Web 2.0 Security and Privacy, W2SP (2011)
15. Malaria - i'm in your browser, surfin your webs (2010), http://erlend.oftedal.no/blog/?blogid=107
16. Nikiforakis, N., Meert, W., Younan, Y., Johns, M., Joosen, W.: SessionShield: Lightweight Protection against Session Hijacking. In: Erlingsson, Ú., Wieringa, R., Zannone, N. (eds.) ESSoS 2011. LNCS, vol. 6542, pp. 87–100. Springer, Heidelberg (2011)
17. Rich internet application (ria) market share, http://www.statowl.com/custom_ria_market_penetration.php
18. Rios, B.B.: Cross domain hole caused by google docs, http://xs-sniper.com/blog/Google-Docs-Cross-Domain-Hole/
19. Ruderman, J.: The Same Origin Policy (August 2001), http://www.mozilla.org/projects/security/components/same-origin.html (October 01, 2006)
20. Russo, A., Sabelfeld, A., Chudnov, A.: Tracking Information Flow in Dynamic Tree Structures. In: Backes, M., Ning, P. (eds.) ESORICS 2009. LNCS, vol. 5789, pp. 86–103. Springer, Heidelberg (2009)
21. De Ryck, P., Desmet, L., Heyman, T., Piessens, F., Joosen, W.: CsFire: Transparent Client-Side Mitigation of Malicious Cross-Domain Requests. In: Massacci, F., Wallach, D., Zannone, N. (eds.) ESSoS 2010. LNCS, vol. 5965, pp. 18–34. Springer, Heidelberg (2010)
22. Shahriar, H., Zulkernine, M.: Client-side detection of cross-site request forgery attacks. In: 2010 IEEE 21st International Symposium on Software Reliability Engineering (ISSRE), pp. 358–367 (2010)
23. Tang, S., Dautenhahn, N., King, S.T.: Fortifying web-based applications automatically. In: Proceedings of the 8th ACM Conference on Computer and Communications Security (2011)
24. Vogt, P., Nentwich, F., Jovanovic, N., Kruegel, C., Kirda, E., Vigna, G.: Cross Site Scripting Prevention with Dynamic Data Tainting and Static Analysis. In: Proceedings of the 14th Annual Network and Distributed System Security Symposium, NDSS 2007 (2007)
25. W3C. Cross-Origin Resource Sharing, http://www.w3.org/TR/cors/
26. The Cross-site Scripting FAQ, http://www.cgisecurity.com/xss-faq.html
27. Zhou, Y., Evans, D.: Why Aren't HTTP-only Cookies More Widely Deployed? In: Proceedings of 4th Web 2.0 Security and Privacy Workshop, W2SP 2010 (2010)

FlashDetect: ActionScript 3 Malware Detection

Timon Van Overveldt[1], Christopher Kruegel[2,3], and Giovanni Vigna[2,3]

[1] Katholieke Universiteit Leuven, Belgium
timon.vanoverveldt@student.kuleuven.be
[2] University of California, Santa Barbara, USA
{chris,vigna}@cs.ucsb.edu
[3] Lastline, Inc.

Abstract. Adobe Flash is present on nearly every PC, and it is increasingly being targeted by malware authors. Despite this, research into methods for detecting malicious Flash files has been limited. Similarly, there is very little documentation available about the techniques commonly used by Flash malware. Instead, most research has focused on JavaScript malware.

This paper discusses common techniques such as heap spraying, JIT spraying, and type confusion exploitation in the context of Flash malware. Where applicable, these techniques are compared to those used in malicious JavaScript. Subsequently, FlashDetect is presented, an offline Flash file analyzer that uses both dynamic and static analysis, and that can detect malicious Flash files using ActionScript 3. FlashDetect classifies submitted files using a naive Bayesian classifier based on a set of predefined features. Our experiments show that FlashDetect has high classification accuracy, and that its efficacy is comparable with that of commercial anti-virus products.

Keywords: Flash exploit analysis, malicious ActionScript 3 detection, Flash type confusion.

1 Introduction

Adobe Flash is a technology that provides advanced video playback and animation capabilities to developers through an advanced scripting language. The files played by Flash, called SWFs, are often embedded into webpages to be played by a browser plugin, or are embedded into a PDF file to be played by a copy of the Flash Player included in Adobe's Acrobat Reader. The technology is nearly ubiquitous on the desktop: over 99% of all PC users have the Flash browser plugin installed, according to Adobe [1]. However, over the last couple of years, the Flash Player has increasingly become the target of exploitation [18,23,10], with at least 134 high-severity vulnerabilities that have been identified in the Flash Player since January 2009 [15].

Since version 9 appeared in 2006, the Flash Player has supported two scripting languages, ActionScript 2 (AS2) and ActionScript 3 (AS3), each with its own virtual machine. Traditionally, exploits have targeted the older AS2 virtual

D. Balzarotti, S. Stolfo, and M. Cova (Eds.): RAID 2012, LNCS 7462, pp. 274–293, 2012.

machine. However, a number of critical vulnerabilities discovered in the AS3 virtual machine have resulted in an ever growing number of exploits targeting this virtual machine. Even exploits targeting the AS2 virtual machine increasingly turn to AS3 to perform part of the attack. For example, a heap spray might be performed in AS3 before running an AS2 script that exploits an old vulnerability.

Despite the increasing importance of successful solutions to Flash exploit detection, academic research describing such solutions has been scarce. A lot of research has instead focused on JavaScript malware detection. However, without a sound detector for Flash malware, even the most advanced JavaScript malware detector could be circumvented by performing all or part of the attack in Flash.

In this paper, we present FlashDetect, an offline Flash file analyzer and malware detector. FlashDetect combines static bytecode analysis with dynamic analysis using an instrumented version of the Lightspark flash player [19] to enable a high detection rate while maintaining a low false positive rate. The analysis of a Flash file is based on a set of simple yet effective predefined features. These features are used to classify a Flash file using a combination of a naive Bayesian classifier and a single vulnerability-specific filter. FlashDetect is an evolution of OdoSwiff, presented by Ford *et al.* in [6]. However, in contrast to OdoSwiff, FlashDetect focuses solely on the analysis Flash files using AS3, while OdoSwiff mainly covered AS2 exploits. Given the significant differences between AS2 and AS3, we had to develop an entire new set of features and detection techniques. Additionally, OdoSwiff did not employ a naive Bayesian classifier, instead relying solely on threshold-based filters.

The contributions made by this paper include:

- **Insight into Common Flash Exploit Techniques.**
  Techniques commonly used in malicious Flash files, such as obfuscation, heap spraying, and type confusion exploitation are discussed.
- **Detection Based on a Combination of Static *and* Dynamic Analysis.**
  A hybrid approach to analyzing Flash files is presented, in which the strengths of static and dynamic analysis are combined.
- **Classification Based on Predefined Features.**
  Classification is performed by a combination of a naive Bayesian classifier and a single vulnerability-specific filter. The naive Bayesian classifier is based on a set of predefined features.
- **Evaluation.**
  The merits of our approach are confirmed. Tests performed on 691 benign files and 1,184 malicious files show low false negative and false positive rates of around 1.87% and 2.01%, respectively. These rates are shown to be comparable with or better than those of commercial anti-virus products

The rest of this paper is organized as follows. A number of common techniques employed by ActionScript 3 malware are outlined in Section 2. FlashDetect's implementation details are discussed in Section 3, while Section 4 lists the set of features used for classification. Section 5 presents the experimental results. Finally, FlashDetect's limitations are discussed in Section 6, while Section 7 discusses related publications.

2 Common Flash Exploit Techniques

This section provides a brief overview of some common techniques employed by Flash malware. Where applicable, comparisons are made with techniques common to JavaScript malware.

2.1 Obfuscation

Obfuscation is often employed by malicious JavaScript or shellcode, and malicious Flash files are no exception. Obfuscation techniques differ according to the technology available for obfuscation. For example, JavaScript is an interpreted language lacking a bytecode representation. Consequently, obfuscation in JavaScript often consists of identifier mangling and/or repeated calls to `eval()`. Shellcode obfuscation, on the other hand, is often achieved by packing the binary data in a specific format, possibly using the XOR operation

We have examined hundreds of malicious Flash files to determine if and how they perform obfuscation, and to develop ways of detecting such obfuscation. The description that follows is the result of that effort.

Flash files are created by compiling ActionScript 3 scripts into a bytecode representation and wrapping that bytecode in an SWF container. Because the ActionScript 3 virtual machine interprets bytecode, a JavaScript-style `eval()` function is not supported. Consequently, Flash obfuscation techniques have more in common with obfuscation techniques for shellcode than with those for JavaScript.

Note that we distinguish two types of obfuscation. The first type is source-code level obfuscation (e.g. identifier renaming). This type of obfuscation is heavily used in JavaScript malware, but given that ActionScript is distributed in bytecode form, it is less prevalent in Flash malware. The second type of obfuscation consists of multiple levels of embedded code. Since our detector will analyze the bytecode of a Flash file, we are most interested in the latter form of deobfuscation.

Though ActionScript 3 does not support `eval()`, it does support the runtime loading of SWF files. This is achieved by calling `Loader.loadBytes()` on a `ByteArray` containing the SWF's data. Using the `DefineBinaryData` SWF tag, arbitrary binary data can be embedded into an SWF file. At runtime, the data becomes available to ActionScript in the form of a `ByteArray` instance.

The `DefineBinaryData` SWF tag is often used in combination with the `Loader.loadBytes()` method to implement a primitive form of obfuscation. However, given that static extraction of `DefineBinaryData` tags is fairly easy using a range of commercial or open-source tools, obfuscation almost never stops there. Instead, malicious Flash files often encode or encrypt the embedded binary data and decode it at runtime before calling `Loader.loadBytes()`.

As is the case with JavaScript [8], obfuscation is actively used by both benign and malicious Flash files, as evidenced by commercial obfuscators such as DoSWF [5] and Kindi [11]. Thus, the mere presence of obfuscated code is not a good indicator of the maliciousness of a Flash file. Therefore, a need for a dynamic extraction method arises, allowing static analysis to be performed after deobfuscation. To this end, we have modified the Lightspark flash player so that each time `Loader.loadBytes()` is called, the content of the `ByteArray` is dumped into a file for later analysis. This allows for reliable extraction of embedded SWFs, as long as the deobfuscation code path is reached. In cases where the deobfuscation code path is not reached and no files are dynamically extracted, we fall back to a simple static extractor.

2.2 Heap Spraying

Heap spraying is an extremely common technique found in contemporary malware, and as such, it is commonly employed in ActionScript 3 malware. As is the case with obfuscation techniques, the way a heap spray is performed depends on the environment in which it needs to be performed. For example, in JavaScript, heap sprays are often performed by creating a string, and repeatedly concatenating the string with itself. In ActionScript 3, the most common way to perform a heap spray is through the use of a `ByteArray`.

The `ByteArray` class available in ActionScript 3 provides byte-level access to a chunk of data. It allows reading and writing of arbitrary bytes, and also allows the reading and writing of the binary representation of integers, floating point numbers, and strings. The implementation of the `ByteArray` class in the ActionScript 3 virtual machine uses a contiguous chunk of memory that is expanded when necessary to store the contents of the array. Therefore, the `ByteArray` class is a prime candidate for performing heap sprays.

Heap spraying code often uses one `ByteArray` containing the shellcode to be written, and a second `ByteArray` to perform the actual heap spray on. A simple loop is then used to repeatedly copy the first array's contents into the second. This results in the second array's memory chunk becoming very large, covering a large portion of the process's memory space with shellcode. Another common way to perform a heap spray is to use a string that contains the hexadecimal, base64, or some other encoding of the shellcode. This string is then decoded before being repeatedly written into a `ByteArray`, until it covers a large portion of the memory space.

2.3 JIT Spraying

The concept of JIT spraying in ActionScript 3 has been introduced in [2]. In that paper, the author shows how the JIT compiler in the ActionScript 3 virtual machine can be forced to generate executable blocks of code with almost identical semantics to some specified shellcode. It is shown that a chain of XOR operations performed on a set of specially crafted integers is compiled to native code in such a way that, when the code is jumped into at an offset, it is semantically equivalent to the given shellcode. The concept of JIT spraying is significant because it allows bypassing the *Data Execution Protection (DEP)* feature present in most modern operating systems.

We have observed that exploits targeting the Flash Player and using JIT sprays are fairly common. It seems that the most common way in which a JIT spray is performed consists of repeatedly loading an embedded SWF file containing the bytecode that performs the repeated XOR operations. This repetition ensures that multiple blocks of executable shellcode are present in the memory of the Flash player. Furthermore, the shellcode itself often contains a NOP-sled to further enhance the chances of successful exploitation.

2.4 Malicious ActionScript 3 as Exploit Facilitator

Although a range of vulnerabilities have allowed the ActionScript 3 virtual machine to be a target for exploitation, another common use of Flash malware seems to be that of a *facilitator* of other exploits. We have observed instances where a malicious Flash file merely performs a heap spray, without trying to gain control of execution. This seems to indicate that the malicious file is meant to facilitate another exploit, for example one targeting a JavaScript engine. The rationale behind this behavior is that Flash files are often embedded in other resources, and as such, they can facilitate exploits targeting the technologies related to those resources.

For example, exploits targeting a JavaScript engine might use ActionScript 3 to perform a heap spray, after which the actual control of execution is gained through a vulnerability in the JavaScript engine. However, the value of such types of exploits is declining, as browsers are increasingly separating the browser from its plugins by running those plugins in separate processes. Another possible scenario in which a malicious Flash file acts as an exploit facilitator is that in which a Flash file is embedded in a PDF. In such a case, the Flash file might perform a heap spray, while the actual control of execution is gained through a vulnerability in the JavaScript engine in Adobe's Acrobat Reader. Finally, as mentioned in the previous section, ActionScript 3 is often used to facilitate exploits targeting the ActionScript 2 virtual machine.

These examples illustrate that Flash is a versatile tool for malware authors, as Flash files can be used both to launch full-fledged attacks, as well as to act as a facilitator for the exploitation of other technologies.

2.5 Type Confusion Exploitation

A relatively recent development has been the exploitation of type confusion vulnerabilities present in both the ActionScript 2 and ActionScript 3 virtual machine. These types of exploits are interesting since they often allow an attacker to construct a very reliable exploit that completely bypasses both *Data Execution Protection* and *Address Space Layout Randomization*, without relying on heap or JIT spraying. There are a number of vulnerabilities for which the exploit code, or at least parts of it, have been published. Among these are CVE-2010-3654 and CVE-2011-0609, which relate to the ActionScript 3 virtual machine, and CVE-2011-0611, which relates to the ActionScript 2 virtual machine.

Listing 1.1. Implementation of the 3 classes used to exploit CVE-2010-3654.

```
class OriginalClass {
    static public function getPointer (o:Object):uint { return 0; }
    static public function tagAsNumber (u:uint):* { }
    static public function fakeObject (p:uint):SomeClass { return null; }
}

class ConfusedClass {
    static public function getPointer(o:Object):Object { return o; }
    static public function tagAsNumber(p:uint):uint { return p | 0x7 }
    static public function fakeObject(p:uint):uint { return p; }
}

class SomeClass {
    public function someMethod():* {}
}
```

Other security advisories, such as CVE-2012-0752, are also known to relate to type confusion vulnerabilities, although their exploit code is not public yet.

An exploit for CVE-2010-3654 is presented by Li in [13]. We present a slightly simplified adaption of this exploit, to illustrate the way in which a type confusion exploit works.

ActionScript 3 Virtual Machine Implementation. The ActionScript 3 virtual machine operates on data types called 'atoms.' Atoms consist of 29 bits of data followed by a 3-bit type-tag. The data can either be an integer or an 8-byte-aligned pointer to some larger data (as is the case for atoms representing objects or floating point numbers). The type-tags allow the virtual machine to support runtime type detection when a variable's type is not specified in the source code.

The ActionScript 3 virtual machine also contains a JIT compiler that compiles ActionScript 3 methods to native code. The native code for such a method works solely with native data types, not atoms. Thus, native code methods return ActionScript objects as pointers, integers as 32-bit integers, and floating point numbers as pointers to IEEE 754 double precision numbers. Code calling a native code method then wraps the result into a type-tag to form an atom, so that the result can be used by the virtual machine. The type-tag that is used when wrapping native code results depends on the method's return type, which is specified by the class definition.

Elements of a CVE-2010-3654 Exploit. An exploit for CVE-2010-3654 consists of three classes. Listing 1.1 lists the implementation of those three classes. To trigger the vulnerability, the list of identifiers in the compiled bytecode is modified such that the name of `ConfusedClass`'s identifier is changed to `OriginalClass`. After this modification, the list of identifiers will have two entries named `OriginalClass`. The result is that in vulnerable version of the Flash player, the `ConfusedClass` is 'type confused' with the `OriginalClass`.

When `ConfusedClass` is confused with `OriginalClass`, calls to methods of `OriginalClass` instead end up calling the native code implementations of `ConfusedClass`'s corresponding methods. However, when wrapping the results of these native code methods, the type-tag that is used depends on the return types defined in `OriginalClass`. This mismatch, where on the one hand `ConfusedClass`'s native code methods are called, while on the other hand `OriginalClass`'s return types are used, results in an exploitable vulnerability.

The following sections show how this vulnerability can be used to leak addresses of objects, read arbitrary memory and gain control of execution.

Leaking Objects' Memory Addresses. Because of the type confusion, the pointers to the `Object`s returned by `ConfusedClass.getPointer` are wrapped in `uint` type-tags (as specified by `OriginalClass`), exposing the pointer to the ActionScript 3 runtime. For example, `OriginalClass.getPointer(new ByteArray())` will actually call `ConfusedClass.getPointer` and return the memory address of the `ByteArray` in the form of an integer.

Reading Arbitrary Memory Addresses. Similarly, the integers returned by `ConfusedClass.tagAsNumber` end being used as if they are atoms with a type-tag. This is because the type of the returned value needs to be inferred at runtime, as no type is specified in `OriginalClass`. The 0x7 type-tag that is added by `ConfusedClass.tagAsNumber` is that of a floating point number atom.

In the ActionScript 3 virtual machine, the data of a floating point number atom is an 8-byte-aligned pointer to an IEEE 754 double precision number. Thus, after a call to `tagAsNumber`, the given integer will be used as if it is an 8-byte-aligned pointer pointing to valid IEEE 754 data.

This effectively allows an attacker to read arbitrary memory locations by passing the memory location to `tagAsNumber` and then writing the 'fake' floating point number to a `ByteArray`. This results in the 8 bytes at the given location being written into the `ByteArray`. These bytes can then be read separately using the methods provided by the `ByteArray` class.

Gaining Control of Execution. Finally, by passing a memory address to the `fakeObject` method, one can create a 'fake' object of type `SomeClass` whose representation is *supposedly* stored at the given memory location. When `someMethod` is called on this fake object, the virtual machine will access the object's vtable at a certain offset from the given memory address, look up the method's address, and then jump to it. Thus, by specifically crafting the memory at the given memory location, the attacker can make the virtual machine hand over control of execution to a piece of memory under his control. Crafting such a chunk of memory can easily be done by using a `ByteArray` instance and then leaking the address of that `ByteArray`'s data using a combination of the previous two techniques.

Bypassing DEP. It is clear that by being able to leak pointers to objects, read arbitrary memory addresses and gain control of execution, one can easily create a basic exploit that bypasses ASLR. However, DEP still prevents such exploits from working, since any shellcode produced in ActionScript will be non-executable.

There is, however, an important element that circumvents this last hurdle: the first 4 bytes of an ActionScript object's representation point to a memory address in the Flash player DLL that is always at constant offset to the start of the DLL. Thus, by reading these 4 bytes, an attacker can infer the base address of the Flash player DLL in the process's memory.

Since the DLL contains a call to `VirtualProtect`, an attacker can use the information to discover the address of that function. Therefore, all an attacker needs to do to circumvent DEP is to create a return-oriented programming attack that calls `VirtualProtect` on some shellcode, using the gadgets available in the DLL. Afterwards, the shellcode, *which can be arbitrarily large*, can be jumped to and executed.

2.6 Environment-Identifying Code

Environment-identifying code reads some property and compares it to some constant. The result is then used in a conditional branch. Such code is often used to selectively launch an exploit only if the Flash file is being executed by a vulnerable version of the Flash player. Therefore, a malicious Flash file might not exhibit distinctive behavior if such environment-identifying code fails to identify a vulnerable Flash player instance. From our observations we have concluded that environment-identifying code is fairly common in malicious Flash files. Since environment-identification is often used to target specific versions of the Flash runtime, it is not possible to instrument the Lightspark player to return a single version string that is vulnerable to all exploits.

To improve the detection rate and to maximize the chances of successfully deobfuscating any embedded SWF files, we have instrumented the Lightspark player to taint all environment-identifying properties. More precisely, all properties of the `Capabilities` class are tainted. These taint values are propagated through all basic ActionScript 3 operations (e.g., string concatenation). Subsequently, whenever an `ifeq` ('if equals') branch depends on a tainted value, that branch is forcibly, taken no matter what the actual result of the comparison operation is. On the other hand, complementary `ifne` ('if not equals') branches depending on tainted values are never taken.

Listing 1.2 contains a simplified excerpt of a real malicious sample performing environment-identification. The sample matches the Flash player's version (obtained from the `Capabilities.version` property) to a predefined set of versions. Each version has an accompanying embedded SWF file containing an exploit targeting that version. With the original Lightspark player, chances are high that both `ifne` conditional branches would be taken, resulting in the malicious Flash file not launching any exploit. However, because of the use of taint-propagation, none of the `ifne` branches are taken in our analysis environment, resulting in the last exploit being loaded.

Listing 1.2. Excerpt of a sample using environment-identifying bytecode

```
    getlex           flash.system::Capabilities
    getproperty      version
    coerce           String
    setlocal         15
    getlocal         15
    pushstring       "WIN 9,0,115,0"
    ifne             L1

    // load exploit targeting Windows Flash Player version 9.0.115

L1: getlocal         15
    pushstring       "WIN 9,0,16,0"
    ifne             L2

    // load exploit targeting Windows Flash Player version 9.0.16

L2: // this Flash player is not vulnerable, do not load any exploit
```

Note that this simple approach works well, as most environment-identifying code uses inclusive rules to determine vulnerable Flash players. That is, instead of determining that a given Flash player instance *is not* vulnerable, most environment-identifying code determines that a given instance *is* vulnerable. However, it is clear that this approach can be circumvented, a scenario that is discussed in more detail in Section 6.

3 Detector Implementation

FlashDetect's analysis of a Flash file is split into three phases. In the first phase, the Flash file is dynamically analyzed using an instrumented version of the Lightspark flash player. The second phase leverages a static analysis of the ActionScript 3 bytecode of the Flash file (including any bytecode found through deobfuscation during the dynamic analysis phase). Finally, in the last phase, the Flash file is classified using the set of features described in Section 4.

3.1 Phase I: Dynamic Analysis

The dynamic analysis of submitted Flash files is performed by an instrumented version of the Lightspark flash player that saves a trace of interesting events, such as the calling of methods, access to properties, or the instantiation of classes.

After the dynamic analysis, this trace is analyzed to determine the values of the features used for classification.

Additionally, the instrumented Lightspark version saves every SWF file that is loaded at runtime. This allows the subsequent static analysis to take into account both the original SWF file and any other embedded, possibly obfuscated, SWF files that were loaded at runtime.

During the dynamic analysis, as soon as the first bytecode instruction is executed, a timer is started with a given timeout value. When the timer runs out, the dynamic analysis ends. This way, each Flash file is analyzed for more or less the same amount of time. At the same time, starting the timer only after the first instruction has executed ensures that the time spent loading a Flash file has no effect on the actual amount of time for which the file's scripted behavior is analyzed.

3.2 Phase II: Static Analysis

After performing the dynamic analysis, we perform a static analysis of both the original SWF file as well as any deobfuscated SWF files. The majority of the static analysis phase is spent analyzing the ActionScript 3 bytecode found in the SWFs. However, a small part of the static analysis checks for commonly exploited vulnerabilities in the Flash player's SWF parser, such as integer overflows in SWF tags.

As is the case with the dynamic analysis, the goal of the static analysis is to determine the values for the set of features used for classification. Some feature values are determined during both the dynamic and static analysis phase. For example, the feature that captures a Flash file accessing the `Capabilities.version` property, commonly used for environment-identification purposes, is detected during both analysis phases.

Checking the same feature during both phases might seem redundant, but it has a practical advantage. As the Lightspark flash player is a fairly recent project, it does not yet run all Flash files correctly. Hence, it is possible that the dynamic analysis phase will be cut short because of an unexpected error. However, because the same feature is also checked in the static analysis phase, chances are high that the feature will still be correctly detected. On the other hand, since it is possible to obfuscate property or method names, static analysis might fail to detect certain features. However, as long as the dynamic analysis succeeds, that feature will still be detected correctly.

3.3 Phase III: Classification

A naive Bayesian classifier is used to classify submitted samples using the set of features mentioned earlier. The classifier accepts features with both a Boolean value domain and a continuous value domain. A Laplacian correction is applied to Boolean features to convert zero-probabilities to very small probabilities.

As is common with naive Bayesian classifiers, the normal distribution is used to model continuous features. Thus, the probability of continuous features is estimated using the normal probability density function.

$$f(x, \mu, \sigma^2) = \frac{1}{\sigma\sqrt{2\pi}} exp\Big(\frac{-(x-\mu)^2}{2\sigma^2}\Big)$$

Additionally, calculated probabilities are clamped to a minimum probability of 1e-10 , so as to prohibit any single feature from influencing the classification decision too much.

Finally, there is one feature that we consider a definitive indicator of maliciousness, and for which we bypass the naive Bayesian classifier. This feature is discussed in Section 4.

4 Features Used for Classification

Our features can be grouped according to the type of behavior they characterize. Boolean features are marked by *(B)*, while continuous features are marked by *(C)*.

Features Related to Embedded and Obfuscated SWF Files or Shellcode. The group consists of the following features:

- **shellcode (B):** indicates whether the `sctest` tool from the *Libemu* library detected shellcode embedded in the Flash file.
- **load-loadbytes, loader-load (B):** indicate whether the Flash file uses the `loadBytes` or `load` methods of the `Loader` class to load SWF files at runtime.
- **obfuscation-method-ratio (C):** represents the ratio of deobfuscation-related method calls to the overall number of method calls; the methods are `fromCharCode()`, `charCodeAt` and `slice()` of the `String` class, in addition to `parseInt`. These methods are frequently used in deobfuscation in the wild.
- **bytearray-method-ratio (C):** the ratio of `ByteArray`-related method calls to the total number method calls. A large ratio is indicative of deobfuscation and/or heap spraying.
- **bytearray-callprop-ratio (C):** the ratio of `ByteArray` method-calling bytecode instructions to the total number method-calling instructions. A large ratio is indicative of deobfuscation and/or heap spraying.
- **avg-pushstring-char-range (C):** the average range of characters in `pushstring` instructions; indicative of strings containing binary data such as shellcode.
- **avg-base64-pushstring-length (C):** the average length of strings pushed by `pushstring` instructions that match the base64 character set; indicative of obfuscated data.

Of the above features, the *shellcode*, *bytearray-callprop-ratio*, *avg-pushstring-char-range* and *avg-base64-pushstring-length* features are only checked during the static analysis phase. The *obfuscation-method-ratio* and *bytearray-method-ratio* features are only checked during the dynamic analysis phase. The *loader-loadbytes* and *loader-load* features are checked during both phases of the analysis.

Features Related to Environment-Awareness

- **checks-url (B):** indicates whether the Flash file checks the URL of the webpage it is embedded in; the feature indicates access to the `LoaderInfo.url` property or access through the `ExternalInterface` class to the `window.location` property of the embedding HTML page. Most malicious files *do not* check the URL, while a lot of benign files *do*, so this feature is indicative of benign behavior rather than malicious behavior.
- **external-interface (B):** indicates whether the Flash file uses any of the methods of the `ExternalInterface` class to communicate with the external host of the Flash plugin, such as the web browser. Most malicious files *do not* use `ExternalInterface`, while a lot of benign files *do*, so this feature is indicative of benign behavior rather than malicious behavior.
- **checks-capabilities (B):** indicates whether the Flash file uses any of the properties of the `Capabilities` class; indicative of environment-identification often used by malicious files in the wild. Additionally, there are five features that specifically indicate access to the `version`, `playerType`, and `isDebugger` properties of the `Capabilities` class.

All features in this group are checked during both analysis phases.

Features Related to General Runtime Behavior. The features in this group all provide an indication how a given Flash file behaves in very general terms. For example, a very high amount of method calls or high ratio of `push` opcodes is often indicative of deobfuscation or heap spraying.

- **method-calls-per-second, method-calls-per-cpusecond (C):** the number of built-in methods (provided by the Flash Player runtime) called during the dynamic analysis phase, normalized by the running time in terms of wall clock or CPU time, respectively. These features each have another variant in which the value is additionally normalized by the size of the bytecode present in the SWF file.
- **avg-*opcode*-ratio (C):** a group of features that indicate the ratio of a certain opcode to the total number of opcodes. The opcodes for which ratios are determined are `bitxor`, the `push` opcodes used to push data on the stack, and the `call` opcodes used to call methods.

The values to the *avg-opcode-ratio* features are determined during the static analysis phase, while the other features' values are determined during the dynamic analysis phase.

Vulnerability-Specific Features

- **bad-scene-count (B):** indicates whether an invalid `SceneCount` value was detected inside the `DefineSceneAndFrameLabelData` SWF tag. This is indicative of an exploit targeting the CVE-2007-0071 vulnerability.

The value to this feature is determined during the static analysis phase.

This feature is used because some malicious files only trigger the CVE-2007-0071 vulnerability, without containing any other malicious content or bytecode. These files are probably meant to be embedded inside other files that do contain malicious content (e.g. shellcode), to act as exploit triggers.

Given that these types of malicious files would not be detected by the other features, we use this vulnerability-specific feature, enabling a fair comparison with other detection products. Given that the presence of this feature is such a definitive sign of maliciousness, we immediately consider files exhibiting the feature to be malicious, bypassing the naive Bayesian classifier, as mentioned earlier in Section 3.3.

5 Evaluation

5.1 Sample Selection

To evaluate the accuracy of FlashDetect, we tested the classifier on a set of Flash files that are known to be malicious or benign.

Benign Samples. The benign samples were manually verified to be benign and were gathered by crawling the following sources:

- Miniclips.com, an online games website.
- Various websites offering free Flash webdesign templates.
- Google search results for the query `filetype:swf`.

Additionally, the benign sample set includes Flash files submitted to Wepawet that, after manual verification, were found to be benign. In total, the benign sample set consists of 691 Flash files.

Malicious Samples. The malicious samples were gathered from files submitted to Wepawet, and they were manually verified to be malicious. Hence, the samples were categorized according to their similarity, as shown in Table 1. The table also shows whether or not the files in each category contain embedded/obfuscated SWF files, and whether or not the environment-identifying `Capabilities` class is accessed. Note that 7 out of 12 categories use embedded, possibly obfuscated, SWFs, while 4 out of 12 categories perform environment-identification.

Table 2 shows the fraction of benign and malicious files that access the different properties of the `Capabilities` class that are used for environment-identification, and that are also used as features for classification.

Table 1. Categorization of malicious samples with the number of samples. Also shown is whether or not embedded SWF files are used, and whether or not the environment-identifying `Capabilities` class is accessed.

Group	Number of samples	Embedded files	`Capabilities` access
zasder	1016	✓	✓
corrupt-obfuscated-avm1	49	✓	✓
uncategorized	24	✓	✓
loaderinfo-parameters-sc	16		
pop	15		
hs	14	✓	
woshi2bbd-jitspray	11	✓	✓
jitegg	10	✓	
flashspray	9		
badscene	9		
doswf-sc1	6	✓	
heapspray-1	5		
Total	1184		

Table 2. Fraction of files that access the different properties of the `Capabilities` class

	`isDebugger`	`playerType`	`version`
Benign files	0.330	0.378	0.421
Malicious Files	0.003	0.872	0.870

Finally, some of the categories are notable for the way the exploits work.

- **Zasder.** Contains files that employ environment-identification and multiple levels of obfuscation, and eventually try to exploit CVE-2007-0071 [14].
- **Corrupt-obfuscated-avm1**. Contains files that load an obfuscated ActionScript 2 Flash file with a corrupt structure that presumably triggers some vulnerability in the Flash player. As such, these files are good examples of ActionScript 3 being used as an ActionScript 2 exploit facilitator.
- **Woshi2bbd-jitspray.** These files repeatedly load an obfuscated SWF that contains JIT spraying bytecode, after which a final SWF is loaded that presumably tries to exploit some vulnerability.
- **Flashspray.** These files only call a JavaScript function called `FLASHSPRAY` in the HTML page embedding the file. They are presumably used to circumvent JavaScript malware detectors. Again, these files are good examples of malicious ActionScript 3 in an exploit facilitating role, this time probably facilitating an exploit targeting a browser vulnerability.

5.2 Experimental Results

To test the efficacy of the classifier, we hand-picked a set of training samples from the malicious samples. This set of 47 samples includes at least one sample from each category. This way, no single category is over-represented in the training data, even if that category has many more samples than the others. For the benign training data, we randomly selected 47 benign samples. The classifier was then trained on the training samples, and tested on the remaining 1781 samples. We repeated this test 20 times, each time with a different set of randomly selected benign training samples.

In addition to testing the classifier using manually selected malicious training samples, we tested the classifier using randomly selected samples. For this purpose, we partitioned the randomized malicious samples into 20 disjunct sets. Subsequently, we used each set in turn as the malicious training sample set. An equal number of benign training samples accompanying the malicious training samples were again randomly selected. By the pigeonhole principle, this setup ensures that for all but the three largest malicious categories, there is at least one test in which such a category is not represented in the training samples. This provides a way to test the classifier's performance on malicious flash files of a previously unknown category.

Error Rates. Figure 1 contains the ROC curves displaying the accuracy of our classifier at various classification thresholds, using both manually selected and randomly selected malicious training samples. These ROC curves show the true positive rate as a function of the false positive rate, visualizing the trade-offs required to achieve a given accuracy.

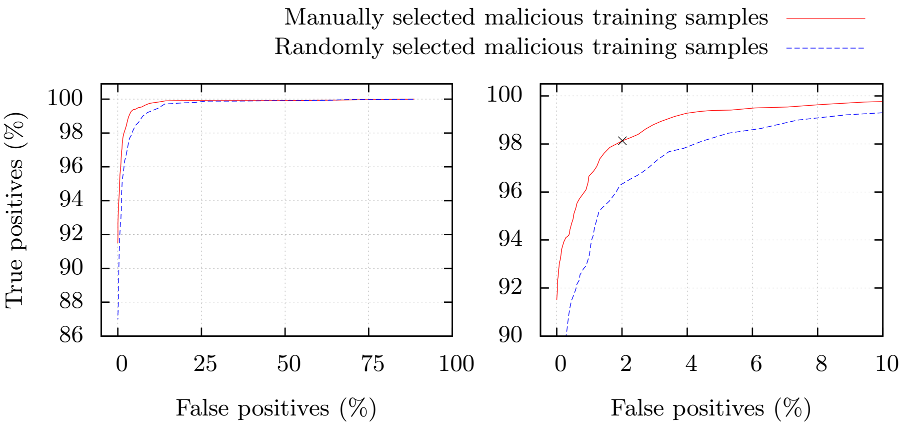


Fig. 1. ROC curves displaying FlashDetect's accuracy at various classification thresholds, when using manually or randomly selected training samples. The right-hand plot zooms in on the upper corner of the left-hand plot.

As shown in the plot, when using manually selected training samples, a false positive rate of 0% is reached at a true positive rate of around 91.5% (i.e. a false negative rate of around 8.5%). A true positive rate of more than 99.9% is reached at a false positive rate of around 16%. However, a true positive rate of exactly 100% is only reached at a false positive rate of around 88.7%. Also note that an equilibrium between false positive and false negative rates is achieved at around 2%, at a classification threshold of 25 (marked with an 'x'). It is at this equilibrium that the classifier's accuracy is most balanced, and as such, this is the threshold we use in the next experiment, comparing our performance to that of commercial anti-virus products.

The plot also shows the performance of the classifier using randomly selected training samples, as described earlier. It is clear that the classifier is less accurate when using randomly selected samples than when using manually selected samples. However, the performance is not significantly worse, with an equilibrium between false positive and false negatives at around 3%, compared to 2% when using manually selected samples. Thus, we conclude that while manual sample selection increases the classifier's accuracy, it does not significantly bias towards malicious files of known categories.

Comparison with Commercial Anti-Virus Products. To support our claim that FlashDetect's performance is comparable with or better than that of commercial anti-virus (AV) products, we include a comparison of FlashDetect's efficacy with that of AV products. We used the VirusTotal [24] service to run 43 commercial and open-source AV products on our test sample set. The results of the five best-performing AV products, as determined by the false negative rate, are compared to FlashDetect's results, at a set classifier threshold of 25.

The first row of Table 3 lists the false negative rates for the five best-performing AV products, together with FlashDetect's rates. As shown, FlashDetect's average false negative rate is less than that of four out of the five best-performing AV products. It is also interesting to note that out of the 43 AV products we tested, 35 products had a false negative rate in excess of 65%, and 32 products had a false negative rate in excess of 90%.

The second row of Table 3 lists the false positive rates. Note that FlashDetect's false positive rate is worse than that of the top five AV products. This can be explained by the fact that these AV products probably use signature-based detection methods (confirmed for four out five). However, given that FlashDetect is able to reach a comparable false negative rate with a relatively low false positive rate, we conclude that FlashDetect's performance is certainly competitive with that of commercial AV products.

6 Limitations

Identifying the Presence of FlashDetect. Malicious Flash files might try to identify the presence of FlashDetect. The Lightspark player is still a

Table 3. FlashDetect's false negative and false positive rates at a classifier threshold of 25 compared to the five best-performing AV products

	FlashDetect	AV1	AV2	AV3	AV4	AV5
False negatives	1.87%	0.97%	2.64%	2.73%	2.74%	2.81%
False positives	2.01%	0.42%	0.00%	0.27%	0.27%	0.27%

relatively young project, and thus, it still has a long way to go before its behavior perfectly matches that of the official Flash player. This provides malware writers with a number of ways to detect the presence of Lightspark, and as a result, FlashDetect. This limitation is inherent to the implementation of FlashDetect's dynamic analysis phase. However, we assume that as the Lightspark project matures it will become increasingly harder to differentiate Lightspark from the official Flash player.

Environment-Identifying Code Circumvention. The method for handling environment-identifying code described in Section 2.6 is obviously not very robust. For example, the current method can easily be circumvented by using exclusive matching instead of inclusive matching. That is, matching against non-vulnerable version instead of vulnerable versions. Additionally, comparison instructions besides `ifeq` and `ifne` are not checked for environment-identification.

However, from our observations we conclude that, since most malicious Flash files currently found in the wild use inclusive environment-identification based on direct equality instructions, the current method is quite effective at enhancing detection rates. Nevertheless, in the future, a more robust method to handle environment-identifying code would be in order. Such a method could consist of a multi-execution virtual machine capable of simultaneously analyzing multiple code branches, such as described for JavaScript in [12].

Dependence of Certain Features on a Measure of Time. During the dynamic analysis phase, the Flash file is run for a limited amount of time. Therefore, the usefulness of the dynamic analysis phase inherently depends on the fact that, in general, malicious files will attempt exploitation as soon as possible. Additionally, certain dynamic features (e.g., *method-calls-per-second*) used for classification are inherently dependent on a measure of time.

The usefulness of time-dependent features and indeed, the dynamic analysis phase in general, could be reduced if malicious files were to delay the start of exploitation for a certain amount of time. Therefore, the dependence of certain features on a measure of time is an inherent limitation of our system.

However, launching an exploit as soon as possible is advantageous to malware authors as it increases the chances of the exploit being successful. Consequently, one can argue that maximizing the number of successful exploitations is more important to malware authors than evading detection. Indeed, the complete lack of obfuscation in some of the malicious samples we have observed indicates that some malware authors do not even bother to evade detection anymore.

Overall Robustness of Features. Some of the individual features used for classification may not be very robust. Examples are the time-dependent dynamic features. However, the combination of all features being used together results in a robust system. Indeed, the features used detect a number of different types of behavior, such as obfuscation, JIT spraying, or environment-identification. Most of these types of behavior are detected by more than one feature, and they are often detected during both the dynamic and static analysis phase. This results in a robust system capable of detecting a wide range of low-level exploits.

7 Related Work

7.1 Exploit Techniques

Blazakis [2] discusses JIT spraying attacks against the ActionScript 3 virtual machine. Li [13] discusses the exploitation of the type confusion vulnerability CVE-2010-3654, while [9] discusses a very similar vulnerability CVE-2011-0609. An exploit using sophisticated obfuscation techniques that targets CVE-2007-0071 is dissected by Liu in [14].

7.2 Malware Detection

FlashDetect's implementation is an evolution of OdoSwiff described by Ford *et al.* [6]. Additionally, [6] is one of only a very limited set of publications on the specific topic of malicious Flash file detection. Instead, most of the closest related research discusses malicious JavaScript detection approaches.

Cova *et al.* describe JSAND [3], a tool for analyzing JavaScript with an approach that is related to FlashDetect's approach, using classification based on a set of dynamic and static features. However, JSAND and FlashDetect use different features, due to the different nature of JavaScript and ActionScript.

Ratanaworabhan *et al.* [21] discuss Nozzle, a dynamic JavaScript analyzer that specifically focuses on detecting heap spraying code injection attacks. Nozzle's approach consists of interpreting individual objects on the heap as code and statically analyzing that code for maliciousness. This approach differs substantially from FlashDetect's approach, as FlashDetect's analysis is based on determining the general behavior of a Flash file through indicators such as the methods called by the file. Additionally, FlashDetect is not specifically focused on the detection of heap spraying exploits, but instead focuses on the more broader set of low-level exploits.

Zozzle [4] is a static JavaScript analyzer by Curtsinger *et al.* that also uses a naive Bayesian classifier to detect malicious files. However, the features used by Zozzle for classification are automatically extracted from the JavaScript's abstract syntax tree. In contrast, FlashDetect's static features are predefined, and it also uses predefined, dynamically extracted features.

The recent work on Rozzle by Kolbitsch *et al.* [12] describes an implementation for multi-execution in JavaScript. Their approach to multi-execution could be applied to Lightspark to enable the robust handling of Flash files using environment-identification.

There are several malware detection systems that use low-interaction or high-interaction honeyclients. Examples are HoneyMonkey [25], Capture-HPC [22], Moshcuk *et al.* [16,17], Provos *et al.* [20], and Monkey-Spider [7]. FlashDetect differs from such systems in that it does not automatically crawl websites. Instead, FlashDetect is designed to be used in conjunction with some other analyzer that feeds samples into FlashDetect for further analysis. Additionally, in contrast to high-interaction honeyclients, FlashDetect's analysis provides more insight into how an exploit works. High-interaction honeyclients on the other hand provide more insight into the effects of an exploit, something which FlashDetect does not currently do. However, in high-interaction honeypots, exploits must succeed for them to be detected, while this is not the case for FlashDetect.

8 Conclusion

We discussed several techniques commonly used by Flash malware. We have discussed how malware using ActionScript 3 often takes on a role of exploit facilitator, showing that a successful solution to detecting malicious Flash files is crucial. Subsequently, we have introduced FlashDetect, which uses a novel approach combining static and dynamic analysis to examine Flash files. FlashDetect's classification is based on a combination of predefined features. We have shown how these features, when used with a naive Bayesian classifier and a single vulnerability-specific filter, allow for high classification accuracy with a minimal amount of false negatives.

References

1. Adobe: Statistics: PC penetration, `http://www.adobe.com/products/flashplatformruntimes/statistics.edu.html` (accessed on June 15, 2012)
2. Blazakis, D.: Interpreter exploitation: Pointer inference and JIT spraying (2010), `http://www.semantiscope.com/research/BHDC2010/BHDC-2010-Paper.pdf` (accessed on June 15, 2012)
3. Cova, M., Kruegel, C., Vigna, G.: Detection and Analysis of Drive-by-Download Attacks and Malicious JavaScript Code. In: Proceedings of the World Wide Web Conference (WWW), Raleigh, NC (April 2010)
4. Curtsinger, C., Livshits, B., Zorn, B., Seifert, C.: Zozzle: Low-overhead mostly static JavaScript malware detection. In: Proceedings of the Usenix Security Symposium (August 2011)
5. DoSWF.com: DoSWF - Flash encryption, `http://www.doswf.com/doswf` (accessed on June 15, 2012)
6. Ford, S., Cova, M., Kruegel, C., Vigna, G.: Analyzing and detecting malicious flash advertisements. In: Proceedings of the 2009 Annual Computer Security Applications Conference, ACSAC 2009, pp. 363–372. IEEE Computer Society, Washington, DC, USA (2009)
7. Ikinci, A., Holz, T., Freiling, F.: Monkey-spider: Detecting malicious websites with low-interaction honeyclients. In: Proceedings of Sicherheit, Schutz und Zuverlässigkeit (2008)

8. JavaScript-Source.com: JavaScript obfuscator, http://javascript-source.com (accessed on June 15, 2012)
9. Joly, N.: Technical Analysis and Advanced Exploitation of Adobe Flash 0-Day, CVE-2011-0609 (2011), http://www.vupen.com/blog/20110326.Technical_Analysis_and_Win7_Exploitation_Adobe_Flash_0Day_CVE-2011-0609.php (accessed on June 15, 2012)
10. Keizer, G.: Attackers exploit latest Flash bug on large scale, says researcher, http://www.computerworld.com/s/article/9217758/Attackers_exploit_latest_Flash_bug_on_large_scale_says_researcher (accessed on June 15, 2012)
11. Kindi: secureSWF, http://www.kindi.com (accessed on June 15, 2012)
12. Kolbitsch, C., Livshits, B., Zorn, B., Seifert, C.: Rozzle: De-cloaking internet malware. In: IEEE Symposium on Security and Privacy (May 2012)
13. Li, H.: Understanding and Exploiting Flash ActionScript Vulnerabilities. In: CanSecWest 2011 (2011), http://www.fortiguard.com/sites/default/files/CanSecWest2011_Flash_ActionScript.pdf (accessed on June 15, 2012)
14. Liu, B.: Flash mob episode II: Attack of the clones (2009), http://blog.fortinet.com/flash-mob-episode-ii-attack-of-the-clones/ (accessed on June 15, 2012)
15. MITRE Corporation: Common Vulnerabilities and Exposures (CVE), http://cve.mitre.org (accessed on June 15, 2012)
16. Moshchuk, A., Bragin, T., Deville, D., Gribble, S.D., Levy, H.M.: Spyproxy: execution-based detection of malicious web content. In: Proceedings of 16th USENIX Security Symposium on USENIX Security Symposium, SS 2007, pp. 3:1–3:16. USENIX Association, Berkeley (2007), http://dl.acm.org/citation.cfm?id=1362903.1362906
17. Moshchuk, E., Bragin, T., Gribble, S.D., Levy, H.M.: A crawler-based study of spyware on the web (2006)
18. Paget, F.: McAfee Blog: Surrounded by Malicious PDFs, http://blogs.mcafee.com/mcafee-labs/surrounded-by-malicious-pdfs (accessed on June 15, 2012)
19. Alessandro, P., et al.: Lightspark flash player, http://lightspark.github.com (accessed on June 15, 2012)
20. Provos, N., Mavrommatis, P., Rajab, M.A., Monrose, F.: All your iframes point to us. In: Proceedings of the 17th Conference on Security Symposium, SS 2008, pp. 1–15. USENIX Association, Berkeley (2008), http://dl.acm.org/citation.cfm?id=1496711.1496712
21. Ratanaworabhan, P., Livshits, B., Zorn, B.: Nozzle: A defense against heap-spraying code injection attacks. In: Proceedings of the Usenix Security Symposium (August 2009)
22. The HoneyNet Project: CaptureHPC, https://projects.honeynet.org/capture-hpc (accessed on June 15, 2012)
23. Tung, L.: Flash exploits increase 40 fold in (2011), http://www.cso.com.au/article/403805/flash_exploits_increase_40_fold_2011 (accessed on June 15, 2012)
24. VirusTotal: VirusTotal service, https://www.virustotal.com (accessed on June 15, 2012)
25. Wang, Y.M., Beck, D., Jiang, X., Roussev, R.: Automated web patrol with strider honeymonkeys: Finding web sites that exploit browser vulnerabilities. In: IN NDSS (2006)

ALERT-ID: Analyze Logs of the Network Element in Real Time for Intrusion Detection

Jie Chu[1], Zihui Ge[2], Richard Huber[3], Ping Ji[1],
Jennifer Yates[2], and Yung-Chao Yu[3]

[1] Graduate Center CUNY, 365 Fifth Avenue, New York NY 10016
[2] AT&T Labs - Research, 180 Park Avenue, Florham Park NJ 07932
[3] AT&T Labs, 200 Laurel Avenue, Middletown, NJ 07748

Abstract. The security of the networking infrastructure (e.g., routers and switches) in large scale enterprise or Internet service provider (ISP) networks is mainly achieved through mechanisms such as access control lists (ACLs) at the edge of the network and deployment of centralized AAA (authentication, authorization and accounting) systems governing all access to network devices. However, a misconfigured edge router or a compromised user account may put the entire network at risk. In this paper, we propose enhancing existing security measures with an intrusion detection system overseeing all network management activities. We analyze device access logs collected via the AAA system, particularly TACACS+, in a global tier-1 ISP network and extract features that can be used to distinguish normal operational activities from rogue/anomalous ones. Based on our analyses, we develop a real-time intrusion detection system that constructs normal behavior models with respect to device access patterns and the configuration and control activities of individual accounts from their long-term historical logs and alerts in real-time when usage deviates from the models. Our evaluation shows that this system effectively identifies potential intrusions and misuses with an acceptable level of overall alarm rate.

1 Introduction

A fundamental aspect of network security is securing the networking infrastructure itself, which can be particularly challenging in a large scale enterprise or ISP (Internet service provider) network. In such networks, hundreds or thousands of routers and switches are widely dispersed among a geographically diverse set of offices and are typically managed by a large team of network operators. It is imperative that the networking infrastructure and the information contained therein be fully protected against any malicious priers and attackers. For example, information available at networking devices, such as router configuration and traffic statistics, may contain confidential business data of tremendous value to a business competitor. Divulging such information will likely result in a significant disadvantage to the ISP's business. Leakage of some critical security information in the router configuration such as QoS policy or firewall/ACL (Access Control List) settings may subject the network to crafted and targeted attacks such as DDoS (Distributed Denial of Service) attacks. Or in an even more devastating scenario, malicious attackers gaining privileged access to the networking device might

D. Balzarotti, S. Stolfo, and M. Cova (Eds.): RAID 2012, LNCS 7462, pp. 294–313, 2012.

alter the network configuration to create havoc and paralyze the entire network and the services it supports.

Given the risk of severe consequences, large scale networks typically devise and deploy a range of security and protection measures for their networking devices. One common practice is to utilize a combination of *periphery protection* and centralized *authentication and authorization* for communication to networking devices. By restricting premises access, unauthorized persons are blocked from gaining physical access to networking devices. Through careful configuration of ACLs at all network edge routers, unauthorized network traffic is also blocked from reaching network devices. And finally, technologies such as TACACS+ (Terminal Access Controller Access-Control System Plus) [2] and RADIUS (Remote Authentication Dial In User Service) [12], ensure that only authenticated users (i.e. authorized network operators/administrators) have access to routers and switches (either directly or remotely over the network).

The architecture above is very effective against threats from external attackers when working properly. However, there is always the possibility that building security is breached, allowing physical access to router hardware, or that ACLs on an edge router are misconfigured, admitting attacking traffic. Furthermore, with a large team of network operators, compromised users or compromised user accounts can be a critical source of potential security troubles arising inside the network.

In this paper, we propose to add another layer of defense for networking infrastructure by overseeing *all* operations being done in the network, and automatically detecting and raising alarms for "suspicious" activities. We leverage the existing authentication and authorization framework and collect router/switch access logs in real-time. We develop an anomaly detection system that compares on-going router/switch access activities against a set of patterns or profiles constructed from historical data, and once an anomaly is identified, triggers an alarm to network security managers for further investigation of potential intrusions and misuses.

Although the concept of intrusion detection system is well established in computer system security, applying the idea in networking device management remains unexplored, interesting, and challenging. To detect abnormal activities, we must obtain data on routine/normal network management activities in a large scale network, analyze that data, and determine what features best distinguish normal activities from abnormal ones. In our study, we base our analysis on actual network data from one of the largest ISP networks, which comprises tens of thousands of routers distributed worldwide. We conduct an in-depth analysis on a wide range of different characteristics about operators' access patterns and identify useful features. The effectiveness of an intrusion detection system is known to be limited by noisy baseline behavior and hence high false positives. Thus, when developing the detection methodology and the prototype system for capturing potential intrusions and misuses, we focus on managing false positives to be well within an acceptable range. Any given attack is likely to come from a small number of source subnets or accounts. Thus we aggregate detected "threat scores" by their origin source addresses and login accounts. This allows us to amplify the signal of offense and hence be able to detect offenders while they are still exploring the network before large-scale damage is inflicted.

Our contribution in this paper can be summarized as follows:

- We propose to systematically monitor and analyze the networking device access logs to protect the networking infrastructure. To the best of our knowledge, this is the first study that focuses on monitoring and auditing networking device access and control logs to catch anomalous activities.
- We conduct an in-depth analysis and characterization on the TACACS+ logs collected over more than six months from a tier-1 ISP network. We further identify a set of features that can be utilized to distinguish suspicious activities from normal operations — such as the login ID and origin IP prefix association pattern, the daily number of distinct routers accessed, and the number of hops over which an operator logs on to a router from a different router.
- we develop a tool for the ISP network. Our controlled experiment shows that it successfully identifies injected "malicious" activities – with corresponding threat scores significantly higher than those of day-to-day operational activities. When used in real operation, the system produces no more than a few threat alerts per day – a level at which network security managers are comfortable in conducting further investigation. Many of the threats detected have been found interesting and worth examining.

The rest of the paper is organized as follows. In Section 2, we provide an overview of operational management activities in large scale IP networks and a brief introduction of the authentication, authorization, and accounting system from which we collect logs. Section 3 presents our analysis result on the characteristics of normal operation activities. Section 4 describes the rules and detection system that we build for detecting and alerting on suspicious router accesses and controls. We evaluate our overall system performance in Section 5. Based on our operational experience, we propose a further enhancement to the system and evaluate its effectiveness in Section 6. We discuss related work in Section 7 and finally conclude the study in Section 8.

2 Background

2.1 Managing IP Networks

We first provide an overview of the various types of management activities in large scale IP networks. We describe these in the setting of a global ISP network although many of them are fundamental to large enterprise networks or regional ISP networks as well.

Managing a global ISP network requires a large team of network operators. These operators are typically organized in tiers – lower tier operators respond to routine issues following a set of predefined standard procedures, while more complex matters are escalated to upper tier operators, who have deeper knowledge and understanding of the network. Extremely complicated issues are escalated to a small group of experts, possibly including designers and support teams of vendors of involved devices.

Different tiers of operators have different functional roles. Some may be dedicated to the care of a high profile enterprise customer, in which case they will frequently access provider edge (PE) routers but seldom touch backbone routers. Some operators may be responsible for servicing the metropolitan area network for a certain region. Others may oversee control plane health (e.g., router CPU utilization) for the entire network. Depending on their role, operators are expected to have distinct patterns of network management activities.

Network operators often exercise control over routers and switches by logging on to the device. Today, nearly all networking devices support console access via direct connection to them and remote access via `ssh/telnet`. Control is exercised by invoking a sequence of commands through the Command Line Interface (CLI) of the device's operating system. For example, on Cisco IOS, typing

```
ping 1.2.3.4
```

triggers a ping test from the router to the IP address. And typing

```
enable
configure terminal
interface Ethernet0
shutdown
exit
```

administratively shuts down the interface *Ethernet0* at the router.

Note that Cisco IOS supports two different access levels – user level and privileged level. The `enable` command in the above example enters the privileged level, in which configuration change (`configure terminal`) is allowed. Such capability is widely supported on other vendor systems such as Juniper JunOS as well. In addition, AAA systems (described below) support finer grained command groups. A user cannot invoke commands outside of his/her predetermined access levels or command groups.

In addition to operators typing commands via the CLI, ISPs rely on a broad range of automated tools for their network management activities. These tools are typically designed to perform a specific set of functions. For example, automation tools/systems that perform configuration auditing periodically sweep through the entire network issuing a `show running-config` command to collect active router configurations. Another tool might collect hardware, traffic, or protocol status and statistics information by logging into the routers of interest and invoking commands such as `show process CPU history`, `show interfaces POS 1/0`, and `show ip bgp summary`. The tools may use designated logins when requesting access to networking devices.

Using the combination of function-level controls via various automated systems and manual command-level controls, operators are able to accomplish a wide range of network management tasks including provisioning and decommissioning customer services, troubleshooting networking and service problems, performing device life cycle management, taking measurements, and monitoring the health of the network and services.

2.2 Authentication, Authorization and Accounting

The networking infrastructure in large scale networks is typically protected by an AAA (Authentication, Authorization, and Accounting) system. There are two mainstream AAA frameworks widely used commercially – TACACS+ (Terminal Access Controller Access-Control System Plus) [2] and RADIUS (Remote Authentication Dial In User Service) [12]. While differing in some specifics, such as whether authentication and authorization are separately maintained in user profiles, both systems use one or more common servers to verify a user's identity (authentication) on login, verify access privilege (authorization) on a per command basis, and record all users' activites in their logs (accounting). The log entries contain critical information which includes

(i) the timestamp of the access request
(ii) the IP address of the targeted network device (e.g., Loopback address)
(iii) the IP address of the remote user requesting access
(iv) the user's login ID
(v) the command line executed
(vi) other information such as user terminal, privilege level, and timezone.

To use a TACACS+ or RADIUS system in a network, all routers in the network need to be configured with the IP addresses of the servers – typically there are multiple replicated servers for redundancy. A large network can further be divided into multiple zones, for example, by the device type or by the autonomous system (AS) that they belong to. Different zones may contain different user account and privilege settings.

3 Characteristics of Normal Operation Activities

As described in the Introduction, our objective is to monitor all operational activities in the network and detect potential intrusions and misuses. We start by examining normal network operational activities recorded in the AAA logs. We focus on aspects that would best distinguish normal activities from actions that an external or internal attacker might take. In the following analyses, we use data collected from a global tier-1 ISP network which generates tens of millions of TACACS+ log entries from tens of thousands of routers per day.

3.1 Failed Login Attempts

The most intuitive way to separate potential attacks from legitimate accesses is to check whether they can readily pass authentication. Attackers may expose themselves by inputting wrong login credentials. However, it is also expected that legitimate users sometimes "fat finger" their login ID or password. Thus, we examine failed login attempts in normal AAA logs (using one month's data).

Figure 1 plots the cumulative distribution function (CDF) of the number of consecutive login attempts before a successfully authenticated login. We consider a login request within one minute of a preceding one with the same origin IP, the same login ID, and the same target networking device as a consecutive login. We observe that more than 99.992% of logins pass authentication the first time. More than 85% of the remaining ones input the correct credentials the next time, and it is extremely rare that a user fails more than five times before finally getting it right. The ratio of login failure is considerably lower than that typically seen in computer systems[18]. This is likely due to the predominance of logins generated by automated network management tools – a unique characteristic of network infrastructure operations. Figure 1 demonstrates the potential of alarming on intruders when a small number (e.g., 6) of repeated failed logins are observed.

The above type of monitoring can be defeated if the attacker has a list of valid login IDs and device names – they can use a different login ID or target a different device

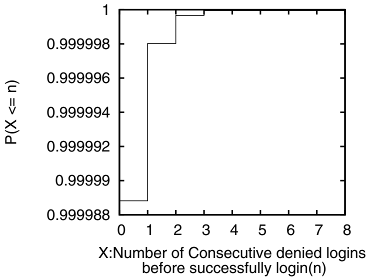


Fig. 1. CDF of the number of consecutive login attempts before a success

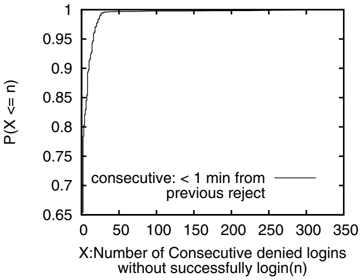


Fig. 2. CDF of the number of consecutive failed login from a common IP

when an attempt fails. We can improve detection by looking for consecutive login failures from a common origin IP irrespectively of the login ID attempted. Figure 2 plots the CDF of the number of consecutive (i.e., less than one minute apart from the preceding one) login failures. We observe that around 85% of the rejected login attempts are either rectified or abandoned in six times or fewer. However, there are some login attempts going as high as a few hundreds in a row. Manual inspection finds that they were due to network management scripts running out of sync with router CLI (e.g. sending password when login ID is expected or *vice versa*). This rarely occurs, but when it does it produces many consecutive login failures – and should correctly trigger an alarm.

3.2 Login Access Pattern

As described in Section 2, an AAA log entry contains login access information characterized by the user login ID, the origin IP address, and the target router IP address. We define a *login session* as the network management activities sharing the common triple and being close in time (e.g., with an idle timeout of 10 minutes).

The login access information can be valuable in capturing attackers. For example, an origin IP that is not part of a block of addresses previously seen as an originating address in the logs is a strong indication that the network periphery protection may have a hole. Furthermore, each network operator typically has a rather stable set of work locations from which he/she manages the network, and due to his/her role, there can be a fixed set of network devices that the operator typically manages. So source and destination IP addresses will tend to be consistent over time for many operators.

We first look at the association of the operators' login and the origin IP address. Figure 3 plots the CDF of the number of distinct origin IP addresses associated with a login ID in a month. We observe that 68% of login IDs manage the network from only one IP address. If we consider common origin subnets (with varying size), the number rises to 75% for /24 IP prefixes and 80% for /16 IP prefixes. In the rest of the paper, we will use /24 IP prefixes when aggregating origin IP address – it is not excessively large, yet can accommodate most of logins from the same facility/office.

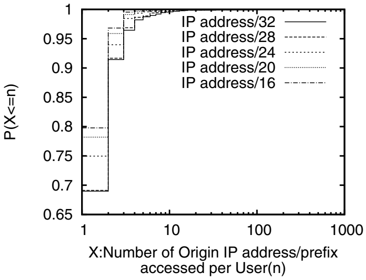


Fig. 3. CDF of the number of origin IP (prefix) per login ID

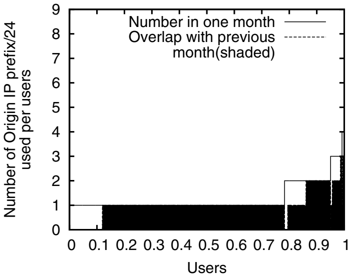


Fig. 4. Stability of the login ID and origin IP prefix association

Figure 3 also shows that even with /24 origin IP prefixes, about 1% of the login IDs access the network from more than 10 distinct IP prefixes. Looking into those, we find that there are cases when an operator first logs on from a gateway server to a router, and then logs on to other routers from that router. The loopback IP address[1] of the first hop router appears as the origin IP for the second access session. While such "stepping-stone" access sessions are not common, they do occur – operators use this either for convenience or under certain network conditions, for example, when direct access to the other routers is unavailable. We can tighten our rule to deal with this situation by excluding sessions which originate on a router or switch. This removes "stepping stone" sessions from the analysis. The solid line in Figure 4 plots the number of distinct origin IP prefixes against the rank of login IDs – to protect proprietary information, we normalize the rank of login IDs to be between 0 and 1. We find above 78% of users have only one (non-stepping-stone) origin IP prefix and no one logs on from more than 4 distinct IP prefixes. This indicates that there exists a strong stability in the access pattern characterized by login ID and origin IP prefix combination – deviating from it can be a symptom of attacks. Figure 4 also plots the stability of this access pattern month by month – the shaded area indicates that the same login ID and IP prefix association has appeared in the preceding month. This demonstrates strong predictability based on past access behavior pattern (the unshaded area is mostly due to new users or infrequent users who only access the network in the second month).

Going back to the "stepping-stone" sessions, by matching the *ssh* command on the first hop router and the remote login request on the second router, we can reconstruct the chain of stepping-stones. Figure 5 plot the distribution function of the length of these chains and the outbound fan-out of these chains. It is evident that both attributes are bound by a small number (e.g., 7) in normal operational activities. In contrast, an intruder working from a compromised router may attempt to gain information from a large number of other routers, which is likely to produce long chains or high fan-outs. Watching those attributes closely can be an effective way to catch the intruder.

We next turn to the association between the networking device and the IP prefix from which management activities originate. The solid line in Figure 6 plots the number of

[1] Address assigned to a virtual interface commonly used for network management.

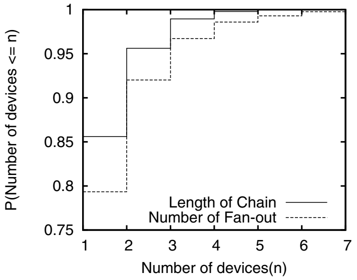


Fig. 5. CDF of the number of routers in a "stepping-stone" chain

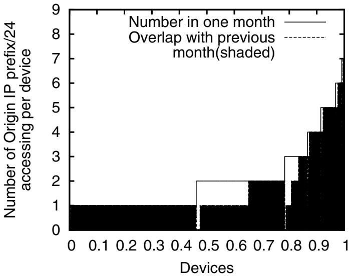


Fig. 6. Stability of association between the routers and origin IP prefix

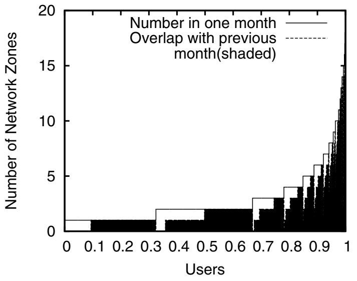


Fig. 7. Stability of the login ID and network zone association

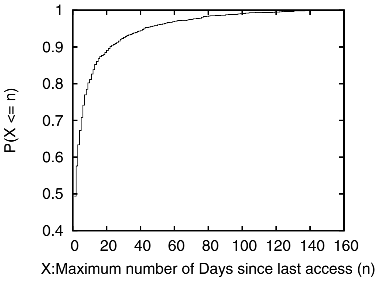


Fig. 8. CDF of the maximum gap in days between consecutive logins per ID

distinct origin IP prefixes versus the rank of the networking device IDs. As in Figure 4, we normalize the ranks to be between 0 and 1 to protect proprietary information. We find that 48% of the routers are only controlled from hosts within one /24 IP block during a one-month period. These control activities are likely routine network auditing and health monitoring. A small portion of the network devices are managed from a small number of (e.g., 2-7) IP prefixes, which correspond to the network operation centers (NOCs) responsible for those devices. Furthermore, some cross-region access can be unexpected in normal operations – it is suspicious if an operator from a regional NOC in Japan requests access to a router serving IPTV in the USA. Catching abnormal associations between routers and origin IP prefixes can be an effective way of identifying such cases. The shaded area in Figure 6 shows the overlapping associations that have appeared in the preceding month; this demonstrates the predictability of these associations, as the overlapping is very significant.

Finally, we examine the association between login IDs and network devices. Many users or software tools have limited scope in terms of the networking devices managed. The solid line in Figure 7 plots the number of distinct network zones (described in Section 2) that each login ID has accessed in one month. We again normalize the x-axis to avoid disclosing the size of the operator work-force. We observe that the majority of login IDs have a very limited scope (e.g., less than 3 zones) while a few of high-tier operators or software tool IDs access many zones. The stability of the login IDs' access pattern is depicted by the month over month comparison shown in the shaded area. We observe a strong predictability that can be utilized for detecting intrusions or misuses.

3.3 User Behavior

As mentioned in Section 2, different login IDs (corresponding to different operators or network management tools) have different roles/functions. Each user is likely to have a roughly stable behavior in access schedule (frequency), type of control (e.g., monitoring, or troubleshooting, or configuration change), and class of commands (e.g., SONET controller settings versus ACL configurations). Significant deviation from normal behavior can be a symptom of an account becoming compromised and an intruder impersonating the owner of the login account. In this subsection, we examine the properties of such user behaviors exhibited in normal network management activities.

We first examine the inter-session time distribution. Figure 8 plots the maximum difference in days between two consecutive logins from the same ID in a six-month period. We observe a wide variability among different login IDs. Many login IDs access the network on a regular basis, with a gap of at most a few days. But there are a considerable number of login IDs that only access the network occasionally. This suggests that it may be helpful to profile login IDs in different groups according to their access frequency.

Figure 9 shows the CDF of the average number of login sessions per login ID per day in a representative month. Here we exclude the days when the login ID is not active from the average statistic. The tail part of the curve, which goes several order of magnitude larger, is cut off so that we do not disclose the exact number of devices in the network. We observe that the majority of the login IDs have only a few login sessions per day. For example, 65% of IDs log onto the network no more than 5 times daily (on average). There are also many software tools and network management scripts producing over a hundred login sessions on daily basis. A login account suddenly changing its behavior, especially from having a small number of login sessions daily to a large number of them on a given day, is unusual or abnormal behavior and should be examined to see if it indicates a problem. Similarly, Figure 9 also shows the CDF of the average number of distinct networking devices accessed per login ID per day. Compared to number of sessions, it shows even more concentration – 65% of IDs log on to no more than 2 networking devices daily (on average). The tail portion of the curve again is dominated by software tools monitoring a large number of devices regularly, such as network configuration auditing tools. A surge in the number of distinct networking devices that a user initiates in a short period of time might be an intruder scouting for information. To understand the variability on this metric, Figure 10 shows a scatter plot of the coefficient of variation (CV) versus the mean – each point represents one login ID. We find that most of the CVs are bound by a small number (e.g., 3), while the login IDs with large number of average daily device accesses typically have much smaller CVs – suggesting that they can be more tightly bounded.

The set of router control commands and configurations used by a login ID is expected to exhibit some stability too. For a login account used by a software tool, the set of commands is determined by programming and rarely changes. For an operator's login,

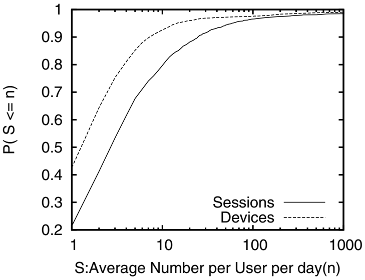


Fig. 9. CDF of the mean number of sessions initiated per login ID per day

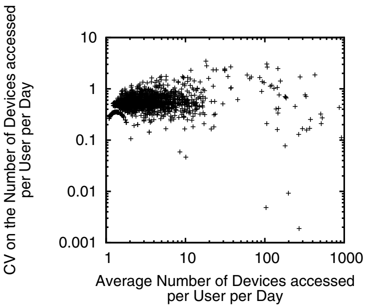


Fig. 10. CV versus mean of the number of distinct devices accessed per login ID per day

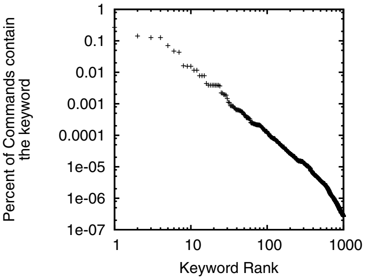


Fig. 11. Keywords Frequency Distribution

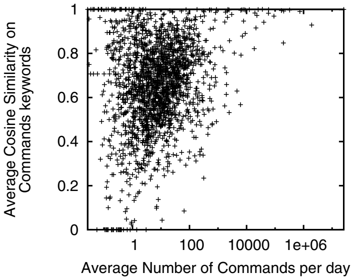


Fig. 12. Average Cosine similarity on the keywords frequency per login ID

the subset of commands should be subject to his/her privilege level and tightly related to his/her job role. However, extracting the exact association between login ID and the subset of the commands from knowledge about network operations is a challenging task – for example through code analysis of the software tools or meticulous review of all operation job functions.

In contrast, we take an approach that is based on historical data analysis and is detached from the semantics of router control commands, as follows: (a) we tokenize the commands (i.e., separate words in the command by white space); (b) we consider the tokens that contain any number as parameters (e.g., IP address) and remove those tokens; (c) we remove any non-alphabetic characters in each of the tokens and convert the remainder into lower case letters – we will refer to these as the *keywords*; (d) we profile each user with the set of keywords used.

Figure 11 shows the likelihood that a keyword is present in a command (sorted in decreasing order) based on one month of logs. We observe a strong skewness in the distribution, which can be well modeled by Zipf's distribution. The high ranked keywords are those used in monitoring network health (e.g., ping, vrf, show). And most of the bottom ranked ones are some arbitrary tokens (such as customer name) referenced in the description field of certain router configurations or some misspellings (due to typos

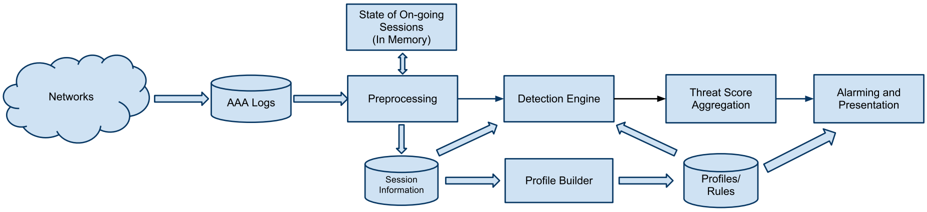


Fig. 13. Detection System Architecture

by operators) of other keywords. It is sufficient to keep track a subset of keywords (e.g., top 1000) and represent the remainder simply as the *other* keyword. Figure 12 shows the stability of the use of keywords per login ID, which plots the average cosine similarity of the keyword frequency distribution comparing one day against the previous active day. Login IDs with a high number of daily log entries trend to have high predictability one day to another. Deviation from the regular command keywords, especially for a software tool account, can be a symptom of an intruder impersonating the owner of the account.

4 Design of an Online Intrusion Detection System

Based on the analysis from the previous section, we design an online intrusion detection system that oversees the network management activities of the ISP network and detects and alarms on anomalous patterns. Figure 13 shows the system architecture. We collect logs from AAA servers in near real-time. The logs are fed into an online preprocessing module, which extracts critical information and updates on an entry by entry basis the running states of sessions, login IDs, origins, and commands that are required for the different intrusion detection rules. Periodically (e.g., once a day) the running states are fed into an offline profiling module in which the different profiles required by the rules are updated – the initial profiles can be constructed via offline analysis of an extended period of historical data. The online rule checkers examine the running states against the profiles and rules and tag the corresponding log entries with a threat score. An aggregation module then sums the threat score in a window according to the login ID or origin IP. Finally, an alarming and presentation module makes the information available to network security operators.

4.1 Domain Knowledge-Based Rules

We first define a set of rules that is specific to the network under study. We maintain a list of the IP address blocks that belong to the ISP network and check the origin IP of each AAA log entry against the list. An IP address from outside of the network indicates a breach of the ISP's periphery protection, and consequently the log entry is given a high threat score.

We also track the timestamp of the last login failure from each origin IP address and if a new failed login attempt is observed within T_1 seconds we update the timestamp and increment the count of consecutive login failures for the origin IP. Once the count exceeds a threshold N_1, we output the entries of these login attempts and assign a threat score to each of them. The timestamp and failure counts are reset when a successful login from the origin IP is made or the timeout T_1 is exceeded. With such a rule in place, an intruder that attempts to stay under the radar has to significantly slow down its attack, reducing the efficacy of the attack and prolonging the exposure.

There is also a rule in this category for stepping stone sessions. We trace stepping stone accesses as they occur and when the length of the access chain becomes greater than a threshold N_2 or the fan out becomes greater than N_3, we assign a threat score to the sessions involved.

Finally, as a countermeasure to sophisticated intruders hiding themselves by disabling AAA logging right after gaining access to the network elements, we assign a high threat score to any log entry that modifies AAA logging settings (e.g., `tacacs-server` command on Cisco IOS). In the ISP network, changes to AAA logging settings should not happen in normal circumstances, so this rule should never generate a false trigger.

4.2 Rules Based on Access Pattern Profiles

In our daily association profile we keep track of the following attributes: (1) origin IP prefix (2) login ID (3) $\langle$ login ID, origin IP prefix $\rangle$ (4) $\langle$ login ID, device zone $\rangle$ (5) $\langle$ origin IP prefix, device zone $\rangle$. For each entry we track the most recent date of appearance and the cumulative number of appearances. We delete an entry when the most recent appearance is more than T_2 (e.g., 180) days and add new entries to the long term profile once their count is sufficiently large.

We assign a threat score to sessions that do not match the existing profile. Note that if a session is from a new origin IP or new login ID, we do not include the threat score due to the lack of associations in (3), (4) and (5). The weight of the threat score of new associations of (3), (4) is set to be higher as the cumulative count of the login ID increases – our confidence to assert suspicious activities increases with more history data. Similarly, the weight of the threat score for (5) increases when the cumulative count for the corresponding origin IP prefix increases.

4.3 Rules Based on Statistical Models of the Access Profile

We track the mean and variance of the following attributes: (1) daily number of sessions per login ID, (2) daily number of distinct routers accessed per login ID, and (3) daily frequency count of command keywords per login ID for top N_4 and the *other* keywords.

We use the EWMA (Exponentially Weighted Moving Average) algorithm in estimating the running statistics for attribute X on day t:

$$Mean_t = \alpha X_t + (1-\alpha)Mean_{t-1}$$

$$Var_t = \alpha(X_t - Mean_t)^2 + (1-\alpha)Var_{t-1}$$

When computing the daily average, we exclude the case where the corresponding attribute is zero on day t – for example, when the user is inactive on the day.

If at any time, the daily cumulative counts reach or exceed a pre-calculated threshold for the attribute, we will assign a threat score to the access sessions involved.

The thresholds are determined as follows. Since the login IDs that have a high number of daily sessions (i.e., the highly active accounts) exhibit low variability as shown in Section 3.3, we set the thresholds in the same way as anomaly detection in Gaussian random variables: Threshold = $Mean + N_4 \times Var^{1/2}$. For the login IDs that have a moderate number of daily sessions, we set the threshold to be the product of a constant factor and the mean value: Threshold = $Mean \times N_5$. This is based on the observation that the coefficients of variation are bound by a small constant. And finally for the large portion of login IDs that only access the network occasionally, we set the threshold to be a small constant: Threshold = N_6. Once an attribute exceeds the defined threshold, a threat score is assigned. The value (weight) of the threat score is set according to a sublinear function of the corresponding attribute (the daily cumulative count) value.

4.4 Aggregation of Threat Scores

Using the rules above, in the course of a day, we maintain an updated set of AAA log entries that are assigned non-zero threat scores together with the rule triggered – to aid further examination by network security operators. Suspicious log entries can be noise (i.e., triggered by abnormal activities of low interest to security) – for example, an operator starting to use a new set of commands can trigger a violation detection by the user-keyword-rule. To reduce the chance that a network security operator has to investigate a non-critical violation, we further aggregate these log entries by login ID and by origin IP. The idea is that real attackers may be caught by multiple rules and by aggregating the threat scores on a per login ID or per origin IP basis they can be further distinguished from non-critical anomalies.

To achieve this, we use a moving window of T_3 (e.g., 1 day), and sum up the threat score within the window for all login IDs and origin IPs. We then set a threshold N_7 based on historical data. When we observe an aggregate threat score exceeding N_7, we generate an alarm to the network security operators. We also display all suspicious activities on a dashboard report from which network security operators can pull information on demand.

5 Evaluation

We evaluate our system from two perspectives – the rate of anomalies detected from day-to-day network management activities and the effectiveness of detecting artificially injected anomalous activities. The former quantifies the resources required to investigate potential misuses and intrusions. The latter quantifies the chance that an anomaly goes undetected by our system.

5.1 Running System Performance

Figure 14 shows the distribution of the aggregate threat score in a month using two types of aggregations – by login ID (solid line) and by origin IP address (dashed line). We observe that about 93% of login IDs and 84% of origin IPs pass the system without raising

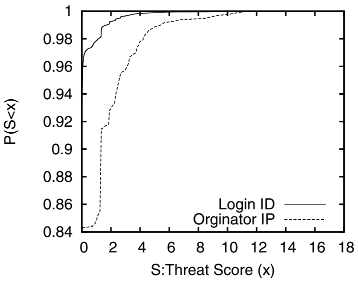


Fig. 14. CDF of aggregate threat score by login ID in one month

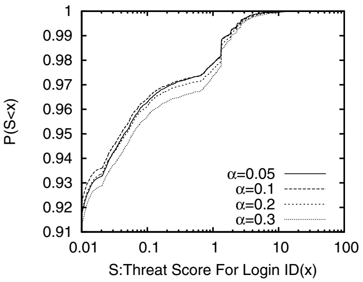


Fig. 15. CDF of aggregate threat score by login ID in one month w.r.t. different α values

any threat score. Meanwhile, there exist a small number of cases in which the system reports a high threat score. By manual inspection, we find most of them correspond to unusual network changes such as a newly deployed network management center or a major software upgrade on an existing network management tool. We will see an example of this in Section 5.3. We also find that a non-negligible fraction of cases, mostly with a relatively low threat score (e.g., less than 4), correspond to a small number of log entries from either a newly enabled or a very infrequently used account. Profiles for such accounts are difficult to construct based on history, and they typically do not generate many activities to drive the threat score high. We will propose and evaluate a system enhancement to further reduce the false alarms due to new user accounts in Section 6.

Figure 14 defines the tradeoff curves between the alarming rate and the sensitivity to anomalous activities. Raising the alarm threshold (the N_6 in Section 4.4), reduces the number of cases that security operators have to investigate but also reduces the chance of catching a stealthy intrusion. For a concrete example, setting N_6 to 5 would produce a few alarms per week on average, which is quite manageable for the network security operators.

We note that the above N_6 and several others as described in Section 4 are parameters used in the system. We do not present the exact values for the parameters in our running system due to security considerations. Instead, we show through an example our reasoning on parameter selection. Figure 15 shows the solid line in Figure 14 with a varying α value used in the EWMA estimate. Note the x-axis is in log scale. Different α values effectively factor in different amounts of history data. Setting $\alpha = 0.05$ effectively ignores (e.g., weight less than 0.01) data more than 90 active days old while $\alpha = 0.3$ effectively ignores data more than 13 active days old. However, Figure 15 shows there is little difference in the threat scores among the four different α values – indicating that a short history is sufficient for the system.

5.2 Controlled Experiment

Due to the lack of real attack observed in the network, we manually generate a set of data with anomalies for the performance test. Using Figure 14 as a reference point, we design a controlled experiment as follows. We first randomly select 50 pairs of non-overlapping login IDs. We then take one day's worth of AAA logs from our running system and substitute the login ID field in the log entries such that the two login IDs in each of the chosen pairs are switched. Finally, we feed the manipulated AAA logs into our running system and monitor the output.

Figure 16 presents the CDF curves of the aggregate threat score based on the original AAA logs (dashed line) and the synthetic data (solid line) respectively. We find that our system is able to detect many of the behavior changes introduced by the login ID swapping. 72 out of the 100 login IDs report non-zero threat score and among them, 30 login IDs have a threat score higher than 10. This result highlights the sensitivity of our system – we do not expect one user impersonating another can always be caught (for example two operators with the same role may hardly be distinguishable), but compared to the baseline threat score distribution under normal operation, the synthetically generated ID-swappings stand out significantly.

For a closer look at how our system detects anomalous behaviors, Figure 17 and Figure 18 compare the contribution to the threat score from rules based on access pattern profiles with the contribution from rules based on statistical models of the access profile respectively. For each of the 100 login IDs, we plot their threat score against the difference in the daily average access frequency to the substituting login ID in the AAA logs. For example, a login ID *abc*, with 16 sessions per day on average, which is replaced by login id *xyz*, with an average of 1024 sessions per day, would have its threat score plotted at 6 (i.e. $\log(1024) - \log(16)$) on the x-axis. We observe that both the access pattern changes and the access statistics changes have contributed to the high threat score. The higher the difference in the amount of access activities between the pair of swapped login IDs, the higher the resulting threat scores – with the exception in Figure 18 on the negative side of the x-axis. The exception arises because the rules based on the statistical models are one-sided, i.e., we do not alert on a "busy" user suddenly becoming less active, as this behavior change does not seem to pose any security threat.

5.3 Operational Case Studies

We now look at an example in which our system alerted with a high threat score. Figure 19 plots the aggregate thread score of a particular login ID over the course of four days. The login ID is used by a software tool that periodically initiates `ping` commands among the various provider edge (PE) routers of the VPN customers to monitor their VPN health.

Starting in the afternoon of day 2 of the plot, we observe a fast increase in the threat score by the login ID. In less than two hours, the threat score passed the 99.5% alarming threshold and kept rising. It turned out that the software tool was upgraded that day and the new control sessions included a `show version` command that collects the router OS version across the network – similar to what might occur if an intruder attempted to collect information as preparation for attacks. After validating the change of behavior due to software upgrade, we included the pattern change in the profile update at the end of day, which greatly reduced the threat score on day 3. The corresponding statistical models were further updated at the end of day 3 and the new pattern then got fully captured by the profiles. Hence, there was no more threat score on day 4.

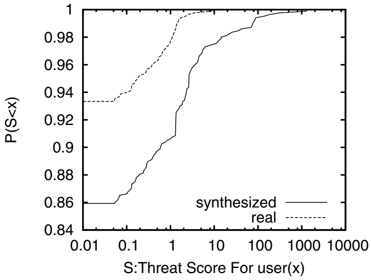


Fig. 16. CDF of aggregate threat score on real data and synthesized data

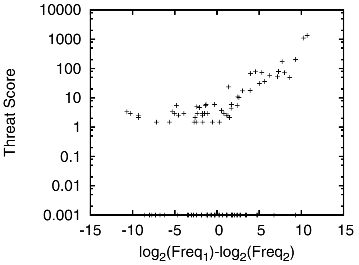


Fig. 17. Threat score by rules based on access pattern profiles

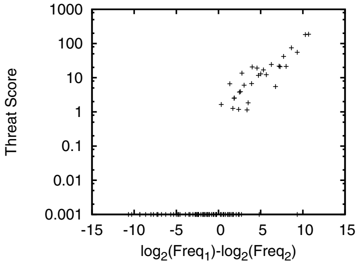


Fig. 18. Threat score by rules based on statistical models of access profile

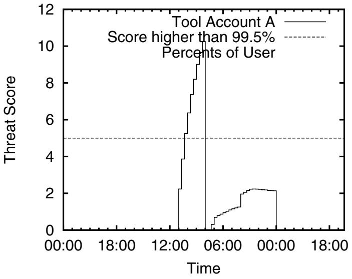


Fig. 19. Threat score For Tool account A

We next take a look at an example where a system alert was triggered due to a new user appearing in the system. In this case, a new operator X started a session to a device between 1:30 and 1:59; since this was a newly created login ID, bearing no historical association to either the network zone or the origin IP, the session produced a threat score of 1.5 at the end of the session. Later on in the day, X logged on to four more devices in the same network zone between 3:20 and 3:40 in 21 different login sessions, pushing the threat score to 7.7 and triggering an alarm. A network security manager examined the alarm and checked X's information from a corporate directory service. The threat was quickly dismissed when it was learned that X was a new member of the organization that manages the network zone in question. We will show next how we take the operational experience acquired in such scenarios and enhance our system specifically for newly provisioned login IDs.

6 A System Enhancement Handling New Login IDs

Operational experience has provided a valuable insight – when investigating a threat alert regarding a user login, network security officers often depend on the user's information in the corporate directory in addition to the access traces and profiles from our system. Inspired by this, we look into how to utilize the corporate directory information to further reduce the false threat alerts.

6.1 Quantitative Analysis

We first create an automation capability that queries the corporate directory service for any given login ID. In particular, we obtain the corporate hierarchy information of the owner of the login ID by following the management chain up until reaching the president of the ISP. We then examine how users across the corporate hierarchy relate in terms of network access patterns and behaviors.

Figure 20 plots the CDF of the profile distance with respect to the access patterns (Section 4.2 (3), (4)) – measured by the $l^2 norm$ of the difference in their profile vector between a pair of user login IDs. We define the corporate distance of a pair of users as the maximum number of hops for them to reach a common manager – two operators sharing the same direct supervisor would have a corporate distance of 1, and two sharing the same second-level-up manager would have a corporate distance of 2, etc. Different lines in Figure 20 correspond to user pairs of different corporate distances.

We observe that users that are organizationally close have similar access profiles. The further apart two users are in corporate distance, the more distinct their network access patterns are. When we focus on the case in which two operators are under the same direct manager (the solid line), we find that in 26% of the cases they have identical access pattern; and in 56% of the cases, their difference is no more than 3 (e.g., having access to 3 different network zones). This suggests that using the profiles of other team members can serve as a reasonable approximation when the operator's own history is not fully established.

Figure 21 and Figure 22 plot the CDF of the absolute difference in the daily average number of sessions (Section 4.3 (1)) and the daily average number of distinct devices accessed (Section 4.3 (2)) respectively. In both cases, we observe decreasing trends in the behavioral similarity as users' corporate distances increase, although the gaps are smaller compared to Figure 20. Focusing on the solid lines, we find that in 65% of the cases when users have a common direct reporting manager, their daily average number of sessions exercised differs by no more than 5 and their daily average number of distinct devices accessed differs by no more than 2. We have also observed a similar commonality in the profile of command token frequencies. These point to the good potential of bootstrapping profile building for newly provisioned login IDs using their peers' profiles.

6.2 Profile Bootstrapping

As described in the operational case examples, network security managers sometimes dismiss system generated threats based on the additional information regarding the job function of the operator. Ideally, if domain knowledge of all different job functions and their expected access profiles were available, we could incorporate it into the system. However, such domain knowledge is implicit, highly distributed across various organizations, and evolving over time. Hence it is operationally challenging to acquire and

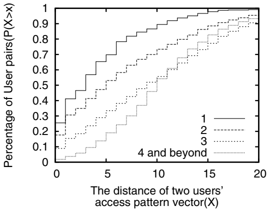


Fig. 20. CDF of access pattern profile distance for user pairs grouped by the corporate distance

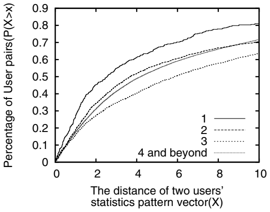


Fig. 21. CDF of average difference in daily session count for user pairs grouped by the corporate distance

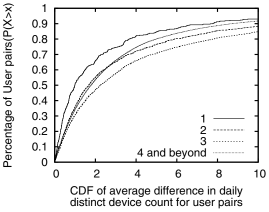


Fig. 22. CDF of average difference in daily distinct device count for user pairs grouped by the corporate distance

maintain. Instead, we propose to take a data-driven approach that does not require domain knowledge input, based on the analysis above.

Specifically, we organize user login IDs into classes based on the login owner's management hierarchy. We then construct class-profiles based on the profiles of the members of the class. When a new login ID appears in the AAA log, the system queries the corporate directory service to obtain the user class to which the new login ID belongs. The class profile is then used as the bootstrapping profile for the user. The bootstrapping profile is replaced by the user's own profile once adequate history has been observed for the login ID. This boostrapping process aligns in principle with the model training techniques using scarce data [13].

To construct the class-profiles, we take the median statistic on each dimension (i.e., a feature in the profile of login ID) among all users (u) belonging to the class (C). Specifically,

$$\text{Profile}^C = \langle f_1^C, f_2^C, ...\rangle, \text{where} f_i^C = \text{median}_{u \in C} f_i^u$$

This is applicable to both binary flags (e.g., whether a login ID and network zone association exists) or numerical values (e.g., average number of sessions per day). Using the median is known to produce an estimator of cluster centroid that is robust to outliers[9].

6.3 Effectiveness of the Enhancement

Comparing the threat alert rate with and without profile bootstrapping, for the same month of data from Section 5.1, we find that profile bootstrapping has reduced the threat alerts (using a threshold $N_6 = 5$) by 60% for login IDs with inadequate history (less than or equal to seven active days of history). The overall threat alert rate (for all login IDs) is consequently reduced by 17% with profile bootstrapping.

7 Related Work

Our work falls into the area of IDS (Intrusion Detection Systems) in computer and networking security, which dates back to 1980 when Anderson[1] first proposed a computer security surveillance system. Over time, the research area has become more active as the Internet grew in scale and application diversity and new security threats constantly emerged.

IDS broadly divide into two categories: host-based (HIDS)[7,8,4,10,15] and network-based (NIDS)[6,16,5,3] — HIDS typically rely on information about running processes to catch intrusions to computer host(s); NIDS typically analyze network traffic in order to detect attacks.

Another taxonomy of IDS is based on detection principles[17]: anomaly-based IDS (AIDS) [16,7,5,8,4,10,15] capture anomalous traffic or processes based on analysis of normal patterns. Signature-based IDS (SIDS) [11,14,3,6] use known signatures of attacks to alert on viral activities. Our work aligns with AIDS in principle.

Masquerader detection is a branch of IDS. A masquerader is an attacker who obtains a user's password, penetrates the access control system and impersonates a legitimate user. Lunt et al [7] designed IDES as the first IDS handling masquerader detection, using a simple yet effective statistical model. Recently, different machine learning techniques such as Genetic Algorithm [4], Naive Bayesian classification[10], and Support Vector Machine [15] have been applied in this area. In this study, we build user behavior models from access and command invocation patterns using statistical methods and alert based on deviation from the model. It remains as future work to evaluate whether more sophisticated machine learning algorithms can improve sensitivity and accuracy in our problem setting.

8 Conclusion

In this paper, we have studied the problem of protecting the networking infrastructure and the information available therein for large scale enterprise or ISP networks. We have proposed to enhance existing security measures with an intrusion detection system overseeing all network management activities. By analyzing device access logs collected via the AAA system in a global tier-1 ISP network, we have gained tremendous insights on the features that distinguish normal operational activities from rogue/anomalous ones. We have further developed a real-time intrusion detection system that builds statistical models to profile normal operational activities and alerts in real-time on any deviation from the profiles. Our evaluation demonstrates that this system effectively identifies potential intrusions and misuses with an acceptable overall alarm rate.

For future work, we would like to explore using more sophisticated machine learning techniques in addition to statistical methods to capture anomalous activities, using other network logs, and using other information such as network maintenance schedules to suppress alarms about "intended anomalies". We are also interested in further introducing automated mitigation control based on detected anomalies to the AAA system such that an attack or intrusion can be stopped as early as possible.

Acknowledgments. This research is supported in part by the National Science Foundation under NSF grant CNS-0904901.

References

1. Anderson, J.P.: Computer security threat monitoring and surveillance. Technical Report James P Anderson Co Fort Washington Pa, p. 56 (1980)
2. Carrel, D., Grant, L.: The TACACS+ protocol (January 1997)
3. Dreger, H., Feldmann, A., Mai, M., Paxson, V., Sommer, R.: Dynamic application-layer protocol analysis for network intrusion detection. In: Proceedings of the 15th conference on USENIX Security Symposium, vol. 15. USENIX Association, Berkeley (2006)
4. Iglesias, J.A., Ledezma, A., Sanchis, A.: Creating User Profiles from a Command-Line Interface: A Statistical Approach. In: Houben, G.-J., McCalla, G., Pianesi, F., Zancanaro, M. (eds.) UMAP 2009. LNCS, vol. 5535, pp. 90–101. Springer, Heidelberg (2009)
5. Krishnamurthy, B., Sen, S., Zhang, Y., Chen, Y.: Sketch-based change detection: methods, evaluation, and applications. In: Proceedings of the 3rd ACM SIGCOMM Conference on Internet Measurement, IMC 2003, pp. 234–247. ACM, New York (2003)
6. Li, Z., Xia, G., Gao, H., Tang, Y., Chen, Y., Liu, B., Jiang, J., Lv, Y.: Netshield: massive semantics-based vulnerability signature matching for high-speed networks. In: Proceedings of the ACM SIGCOMM 2010 Conference on SIGCOMM, SIGCOMM 2010, pp. 279–290. ACM, New York (2010)
7. Lunt, T.F., Jagannathan, R., Lee, R., Listgarten, S., Edwards, D.L., Neumann, P.G., Javitz, H.S., Valdes, A., Lunt, T.F., Jagannathan, R., Lee, R., Listgarten, S., Edwards, D.L., Neumann, P.G., Javitz, H.S., Valdes, A.: Ides: The enhanced prototype - a real-time intrusion-detection expert system. Tech. rep., SRI International, 333 Ravenswood Avenue, Menlo Park (1988)
8. Maggi, F., Matteucci, M., Zanero, S.: Detecting intrusions through system call sequence and argument analysis. IEEE Transactions on Dependable and Secure Computing 7, 381–395 (2010)
9. Maronna, R., Martin, R., Yohai, V.: Robust statistics: theory and methods. Wiley series in probability and statistics. J. Wiley (2006)
10. Maxion, R.: Masquerade detection using enriched command lines. In: Proc. of 2003 International Conference on Dependable Systems and Networks, pp. 5–14 (June 2003)
11. Paxson, V.: Bro: a system for detecting network intruders in real-time. Comput. Netw. 31, 2435–2463 (1999)
12. Rigney, C., Willens, S., Rubens, A., Simpson, W.: Remote authentication dial in user service, radius (2000)
13. Robertson, W., Maggi, F., Kruegel, C., Vigna, G.: Effective Anomaly Detection with Scarce Training Data. In: Proceedings of the Network and Distributed System Security Symposium (NDSS), San Diego, CA (February 2010)
14. Roesch, M.: Snort - lightweight intrusion detection for networks. In: Proceedings of the 13th USENIX Conference on System Administration, LISA 1999, pp. 229–238. USENIX Association, Berkeley (1999)
15. Salem, M.B., Stolfo, S.J.: A comparison of one-class bag-of-words user behavior modeling techniques for masquerade detection. Security and Communication Networks (2011)
16. Song, Y., Keromytis, A.D., Stolfo, S.J.: Spectrogram: A mixture-of-markov-chains model for anomaly detection in web traffic. In: NDSS. The Internet Society (2009)
17. Stefan, A.: Intrusion detection systems: A survey and taxonomy. Technical Report 99(Technical report 99-15), 1–15 (2000)
18. Suo, X., Zhu, Y., Owen, G.S.: Graphical passwords: A survey. In: Proceedings of the 21st Annual Computer Security Applications Conference, pp. 463–472. IEEE Computer Society, Washington, DC (2005)

A Lone Wolf No More: Supporting Network Intrusion Detection with Real-Time Intelligence

Johanna Amann[1], Robin Sommer[1,2], Aashish Sharma[2], and Seth Hall[1]

[1] International Computer Science Institute
[2] Lawrence Berkeley National Laboratory
ja@so.in.tum.de

Abstract. For network intrusion detection systems it is becoming increasingly difficult to reliably report today's complex attacks without having external context at hand. Unfortunately, however, today's IDS cannot readily integrate *intelligence*, such as dynamic blacklists, into their operation. In this work, we introduce a fundamentally *new capability* into IDS processing that vastly broadens a system's view beyond what is visible directly on the wire. We present a novel *Input Framework* that integrates external information in real-time into the IDS decision process, independent of specific types of data, sources, and desired analyses. We implement our design on top of an open-source IDS, and we report initial experiences from real-world deployment in a large-scale network environment. To ensure that our system meets operational constraints, we further evaluate its technical characteristics in terms of the intelligence volume it can handle under realistic workloads, and the latency with which real-time updates become available to the IDS analysis engine. The implementation is freely available as open-source software.

1 Introduction

For network intrusion detection systems (IDS) it is becoming increasingly difficult to reliably report today's complex attacks purely by looking at traffic on the wire, without having any further external context at hand. For example, often the best way to detect botnet communication is to monitor for connections to known C&C servers that the security community has already identified. Likewise, external malware registries can help determine if downloaded files contain malicious code. A variety of efforts are collecting and disseminating such third-party *intelligence* systematically, including blacklists such as Google's Safebrowsing URL list [11] and VirusTotal's hash-based malware identification [29]. More sophisticated federated sharing initiatives—operated, e.g., by REN-ISAC for the education community [18] and the Department of Energy's *Joint Cybersecurity Coordination Center* (JC3) [1]—enable real-time propagation of incident information across their member institutions.

Unfortunately, however, today's IDS cannot readily integrate such external information into their processing. Their standard approach for using intelligence remains to statically convert it into their rule languages, which severely limits the attainable benefits. If they offer direct interfaces to the external world at all, they typically restrict them to a small set of individual hard-coded applications.

D. Balzarotti, S. Stolfo, and M. Cova (Eds.): RAID 2012, LNCS 7462, pp. 314–333, 2012.

In this work, we introduce a fundamentally *new capability* into IDS processing that vastly broadens a system's view beyond what is visible directly on the wire. We present a novel *Input Framework* that integrates external intelligence in real-time into the IDS decision process, independent of specific types of data, sources, and desired analyses. We design the framework so that it offers a simple interface to IDS users while providing the flexibility to interface to a range of local and remote intelligence sources. On the architectural side, we ensure that even with rich external information, the Input Framework heeds to the stringent performance requirements of high-volume, soft real-time packet processing.

We implement the Input Framework design on top of the open-source Bro IDS. By offering a Turing-complete scripting language for expressing local policies, Bro is ideally suited to exploit the full power of the new capability. Using our implementation, we demonstrate three real-world use cases: *(i)* integration with REN-ISAC and JC3 feeds; *(ii)* online virus checks of executables observed on the network; and *(iii)* real-time database queries, with results integrated back into the IDS decision process on the fly.

The goal of our work is to provide a new capability suitable for operational deployment. As such, it is crucial to ensure that interface and implementation meet operational demands, and we hence evaluate our work from that perspective. First, we report operational experiences from a large-scale network environment where operators are already deploying the implementation experimentally. Second, we instrument our implementation to understand its technical properties, such as the volume of intelligence it can handle in parallel to processing traffic, and the latency with which updates become available to the IDS analysis.

Based on encouraging feedback from operators, we anticipate that our implementation will become part of operations at further sites in the near future. We release our code as open-source under a BSD license, and most of it is already integrated into the standard Bro distribution. We emphasize, however, that conceptually our approach is not limited to the specific IDS we used, but can similarly enable others systems to leverage intelligence effectively.

We structure the remainder of this paper as follows. §2 discusses related work. §3 presents the design and architecture of the Input Framework, and §4 details our implementation. §5 discusses three concrete use cases, including initial deployment experiences with one of them at a large-scale network site. We evaluate the performance of our implementation in §6, and we finally summarize in §7.

2 Related Work

The Input Framework provides a generic platform for integrating external intelligence into an IDS' live packet processing. While we are not aware of any current IDS that provides such a capability with a similar degree of flexibility, a number of existing efforts provide evidence for the utility of our approach.

We find a wide variety of online services available that provide third-party intelligence to network sites aiming to support their security efforts. Most commonly, these offer blacklists of known bad actors or content. Examples include web sites listed on Google's Safebrowsing URL list [11]; mail servers on Spamhaus's Block List (SBL, [23]); malware listed by registries like VirusTotal [29] and Team Cymru's malware hash registry [7]; and suspicious IP addresses reported to DShield [10]. Whitelists alternatively help avoiding false positives; NIST for example provides a list of hashes of known *benign* files that are part of OSs and applications [2]. More general data sources can provide further context like *whois* domain information and Team Cymru's *Bogon List* of unroutable IP space. In addition to such public services, a number of closed federations have emerged that distribute non-public incident information across member institutions. This information is much more context rich than the simple aforementioned blacklists, often containing features like the IP address, URL, downloaded malware md5 hashes, and timestamps for each incident. Examples include REN-ISAC's *Security Event System* [18], DOE's JC3 feeds [1], and Argonne Lab's *Federated Model* [6]. In §5 we show how our Input Framework integrates with the former two specifically. A particular benefit of such federations is that sites with lower technical expertise can benefit from findings and capabilities of their peers. In addition, a site may also have further local resources to support an IDS: a database of valid user accounts can help detecting brute-force SSH attacks, and a list of software running on local end hosts suggests whether a victim is vulnerable to a specific exploit. It is generally all such context information that we collectively refer to as *intelligence* .

Past studies show the benefit of integrating external information into security decisions. A recent study [19] at NCSA found that for 27% of all tracked incidents *external notifications* triggered their investigation (and not the local IDS). Verizon reports that "third parties discover data breaches much more frequently than do the victim organizations themselves" [28]; they found 92% of all breaches to fall into that category (49% when only considering larger organizations). It is such experience that motivates federations like SES to automate intelligence sharing. Another study [14] shows that attacks on different sites are often correlated and hit the separate networks within minutes. The authors recommend to rapidly share IDS state as a countermeasure. In [20], the authors analyze e-mail spam blacklists and find that local aggregation and reputation assignment can improve their accuracy. Our approach aligns with that by making complex intelligence available to the IDS and not only working on blacklists with predetermined yes/no decisions. We also find a number of specific detectors in the literature that leverage intelligence as key ingredients, such as BotHunter [12].

Current IDS do not provide flexible mechanisms to integrate external information. In our experience network operators today leverage intelligence by writing scripts that either turn it into static IDS configurations or post-process the *output* of an IDS offline. Indeed, Snort [17] distributes most of its blacklists in the form of rules [4]; and the software underlying SES provides an option to directly output Snort rules generated from received intelligence [5]. Likewise, users of Suricata [24] and Bro [16] often auto-generate static configurations. Doing so however tends to incur major performance hits, and also makes updating an expensive operation that typically requires a restart. Worse, with signature-based systems this approach constraints any analysis to basic byte-level pattern matching—a model rarely constituting a good fit for higher-level intelligence that often only *augments*, and not controls, security decisions.

Newer Snort versions feature an IP reputation preprocessor [21] that directly imports IP-based black- and whitelists. It, however, requires specifically formatted input, does not operate on other information than IP addresses, and cannot leverage the lists for any analysis going beyond simple allow/drop decisions. Bro provides a generic communication interface [22] that can update state dynamically. However, while a user could use this for integrating intelligence (and some do, for a lack of alternatives), it remains low-level with no specific support for interfacing intelligence sources.

In the literature, work on adding context to IDS decisions tends to focus on correlation between IDS nodes (e.g., [8,26,27]), not higher-level intelligence sharing with external entities. On the commercial side, many security appliances seem to leverage forms of intelligence. For example, Symantec's firewalls support blacklists and blacklist sharing [25], and Damballa's product line includes *dynamic reputation* modules [15] based on Notos [3]. While taking a similar approach as we advocate, the Input Framework is not limited to a specific data source or analysis. Generally, we note that for commercial solutions it tends to be hard to say what they do *exactly*, as specifics of their internals are rarely public.

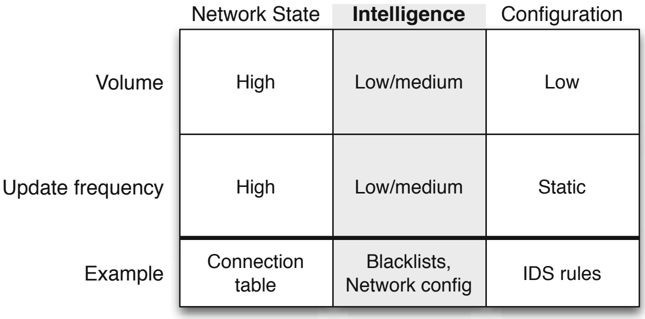
3 Design

We now discuss the design of the Input Framework. We begin by looking at the type of state that it targets in §3.1: external "intelligence" of low to medium volume with potentially frequent updates. We discuss our main design objectives in §3.2 and then present the high-level Input Framework architecture in §3.3.

3.1 Intelligence

Of all the run-time state that a typical IDS manages, the Input Framework targets a specific subset that today's systems support only insufficiently. Most IDS implementations focus on two groups of state (see Fig. 1): *(i) network state* derived directly from the monitored packet input, and *(ii) configuration state* describing the types of analyses to perform, such as a set of signatures or specific hosts to watch out for. The former group consists of volatile, high volume data (e.g., the current set of active connections along with TCP and application-layer information), and requires sophisticated schemes for efficient management [9]. The configuration of an IDS, on the other hand, is of low volume and static: changes tend to require an expensive reload operation that interrupts the current analysis, often in the form of a full system restart.

We argue, however, that there is a third group of state that we term *intelligence* state: externally provided context that, when correlated with the traffic on the wire, can significantly increase the system's detection capabilities. As discussed in §2, such state includes blacklists of known bad actors and specifics of the local environments. Conceptually, intelligence falls in between the two former groups: it is of much lower volume, and more stable, than network state.



	Network State	**Intelligence**	Configuration
Volume	High	Low/medium	Low
Update frequency	High	Low/medium	Static
Example	Connection table	Blacklists, Network config	IDS rules

Fig. 1. IDS State

Intelligence however changes frequently, possibly multiple times per second, and thus cannot become part of the IDS's static configuration. Our Input Framework specifically focuses on integrating such state into the IDS analysis.

3.2 Objectives

The goal of the Input Framework's design is to offer a flexible mechanism to integrate intelligence from a variety of sources, without negatively impacting the IDS' main task of analyzing a high-volume packet stream under soft real-time constraints. To this end, we identify the following objectives for its design:

Adaptable to Different Sources. A crucial design goal is the ability to interface to a range of potential input sources and formats. We require the Input Framework to accommodate sources as different as flat files of ASCII and binary data, sockets for live feeds, databases, and web services. Along with that comes the requirement to support different modes for updates, including processing intelligence in regular batches as well as "pull" and "push" operation for real-time streams. While adding new input sources necessarily requires tailored interface code, the Input Framework should make extensions easy.

Simple, Yet Flexible User Interface. The interface that the Input Framework exposes to the user should be easy to use and concise, with reasonable defaults where it offers options. It needs to provide a unified view of all input sources, abstracting from their individual characteristics. As intelligence will originally arrive in a variety of formats that external parties determine, we need to provide customization hooks that allow for on-the-fly preprocessing and filtering. However, all external intelligence should fully integrate with the IDS' standard analysis capabilities. Where possible, we want to transparently incorporate intelligence into the existing decision process.[1]

[1] The specifics here depend on the capabilities the IDS provides. For example, for a typical signature-based IDS it should be straight-forward to adapt the rule language for doing simple match/no-match decisions derived from external blacklists. However, it will be challenging to use such lists as a reputation indicator that only *contributes* to a decision if such a concept does not already exist in the system.

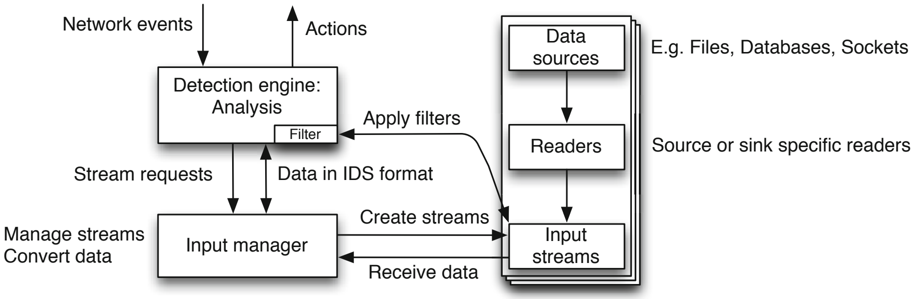


Fig. 2. Framework Architecture

Asynchronous Operation. All I/O must execute fully asynchronously. Accessing external state can take a significant amount of time during which packet processing needs to proceed normally without blocking. This is particularly crucial for high-latency sources such as databases or web services hosted remotely. Likewise, as a stream of intelligence is coming in, processing must interleave with traffic analysis to avoid causing packet drops. Nevertheless, from the analysis' perspective, the state must be consistent at all times.

Real-Time Operation. The Input Framework needs to make incoming intelligence available to the IDS analysis rapidly. While we cannot control lags introduced by an external source (like the time it takes a database to respond to a query), we strive to keep the Input Framework's internal latency low.

3.3 Architecture

Fig. 2 shows the architecture for the Input Framework that we design to address the above objectives. In the following we discuss the frameworks main components in an order roughly following the data flow. As some parts of the architecture depend on the specific IDS that one integrates with, we can only sketch them in abstract terms. However, the discussion will become more concrete in §4 where we describe our implementation inside a specific IDS.

When integrating the Input Framework, the IDS' core *analysis and detection engine* remains mostly unchanged but gains the capability to specify external intelligence sources that it wants to access. From the engine's perspective, each intelligence source corresponds to a *stream* that it creates on-demand as its configuration defines. When opening a stream, it passes along the necessary control information such as type of input (e.g., file, database), location (i.e., a filename or a remote socket), as well as the expected layout of the data that the stream will later forward. For the latter, we represent all intelligence in a unified, column-based format and pass a description along with the stream request.

The Input Framework's *manager* is the central interface between analysis engine and external intelligence sources. It receives the engine's request to open a stream, spawns a new *reader* instance, and instructs it to connect to the corresponding source. We differentiate between types of readers: a *file reader*, e.g., reads from local files, and a *database* reader queries a remote database.

The readers forward all intelligence to the manager, which passes it on the analysis engine. However, rather than directly making it available, data from the readers first passes through an optional set of *filters* that may reduce and transform the input before it gets applied. As the filters run inside the analysis engine, they have access to the full IDS state.

Generally, each reader decides on the model for forwarding input to the manager. A file reader could, for example, read a file once at startup and then keep monitoring it for changes in regular intervals, passing updates on as noticed. On the other hand, a reader connecting to a real-time network feed would instead forward intelligence immediately as it arrives on its input socket.

In our architecture, manager and readers communicate via a simple API that is fully decoupled from the core of the IDS. This make it particularly easy to add new readers as they are not at all concerned with the system's potentially complex internals. In principle, one could even connect a single manager implementation to different IDS implementations just by adapting the upstream interface accordingly. On a technical level, decoupling the readers makes it straightforward to run them in separate threads, which simplifies the implementation of asynchronous I/O. With that, the only critical point potentially impacting the IDS' packet processing remains the manager/engine interface.

4 Implementation

We implement the Input Framework architecture on top of the Bro IDS. By providing a Turing-complete scripting language for expressing custom detection policies, Bro fits well with the capabilities that the Input Framework offers: we add a new script-level API that allows users to configure external intelligence sources, which Bro then maps transparently into standard data structures. In the following, we discuss the main aspects of our implementation in terms of its internal structure (§4.1) and its user interface (§4.2).

4.1 Integration

Fig. 3 shows how our implementation integrates into Bro.When processing network input, Bro internally reduces the voluminous stream of packets to a series of higher-level network *events* that reflect the key steps of the underlying activity.[2] A *policy interpreter* then executes scripts written in a specialized, high-level language[3] that expresses both a site's custom security policy and general forms of high-level analysis (e.g., scan detection [13]) in terms of the event stream. A crucial point is that these events are *policy neutral*: Bro itself makes no judgment

[2] Bro provides both generic transport analysis and application-specific analysis. It understands for example specifics of HTTP, DNS, SMB, and many other protocols.

[3] "A domain-specific Python."

as to whether the events reflect malicious or benign traffic but rather leaves that determination to a user's custom scripts.

As layed out by our general architecture (see §3.3), a central manager acts as the interface between the Bro core and intelligence sources. The manager spawns separate reader threads for each source. When reading data, these readers pass it on to the manager via thread-safe queues. The manager then feeds the information into either the event stream or directly into the policy interpreter by adding to its data structures, executing filters, and calling script layer functions. We detail all of these further below.

The interface between the manager and Bro's core is performance critical as it needs to trade-off processing incoming intelligence with that of network traffic. Generally, the latter receives priority as its volume prevents Bro from buffering packets to any significant degree. However, to satisfy our real-time constraint, incoming intelligence cannot just wait for lulls in the traffic stream (which in most environments will never occur) but instead propagates incrementally, interleaved with traffic analysis. Internally, Bro already provides an I/O loop structure that allows to balance packet processing with further asynchronous input. The input manager hooks into this loop structure as an additional data source.

4.2 User Interface

Our Input Framework implementation integrates fully into Bro's domain-specific scripting language. In the following, we walk through the main parts of the script-level interface that the Input Framework exposes to the user. As a simple running example, we consider importing a blacklist of hosts, formatted as a 3-tuple (*IP address*, *reason*, *timestamp*) where *IP address* is the host's address, *reason* a textual description of the host's offense, and *timestamp* a Unix-timestamp indicating since when the list entry exists. Stored in a tab-separated file, the list could look like this:

```
ip              reason                timestamp
66.249.66.1     connected to honeypot 1333252748
208.67.222.222  too many DNS requests 1330235733
192.150.186.11  sent spam             1333145108
```

Reading Files. The Input Framework can directly import files such as the above into *tables*, a associative array type that Bro's scripting language provides, much like hashes in Perl. To do so, the user declares the columns to extract from the file by defining two corresponding *record* types (records are similar to *structs* in C): one for the table index and one for its values. In our example, assuming we want to use the IP address as the table index and the other two columns as its value, we can define the following types:

```
type Index: record { ip: addr; };

type Value: record { reason:    string;
                     timestamp: time; };
```

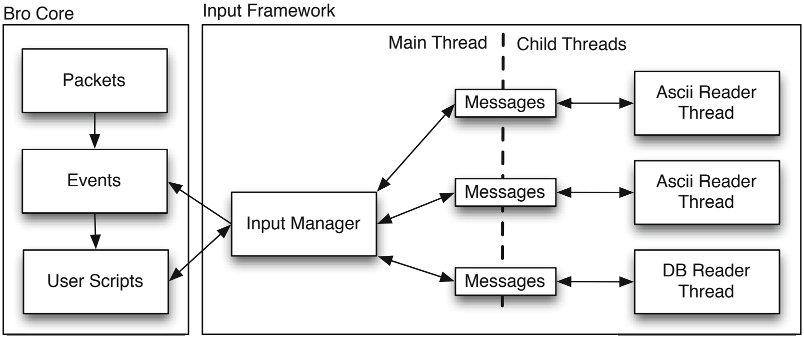


Fig. 3. Input framework implementation in Bro

When reading the blacklist file, the Input Framework will use the records' field names (`ip` and `reason/timestamp`) to locate the corresponding columns, and it will interpret their content according the fields' types (`addr` and `time` are Bro's built-in script types for IP addresses and time values, respectively.).

Next, we define the table that will receive the content of the file:

```
global blacklist: table[addr] of Value;
```

Note that the types for table index and values correspond to the `Index` and `Value` records, respectively.[4] Now that we have defined the types and the table, we use an Input Framework API function to read the blacklist in from a file: [5]

```
Input::add_table(source="blacklist.tsv", idx=Index,
                 val=Value, destination=blacklist);
```

Once read, further script code can test whether the blacklist contains a specific address:

```
if ( 192.150.186.11 in blacklist )
    alarm(...)
```

When executing the `add_table` function, the Input Framework's manager internally spawns a new reader thread and then immediately returns back to the caller. While the new thread is parsing the blacklist in the background, it continuously forwards entries to the manager, which in turn adds them to the `blacklist` table incrementally. Script processing continues in parallel, and other event handlers will hence "see" each new entry immediately. In addition, the Input Framework flags completion by triggering a callback event that users can implement for logic requiring that all data has made it into the table.

[4] For the table index we "roll out" the fields because Bro's tables do not support record types as keys. They do however allow for index *tuples* so if our blacklist were indexed by, say, two addresses, we would write `table[addr,addr] of Value`.

[5] In this and later examples we simply Bro's syntax slightly for better readability.

Updating. Many intelligence sources will see frequent updates, either batched in regular intervals or continuously in the form of a stream. Our implementation provides three mechanisms to accommodate updates.

The most direct mechanism is to call the API's `force_update` function, which will trigger a re-read of a stream's source data (for reader types that support it). The Input Framework will then add any new values to the corresponding table, remove ones that no longer exist, and update any that have changed to their new values. Using `force_update` is a good choice if one knows when to expect a change, such as when one has run an external command. Alternatively, one can also put a reader into *automatic update* mode (via a flag passed to the `add_table` call). In this mode, the reader thread will continually check the source file for modifications and trigger the update operation automatically. Automatic updating works well with files that an external script retrieves regularly from a remote location (for instance via cron).

Finally, for readers that receive continuous streams of intelligence, they may also individually add/delete/modify table entries as they get updates. Keeping with the file example, this mode corresponds to "tailing" a file on a Unix system. Usually, however, such streaming readers will have a persistent connection to an external data source. For example, new blacklist entries could be coming in via a network connection, or a database reader may subscribe to live query updates.

Filtering. Often it is beneficial to prefilter the information coming in from an intelligence source. In our blacklist example we might, for example, only want to consider entries added within the last few days. Our implementation offers corresponding hooks that can modify or remove entries before they land in the table. Such a hook comes in the form of a callback function that the Input Framework runs for any table update under consideration. The hook receives the line to be added to the table and the information whether the entry is new, changed or updated. The hook function returns a boolean that signals if the change is to be applied or rejected. If the hook rejects a change, the element is not added for new elements, not updated for changed elements and not deleted for removed elements. In our blacklist example, we could use that to only consider blacklist entries that are not older than five days.

One assigns such a filter to a stream by passing the function as an additional argument to the `add_table` call. We emphasize that filters can access all of Bro's already accumulated state, including other script-level tables and data structures. They can even *modify* other tables, which for example allows to split intelligence from a single source across a set of related tables.

Triggering Events. Some types of intelligence do not map directly to a table structure. For example, a source may be sending information that Bro must react upon immediately, rather than storing it for later inspection. To support such applications, our Input Framework implementation offers a second, simpler reading mode in which the manager triggers an event callback for every entry it receives. The callback's arguments are similar to that of the filter function described above but do not include table-specific elements like the index value.

4.3 Reader Types

Our current Input Framework implementation supports three types of intelligence readers. As we show in our examples above, it can read ASCII files in the form of typed tab-separated columns. We kept the format's specifics compatible to Bro's structured log files, which now enables users to read *log files back in*, providing a powerful mechanism to maintain rich state persistently across restarts.[6]

The second supported format is "raw" content, in which the reader simply passes on the content of a file as a raw blob. This mode can only trigger events, not table updates, as there is no further structure associated with the data. Optionally, the raw reader can also split a file at predefined separator characters. It is, for example, possible to get one event for each line in a file. As an extension, the raw reader can take its input not only from files but also from the output of custom shell commands. This feature enables in particular to query external web services using a utility like `curl`. The following code snippet demonstrates how Bro can retrieve a JSON file on the fly:

```
# Define the type that will store the data.
type JSON: record { data: string; };

# Define the handler that will process the JSON data.
event got_data(value: JSON) {
    ... Code to process data goes here ...
}

# Trigger the request.
Input::add_event(source="curl www.host.com/list.json |",
                 fields=JSON, event=got_data, reader=Input::RAW);
```

The `add_event` call creates a new reader thread that first executes the external command and returns the output asynchronously by generating a `got_data` event. The event handler can then further parse the data.[7] Note how the `source` argument ends with a pipe symbol to indicate that the value reflects a command to execute, not a file name.

Finally, as our third reader type, we develop a PostgresSQL interface that executes SQL queries and forwards the result back to Bro, mapping it either transparently into a table or into events, as described above. We discuss this reader in more detail in §5.4 where we show a concrete usage scenario. We also add a PostgresSQL log *writer* to Bro. In combination, reader and writer enable users to perform arbitrary bi-directional database transactions in real-time, all in parallel to Bro's normal packet processing and with full access to its global state.

[6] Bro's logging system indeed complements our work by providing a corresponding *output framework*.

[7] Alternatively, one could parse the JSON externally as part of the executed command and use the ASCII reader to receive it in a structured form.

Our Input Framework implementation provides a simple self-contained API for implementing new readers that makes it straight-forward to add further types of input. In particular, we are planing to add interfaces to other databases as well as to syslog clients.

5 Deployment Scenarios

To support our claim that the Input Framework introduces a fundamentally new IDS capability, we now examine its potential from a deployment perspective. In §5.1 we first examine the need for *trusting* external intelligence as an overarching operational concern that current IDS do not sufficiently address. In §5.2 we discuss a real-world application of the Input Framework that is already in operational deployment at the Lawrence Berkeley National Laboratory. Finally, §5.3 and §5.4 discuss two further usage scenarios that we prototype as case studies and expect to similarly move into operations. We note that many specifics of our Input Framework design and implementation evolved through close interaction with network operators at a number of sites, and the discussion in this section captures much of the feedback we received.

5.1 To Trust or Not to Trust?

From an operational perspective, *trust* is a crucial concern when integrating third-party intelligence into a site's security decisions. Consider a site with a policy to automatically block connectivity for malicious external IP addresses. With external IP blacklists coming in for example from a federation like REN-ISAC, the operators need to decide which of the addresses justify a block. On a technical level, simply blocking all of them is rarely feasible due to limits on the number of rules that firewalls can handle. But more importantly, there is rarely any local control on what exactly the intelligence feeds include and hence their information requires additional vetting and a process to develop confidence in the quality of the data. In particular, operational usage needs to account for policy differences between sites—the IP address of a P2P tracker, or an *undernet* IRC server, may be criticial to block for one site, yet tolerated at another. Also, any accidental inclusions risk severely impacting legitimate traffic (finding IPs from Akamai or Amazon AWS blacklisted is not unusual). Furthermore, some feeds (like REN-ISAC's) contain additional qualifying information such as severity ratings, confidence levels assigned to an entry, or number of distinct sources reporting it. Such ratings tend to be highly subjective and are thus often insufficient to trigger automated action on their own. They may however contribute to crossing a threshold when combined with further orthogonal evidence.

Operationally, a crucial shortcoming of many IDS implementations is their lack of support for the fine-granular decisions that such considerations require. Accordingly, we see operators falling back to externally vetting information via custom scripts before then converting them into static IDS rules. That, however, lacks any flexibility to go beyond simple black/white decisions. There is no way to convert a dynamic reputation scheme into a standard signature.

The Input Framework addresses such concerns by providing the means to incorporate intelligence into the IDS decision process itself, rather than leaving it to external pre- or postprocessing. Doing so not only fundamentally improves detection capabilities and response times, but also provides considerable workflow improvements by eliminating the external process that attempts to fit the intelligence into what the IDS configuration language supports.

5.2 Federated Blacklists

Our Input Framework implementation is in operational deployment at the Lawrence Berkeley National Laboratory (LBNL), where the cyber security team uses it to integrate both SES feeds and JC3 feeds into the Lab's Bro installation. In the following we report on their experience with the new capability after nearly 2 months of use. LBNL adopted use of the Input Framework due to its ability to continuously integrate crucial indicators into the monitoring infrastructure as quickly as they are published. Prior to the Input Framework, it was not operationally feasible to repeatedly restart their IDS potentially multiple times an hour as feeds were published in an adhoc manner. Additionally, incorporating policies by hand was an error prone process causing unintentional delays.

LBNL prefers using the SES and JC3 limited-circulation feeds over other public sources as they supply vetted data and are continuously maintained. As such, these feeds allow for tight integration with the IDS and enable to automate decisions as their semantics are well understood. The institutions behind the feeds also allow LBNL to go back upstream and inquire about potential false positives or borderline cases. The SES feed is updated automatically once per day and the JC3 feed is downloaded manually from a secure server when updates are released. The SES feed contains individual subfeeds for spam, scanners, phishing, suspicious nameservers, and suspicious networks. In general, these feeds contain different types and volumes of intelligence in the order of 300–3000 lines per subfeed. Typical entries for SES are an malicious host's IP address, event timestamp, domain, port, URL and file MD5 hash, as appropriate. Each item also comes with a separate severity rating as well as a confidence level. JC3 provides malicious domains and IP addresses, augmented with information about which sites reported the threat and also threat-level estimates.

LBNL uses the Input Framework's table interface (see §4.2) to directly import the feeds into a set of Bro tables. External scripts query the feeds from the providers and write them to disk. The Input Framework then picks up the changes transparently. LBNL also uses the filter mechanism to modify data during imports. For example, some SES subfeeds do not contain hostnames as a separate column but only come with complete URLs to malware. However, these rarely appear as a whole in network traffic and LBNL hence uses a custom filter function to extract the hostname and turn it into a table index on the fly. Furthermore, LBNL joins several feeds into a single table by configuring multiple sources that all write to the same destination.

During the two months of deployment, this setup has proven to improve LBNL's detection capabilities in a number of ways. As an example, HTTP scanners are notoriously difficult to detect reliably because it is difficult to distinguish a malicious scanner from a search engine's web crawler. However, having intelligence feed integration, LBNL can now tie feed data to Bro's TCP-level scan detector. When the latter finds a possible scanner, the IDS checks to see if the IP is blacklisted; if so, it blocks it automatically. Combining the two detectors in this way allows to quickly block HTTP scanners without subjecting *all* blacklisted IPs to that treatment (and thus preventing many unjustified blocks). A similar approach works well for encrypted SSH and HTTPS traffic, which an IDS cannot further inspect at the content-level. The Input Framework has already triggered investigations in several such cases.

More generally, LBNL finds that steering subsets of intelligence into corresponding protocol-specific analyses leads to more reliable alarms than the standard approach of matching broadly against the bulk of the traffic. For example, LBNL uses a hook into Bro's TCP analysis to trigger intelligence lookups for the addresses of every newly established connection. Likewise, all DNS requests and replies lead to checks for the corresponding domain names. While already valuable on their own, one can also correlate matches across protocols. For instance, a blacklisted path may first appear in an HTTP request followed by a DNS lookup for a malicious domain. Indeed, the majority of intelligence-triggered alerts currently corresponds to such DNS-after-HTTP matches.

5.3 Online Virus Checks

As a second deployment scenario, we prototype online virus checks using the popular malware checking service *VirusTotal* [29]. While network-level virus checking is not a novel concept, most solutions operate as proxies that actively intercept TCP sessions. A few commercial systems seem to support passive virus checks (e.g., by FireEye and Netwitness) but we are not aware of available open-source solutions doing live packet-level scans. The Input Framework makes it straight-forward to support such functionality within an existing IDS.

In fact, with the Input Framework in place the main challenge is not interfacing to a virus checker but extracting files from network traffic. Here, we leverage the file extraction features that Bro's HTTP engine provides, including its support for content downloaded in chunks or from multiple sessions simultaneously. Hence it is, for example, possible to recognize viruses in files where a user first aborted a download and later resumed it at the aborted position.

We implemented online virus checking as a plugin to Bro's *file analysis framework* that is currently in development and scheduled to be part of an upcoming release. The framework supports reassembly for files transferred non-linearly and provides a convenient hooking point for handling files across protocols in a standardized way. A plugin for VirusTotal provides two alternative operation modes: it can either *(i)* calculate an MD5 hash for a file on the fly and submit that to the VirusTotal API for checking against its database; or *(ii)* submit the file *in whole* to the service. In both cases, VirusTotal returns a JSON-string indicating the results that we then parse in a Bro script using the Input Framework's event interface. If there is a match, the script can take action such as notifying an operator or disconnecting the victim system. An alternative to using VirusTotal would be to instead call a local virus scan engine. Doing so can be beneficial if privacy concerns prevent online lookups.

The Bro script providing the VirusTotal functionality on top of the Input Framework is just about one hundred lines long, including empty lines and comments. To avoid an excessive number of lookups, it allows to optionally analyze only a subset of all files, such as selected file types, or content from specific IP addresses or CIDR ranges. During our testing, we indeed detected a malicious file transfer from an compromised host in a 600MB real-world trace.

5.4 Database Interface

Our initial implementation also provides a database reader that connects to PostgresSQL. The reader supports both importing data once from a DB table, and continuously as updates from live queries arrive. Compared to the text-based intelligence we have used in our examples so far, the database interface opens up further potential by providing real-time access to external intelligence that exceeds a volume that an IDS could handle itself internally.

To demonstrate this capability, we consider a setting where the IDS flags suspicious activity for its operators but also augments the alert with further context about the attack source. Specifically, we want to integrate *whois* information into the notification, such as the time when a domain was registered and its administrative contact information. The Bro-side for that comes in two parts: *(i)* we hook into Bro's processing to execute a database query via the Input Framework when an alarm triggers; and *(ii)* when the database's reply arrives, we augment the alarm accordingly and then pass it on for further processing. Somewhat simplified[8] the query looks like this:

```
add_event(source="select * from whois where domain='"+ domain +";",
   name=<uid>, event=got_reply, reader=Input::POSTGRES);
```

Here, the `uid` is an automatically generated unique identifier that later allows the `got_reply` handler to associate a reply with the corresponding query.

We test this approach using an internally maintained PostgresSQL database that contains complete whois information for several million domains, and we find it to work as expected. In practice, one could extend this scenario in a number of ways. For example, rather than just augmenting alerts, the IDS can use the database information to further assess the threat, such as by elevating an alarm's priority when it involves recently registered domains.

More generally, this scenario demonstrates how the Input Framework can make intelligence available to the IDS *on demand*, without needing to move all the information into the system itself. Database connections is the most powerful of all our examples, and we expect that operators will start relying on them extensively as they become familiar with the new capability.

[8] We configure the database connection separately. In practice, one must ensure to sanitize the `domain` to avoid SQL injection attacks.

6 Performance Evaluation

To understand the performance of our implementation we perform a set of measurements to determine *(i)* the intelligence volume it can handle under realistic workloads; and *(ii)* the latency with which real-time input becomes available to the IDS analysis.

6.1 Benchmark Reader

We create a dedicated *benchmark reader* for our measurements. Rather than connecting to an actual intelligence source, that reader generates artificial data with characteristics that we can freely configure. The reader first examines the data types requested via the Input Framework API (see §4), and then generates corresponding table updates and events. For example, when the API requests intelligence of string type, the benchmark reader returns a random byte sequence. It recursively fills in fields of record types and can hence generate arbitrarily complex data structures on the fly. One can configure the rate with which it sends updates and also an optional increase of that rate over time.

6.2 Realistic Workloads

It is challenging to benchmark IDS systems with realistic workloads in a way that is repeatable and has reproducible results. We cannot just run the system on live traffic because continuous variations in packet volume and mix would not allow for fair comparisons of different configurations executed sequentially. We however also cannot run offline from traces as Bro would process the packets as quickly possible (i.e., at 100% CPU usage), without the normal lags seen during real-time operation that the Input Framework uses to interleave intelligence updates with the packet processing. To overcome these problems, we leverage Bro's *pseudo-realtime* mode [22] which combines the best of both worlds. In that mode, Bro reads its input from a trace, yet it mimics real-time behavior by introducing artificial delays into the packet processing, corresponding to timestamp differences of consecutive packets. Doing so results in reproducible operation that is comparable to using the Input Framework on live traffic.

For our evaluations we capture a 5-minute packet trace at the uplink of UC Berkeley. The campus' upstream connectivity consists of two 10 GE links, with daytime average rates of 3-4Gb/s total. Such a volume is much more than a single Bro instance can handle, and we thus record only a subset corresponding to a more realistic setting. Specifically, we capture the traffic that a single Bro instance analyzes in the Berkeley campus' NIDS Cluster [27], which corresponds to $\frac{1}{28}$ of all flows. The resulting trace contains 100M unique IP addresses and 330K flows. 81% of the packets are TCP, and port 80 is the most common port (31,7%). The average data rate is 222 MBit/s at about 40K packets/sec.

For our measurements we use a current development version of Bro with a recommended, complex default configuration. When running without the Input Framework, a Bro process exhibits a CPU load of 50-80% while processing the trace on our evaluation system[9], which is about the level realistic for live operation without incurring packet drops.

6.3 Sustainable Load

We measure the load of the main Bro thread when the benchmark reader generates certain fixed numbers of events per second. Besides system characteristics such as CPU and memory resources, the sustainable data rate depends on the complexity of the involved data types (i.e., the record definitions), and on the reading mode in use (table updates or events).

For simple events, consisting just of a timestamp, we measure a limit of about 42,000 events/sec. Fig. 4 compares the CPU utilization for three different rates. For each rate, the plot shows the probability density of CPU load samples measured in 1s intervals over the course of processing the 5-min input trace. For comparison, we also show the load for a baseline run that does not activate the Input Framework. We see that at 10,000 events/sec, the CPU load increases just slightly (average 51% vs. 49%). At 36,000 events/sec it increases more noticably (average 58%), and at 50,000 events/sec individual CPU samples exceed 1.0s, i.e., more than the system can support.[10]

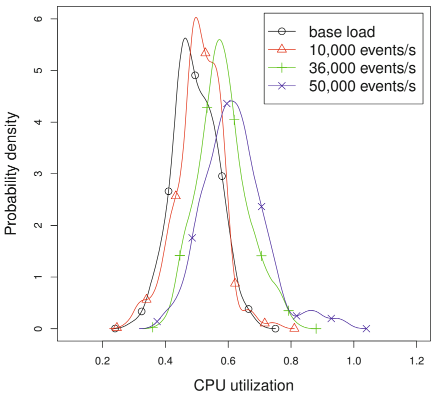


Fig. 4. Input Framework load increases

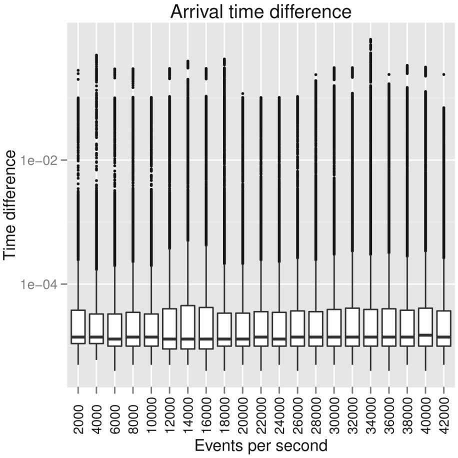


Fig. 5. Event latency evaluation

We repeat similar measurements with more complex events as well as with simple and complex tables. For the complex case, the record type contains 14 different data fields, of which 5 are list types (which is more than operations are likely to use). The sustainable loads for complex events are about 4,000 entries per second. For simple tables, the Input Framework can handle about

[9] The system has two quad-Core Xeon E4530 CPUs @2.66 GHz and 12 GB RAM.

[10] With Bro's pseudo-realtime mode, a CPU sample >1s means that the time required for processing 1s of network traffic exceeded 1s of real-time, which in live operation would have resulted in packet loss.

20,000 entries per second and for complex tables about 2,000 entries per second. The CPU loads at those rates are similar to those in Fig. 4 and we thus skip corresponding plots. We also examine system load with very low data rates (10s of updates/sec), which is more likely what one will see in typical deployments. For these, we do not see any measurable load increase.

Overall, we conclude that even with complex intelligence data, our implementation can sustain more than 3,000 insertions/sec while processing a typical packet load with a complex IDS analysis configuration. With less complex input, it achieves rates matching that of the packet input(!). The observed CPU increase is hardly surprising at such high rates. Operationally, however, the most relevant result is a different one: having headroom to accommodate high update frequencies is good, yet most deployments will never see such rates. For them, the we find that the Input Framework does not increase CPU usage.

6.4 Latency

From an operations perspective, the time it takes to make intelligence updates available to the IDS analysis is another important factor for operations. Consequently, we also measure the Input Framework's latency, i.e., the difference between the time when it receives an update from a source until that becomes available at the scripting layer. We configure the benchmark reader to generate events that include the current timestamp, and a receiving Bro script then calculate the difference. As in the load evaluation, we use Bro's pseudo-realtime mode running again on the same trace file. We performe a series of measurements, each time increasing the rate at which the benchmark reader sent events. We stop the series at the maximal attainable rate of 42,000 events/sec.

Fig. 5 visualizes the measured latencies for each rate in the form of a bar plot. (Note the logarithmic scale on the y-axis.) The whiskers end at 1.5*IQR+Q3, all other points are considered outliers and plotted as a dot. We see that the latencies remain very small, averaging around `1.4ms`. There are a few infrequent cases that have latencies in excess of 100 msec (less than 0.4%)—however, even in the worst case, the latency is under 900 ms. The minimum time difference is 4 μsec and hence in the order of measurement inaccuracies. Interestingly, the latencies do not change much at all as the rate increases, indicating that as long as the Input Framework can operate at a rate, it will forward updates rapidly. Overall, we conclude that the Input Framework does not add significant delays after receiving intelligence from a source.

7 Conclusion and Outlook

The global security community is collecting a treasure trove of third-party intelligence that can support operations staff in automating incident detection and investigation, including many forms of blacklists recording known bad actors and malicious content. Unfortunately, network intrusion detection systems still miss out on fully leveraging this potential for making more reliable decisions as they do not offer corresponding interfaces for flexibly integrating such knowledge.

In our work, we present a novel architecture that adds unconstrained intelligence access as a new capability to the IDS toolbox. We design an Input Framework that adapts to a variety of sources, provides a simple yet flexible user interface, and integrates smoothly with an IDS' main task of analyzing high-volume packet streams under soft real-time constraints. We implement an initial version of the Input Framework on top of the open-source Bro IDS. We also prototype a set of usage scenarios that exploit the power of the new capability, including integration with federated intelligence sharing initiatives, online virus checks for downloaded files, and real-time interaction with a PostgresSQL database for assessing the relevance of alarms on the fly. Furthermore, separate benchmark measurements confirm that our implementation is well-suited to handle frequent real-time intelligence updates while adding virtually no delay before making it available to the IDS analysis.

This Input Framework implementation is already in operational deployment at the Lawrence Berkeley National Laboratory where the Lab's security team finds it to significantly improve their detection capabilities. With the Input Framework's generic approach to integrating intelligence, we are looking forward to the operations community developing further powerful applications as they become familiar with the new capability.

Acknowledgments: We would like to thank the Lawrence Berkeley National Laboratory for their collaboration. This work was supported by the U.S. Army Research Laboratory and the U.S. Army Research Office under MURI grant No. W911NF-09-1-0553; a fellowship within the Postdoc-Programme of the German Academic Exchange Service (DAAD); by the Director, Office of Science, Office of Safety, Security, and Infrastructure, of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231; and by the US National Science Foundation under grant OCI-1032889. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors or originators and do not necessarily reflect the views of the DAAD, the ARL/ARO, the DOE, or the NSF, respectively.

References

1. Department of Energy Cyber Joint Cybersecurity Coordination Center, http://www.doecirc.energy.gov/
2. National Software Reference Library, http://www.nsrl.nist.gov/
3. Antonakakis, M., Perdisci, R., Dagon, D., Lee, W., Feamster, N.: Building a Dynamic Reputation System for DNS. In: USENIX Security (2010)
4. Blacklist.rules, ClamAV, and Data Mining, http://vrt-blog.snort.org/2011/02/blacklistrules-clamav-and-data-mining.html
5. Collective Intelligence Framework, http://code.google.com/p/collective-intelligence-framework/
6. Cyber Fed Model – Community-Wide Cyber Security Alert Distribution, http://web.anl.gov/it/cfm/
7. Cymru, T.: Malware Hash Registry, http://www.team-cymru.org/Services/MHR/

8. Debar, H., Wespi, A.: Aggregation and Correlation of Intrusion-Detection Alerts. In: Lee, W., Mé, L., Wespi, A. (eds.) RAID 2001. LNCS, vol. 2212, pp. 85–103. Springer, Heidelberg (2001)
9. Dreger, H., Feldmann, A., Paxson, V., Sommer, R.: Operational Experiences with High-Volume Network Intrusion Detection. In: ACM CCS (2004)
10. DShield.org Recommended Block List, http://feeds.dshield.org/block.txt
11. Google Safe Browsing API, http://code.google.com/apis/safebrowsing
12. Gu, G., Porras, P., Yegneswaran, V., Fong, M., Lee, W.: BotHunter: Detecting Malware Infection through IDS-driven Dialog Correlation. In: USENIX Security (2007)
13. Jung, J., Paxson, V., Berger, A.W., Balakrishnan, H.: Fast Portscan Detection Using Sequential Hypothesis Testing. In: IEEE Security and Privacy (2004)
14. Katti, S., Krishnamurthy, B., Katabi, D.: Collaborating against common enemies. In: IMC (2005)
15. Ollmann, G.: Blacklists & Dynamic Reputation. White paper (2011), http://www.damballa.com/downloads/r_pubs/WP_Blacklists_Dynamic_Reputation.pdf
16. Paxson, V.: Bro: A System for Detecting Network Intruders in Real-Time. Computer Networks 31(23–24), 2435–2463 (1999)
17. Roesch, M.: Snort: Lightweight Intrusion Detection for Networks. In: Systems Administration Conference (1999)
18. Security Event System, http://www.ren-isac.net/ses
19. Sharma, A., Kalbarczyk, Z., Barlow, J., Iyer, R.K.: Analysis of Security Data From a Large Computing Organization. In: IEEE DSN (2011)
20. Sinha, S., Bailey, M., Jahanian, F.: Improving SPAM Blacklisting through Dynamic Thresholding and Speculative Aggregation. In: NDSS (2010)
21. Snort 2.9.1 release announcement, http://blog.snort.org/2011/08/snort-291-has-been-released-including.html
22. Sommer, R., Paxson, V.: Exploiting Independent State For Network Intrusion Detection. In: ACSAC (2005)
23. The Spamhaus Block List, http://www.spamhaus.org/sbl
24. Open Information Security Foundation: Suricata Download, http://www.openinfosecfoundation.org/index.php/downloads
25. Symantec - Configuring blacklisting for base event types with IDS/IPS on Symantec Gateway Security 5400 Series 2.x, http://www.symantec.com/business/support/index?page=content&id=TECH81936
26. Valdes, A., Skinner, K.: Probabilistic Alert Correlation. In: Lee, W., Mé, L., Wespi, A. (eds.) RAID 2001. LNCS, vol. 2212, pp. 54–68. Springer, Heidelberg (2001)
27. Vallentin, M., Sommer, R., Lee, J., Leres, C., Paxson, V., Tierney, B.: The NIDS Cluster: Scalable, Stateful Network Intrusion Detection on Commodity Hardware. In: Kruegel, C., Lippmann, R., Clark, A. (eds.) RAID 2007. LNCS, vol. 4637, pp. 107–126. Springer, Heidelberg (2007)
28. Verizon: Data Breach Investigations Report. Tech. rep. (2012), http://www.wired.com/images_blogs/threatlevel/2012/03/Verizon-Data-Breach-Report-2012.pdf
29. VirusTotal Public API, https://www.virustotal.com/documentation/public-api/

GPP-Grep: High-Speed Regular Expression Processing Engine on General Purpose Processors

Victor C. Valgenti[1], Jatin Chhugani[2], Yan Sun[1], Nadathur Satish[2],
Min Sik Kim[1], Changkyu Kim[2], and Pradeep Dubey[2]

[1] School of EECS
Washington State University
{vvalgent,ysun,msk}@eecs.wsu.edu
[2] Parallel Computing Lab
Intel Corporation
jatin.chhugani@intel.com

Abstract. Deep Packet Inspection (DPI) serves as a major tool for Network Intrusion Detection Systems (NIDS) for matching datagram payloads to a set of known patterns that indicate suspicious or malicious behavior. Regular expressions offer rich context for describing these patterns. Unfortunately, large rule sets containing thousands of patterns coupled with high link-speeds leave most regular expression matching methods incapable of matching at real-time without specialized hardware.

We present GPP-grep, an NFA-based regular expression processing engine designed for maximum performance on General Purpose Processors. The primary contribution of GPP-grep is the utilization of the data-level parallelism available in modern CPUs to reduce the overhead incurred when tracking multiple states in NFA. In essence, we build and store the NFA in an architecture-friendly manner that exploits locality and then traverse the NFA maximizing the parallelism available and minimizing cache-misses and long-latency memory lookups. GPP-grep demonstrates 24–57× improvement in throughput over standard finite automata techniques on a set of up to 1200 regular-expressions culled from the NIDS Snort, and is within 1.3× of FPGA hardware-based techniques. GPP-grep achieves 2 Gbps throughput on a dual-socket commodity CPU system allowing for line-speed evaluation on *commodity* hardware.

1 Introduction

Pattern matching is a primary component of Network Intrusion Detection Systems (NIDS) that employ Deep Packet Inspection (DPI). DPI necessitates the comparison of every datagram payload against a set of known patterns. Fixed string patterns offer limited ability for expressing the complexities of modern network attacks, especially in the face of evasive techniques employed by attackers [15]. Regular expressions provide much richer context with which to design signatures enabling not only greater precision, but also greater resilience to evasive techniques. However, efficiently matching regular expressions can prove intractable, especially when faced with large sets of regular expressions combined with a high volume of traffic, and can result in near-incapacitation of NIDS when deployed in high-speed environments [12].

D. Balzarotti, S. Stolfo, and M. Cova (Eds.): RAID 2012, LNCS 7462, pp. 334–353, 2012.

To promote efficient regular expression matching the set of regular expressions are reduced to Non-deterministic Finite Automata (NFA) or Deterministic Finite Automata (DFA). While NFA provide for very compact memory utilization, they suffer in throughput as multiple active states must be maintained as all possible paths through the NFA are traversed. DFA exhibit faster throughput, as only one active state is ever needed, but the number of states in the automata can grow exponentially and require excessive amounts of system memory. Current automata solutions such as eXtended Finite Automata (XFA) [24], Hybrid Automata [3], Delayed input DFA (D^2FA) [16], and Ordered Binary Decision Diagrams [30] improve the memory and time efficiency of regular expression matching. Similarly, hardware techniques employing Field Programmable Gate Arrays (FPGA) [19], Graphics Processor Units (GPU) [26,5], or Cell processors [21] utilize specialized hardware to improve matching. However, specialized hardware can prove expensive and unmanageable while purely automata-based techniques do not necessarily exploit the parallelism already embedded in current multi-core and many-core processors.

In this paper, we present **GPP-grep**, a high-speed, NFA-based, regular expression engine for General Purpose Processors (GPP). GPP-grep exploits thread-level and data-level parallelism to achieve gigabit processing rates. Further, GPP-grep utilizes specialized NFA construction and storage to both reduce the total number of active states and exploit locality during NFA traversals. The primary contribution of GPP-grep stems from its ability to merge efficient NFA construction, storage, and traversal techniques with a more complete use of GPP processing power to arrive at a performance of up to 57× faster than traditional NFA engines and within 1.3× of a hard-wired FPGA-based NFA engine (on a 12-core CPU system). With 1,200 real NIDS regular expressions GPP-grep achieves real-time processing rates of 2 Gbps on *commodity* hardware.

2 Related Work

Regular expressions provide signature creators a wide context within which to describe dynamic patterns such as those occurring in polymorphic worms or customized attacks [15]. Unfortunately, matching thousands of regular expressions against the payloads of many thousands of packets-per-second can result in total failure of a NIDS in multi-gigabit environments [12]. DFA provide a fast software implementation for matching but suffer from state explosion when ambiguous characters, such as *wild-card* characters, are used in expressions. Such *wild-card* characters result in an exponential increase in the number of states required for the DFA which, in turn, requires much more memory to store. Since NIDS may employ thousands of regular expressions, nearly 1,600 distinct regular expressions for the default-enabled rules of the Sourcefire Vulnerability Research Team (VRT) Snort rule-set for August 11, 2011 [27], and since these regular expression are complex expressions with an average of six *wild-card* characters, DFA can grow too large to reside in main memory. Conversely, NFA have very compact representations in terms of memory, but require wider bandwidth to traverse as all possible paths through the NFA must be explored simultaneously. Traditionally, NFA are not considered a viable solution for NIDS, nor for regular expression matching, as the single state transitions of DFA appear to offer the best chance at throughput.

However, the growth in number of regular expressions employed by NIDS, as well as the propensity for these regular expressions to use *wild-card* characters, has begun to make DFA matching infeasible as the resultant DFA are simply too large. Compression [1], rule rewriting [31], and add-on data [20] can create smaller DFA, but can also result in added overhead and processing of the DFA.

Much research has sought to improve the efficiency of automata. Smith et al. present XFA [24] which augment Finite State Automata (FSA) with added variables to track state during processing. The added state information serves to provide the FSA with enough hints on processing data that it can both perform faster and in less space, on average, than vanilla FSA. Becchi et al. present Hybrid Finite Automata [3,4] which employ a small, head, DFA for the most common states (closer to the root). However, for matching that extends deep into the automata, tail NFA are employed to succinctly represent these deeper and less traveled regions. This hybrid finite automata demonstrated smaller size than a comparable DFA, but with a much better cache hit ratio and faster processing. Kumar et al. [16,17] introduced Delayed input Deterministic Finite Automata (D^{2}FA) and Content addressed Delayed input Deterministic Finite Automata (CD^{2}FA). D^{2}FA essentially combines identical transitions from multiple states to reduce the total number of states and, ultimately, the size of the Finite Automata. CD^{2}FA use content labels rather than state identifiers that allow skipping default transitions in certain cases. The end results were much smaller than normal DFA that achieved roughly the same memory bandwidth. Yang et al. [30] adopted an approach more closely aligned with GPP-grep in that they sought to improve the throughput of NFA. They employed Ordered Binary Decision Diagrams to maintain the space-efficiency of typical NFA representations while greatly improving the NFA traversal throughput. Also similar to GPP-grep, Shenoy et al [23] attempt to make the storage of state in Finite State Machines more efficient.

Another common tactic is to take advantage of *specialized* high-compute platforms to speed up regular expression matching. Mitra et al. [19] compile PCRE op-codes to Very High speed integrated circuit Description Language (VHDL) so that the matching can be executed on a Field Programmable Gate Array (FPGA). This allows the expression matching to occur in parallel across multiple NFA. The end result is a significant increase in throughput. Other approaches include Smith et al. who map DFA/XFA to Graphics Processing Units (GPU) [26], iNFAnt which also maps NFA to GPUs and provides efficient traversal algorithms [5], the use of the Cell processor as illustrated by Scarpazza et al. [21], and Meiners et al. [18] who employ TCAM to compactly encode multiple DFA and achieve high throughput.

GPP-grep differs from these approaches by creating and implementing an efficient automaton in a manner that fully utilizes the parallelism available in modern multi-core and many-core processors. This presents several advantages for GPP-grep. First, it mitigates the need for *specialized* high-compute platforms by maximizing the full potential of general purpose processors. Thus, GPP-grep can achieve performance benchmarks comparable to hardware implementations on a low-cost, ubiquitous piece of hardware. Secondly, the architecture-friendly layout for automata used in GPP-grep can extend to other approaches, such as those mentioned earlier, to arrive at improved performance. Finally, GPP-grep itself could benefit from the other approaches mentioned earlier thus potentially allowing for complimentary improvement through the combination of different approaches. Ultimately, GPP-grep attempts to merge both finite automata considerations with general purpose processor considerations to arrive at a regular expression matching engine that demonstrates improvements far beyond what either tactic might manage alone.

3 Modern Architectures

Exposing instruction-level parallelism is critical to utilize the multiple functional units within each processor core. This requires unrolling/software pipelining as well as proper instruction scheduling to expose independent instructions for simultaneous execution. In order to utilize all integrated cores, the application must expose thread-level parallelism. For regular expression matching, parallelization can be done across multiple input packets, with each processor executing matches on different packets. Modern processors also have wide vector Single Instruction, Multiple Data (SIMD) units that can execute instructions on many data items in parallel. For regular expression matching using NFA we utilize SIMD parallelism to perform the next state transitions of multiple active states in parallel. These techniques are further described in Section 5.3.

Instructions with high latency can lead to low utilization of functional units since they block the execution of dependent instructions. This is typically due to long latency memory accesses which can be reduced if the working set of the application (the data size for which the number of cache hits is 90%) fits in the last-level cache of the processor architecture. For regular expression matching the core traversal algorithm involves accesses to distinct memory locations which will ordinarily not be present in cache. In Section 5.2, we rearrange the nodes of the automaton to minimize the impact of cache misses. In addition to cache misses, misses to an auxiliary structure called the Translation Lookaside Buffer (TLB), which is used to perform the conversion from virtual to physical memory addresses prior to each memory access, can also result in significant performance degradation. This is also reduced using our hierarchical blocking scheme described in Section 5.2.

Finally, misses in cache also result in increased use of memory bandwidth. While the compute resources in modern architectures have increased rapidly, memory bandwidth is on a slower curve. To minimize bandwidth utilization, we attempt to ensure that every piece of data brought into cache in the form of cache lines is fully utilized before being evicted. This is ensured by the cache line blocking technique that we describe in Section 5.2.

4 Efficient NFA Construction

The first step in matching regular expressions is to build a finite automaton from a set of given regular expressions. Once construction of the finite automaton is complete, strings of symbols are applied to the automaton as an input until a final (accept) state is reached (a match) or the input is exhausted (no match).

An NFA may require as many state transitions per symbol as the total number of states in the NFA plus the possible addition of an epsilon transition. An epsilon transition allows the advancement to a new state without consuming an input symbol. This creates a time complexity for each symbol of $O(m)$ where m defines the total number of states in the NFA. For single-byte symbols, this translates to as many as 256 transitions per state (excluding epsilon transitions). The benefit, however, is that the NFA is very efficient in terms of space as only a single state exists for each possible state from the generating alphabet. On the other hand, a DFA requires only one state transition per symbol, but needs a much larger amount of memory to adequately map all possible traversals through the finite automaton. This burden on memory only grows as the number of complex regular expression components, such as Kleene stars, that directly lead to state explosion continue to increase both in frequency (number of rules containing complex components) and density (number of components per expression) with an average of six such components per regular expression in the Snort VRT [27] rule set.

4.1 Challenges in NFA Construction

In order to design efficient NFA previous approaches emphasized the generic reduction of states and transitions. However, reducing the number of "active" states is more important than simply reducing total states since the CPU has to deal with all the active states for each symbol in the input string. When converting regular expressions into an NFA, Thompson's algorithm [28] is often adopted. In Thompson's algorithm, ε is used to represent an empty string and is termed the epsilon transition. Unfortunately, NFA generated by Thompson's algorithm have two major drawbacks: first, there are many redundant states and transitions in the NFA, and second, there are many redundant epsilon transitions in the NFA that can significantly increase the number of active states for a traversal. In order to reduce active states in NFA we can merge some states. Minimizing NFA is a hard problem [13] thus it is necessary to develop heuristic methods to control minimization in order to make the problem tractable.

4.2 Reducing the Number of Active States

An NFA is defined by a quintuple $A = (S, \Sigma, \delta, s_0, F)$, where S is a finite set of states, Σ is the alphabet, δ is the transition function, s_0 is the initial state and F is the set of final states.

Given an NFA, we reduce the number of active states by combining "mergeable" states into one. Formally, given an NFA $A = (S, \Sigma, \delta, s_0, F)$, let p and q be two different states in S. Then, by combining p and q, we obtain another NFA, $A' = (S', \Sigma, \delta', s_0, F')$, that satisfies the following conditions:

$$S' = S - \{q\}, \tag{1}$$

$$\delta'(s,\omega) = \begin{cases} \delta(p,\omega) \cup \delta(q,\omega) & \texttt{if } s = p \\ \delta(s,\omega) & \texttt{otherwise}, \end{cases} \tag{2}$$

and

$$F' = \begin{cases} (F - \{q\}) \cup \{p\} & \texttt{if } q \in F \\ F & \texttt{otherwise}. \end{cases} \tag{3}$$

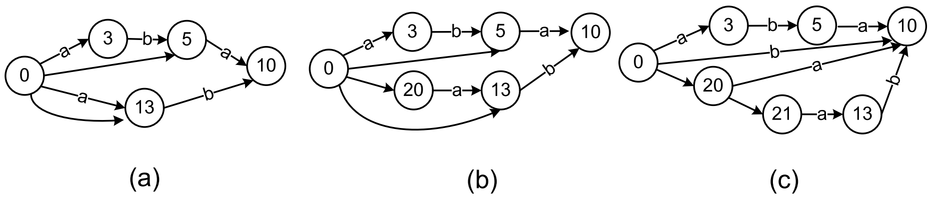


Fig. 1. Various NFA types for the RE ((ab|ε)a|(a|ε)b); (a) MM (b) SM (c) SS

Let $L(A)$ be the language accepted by A. Then states p and q are *mergeable* if and only if $L(A) = L(A')$. In fact, there exists a weaker condition for two states to be mergeable [7].

Definition 1. *The* left language *of a state s in an automaton A is* $\overleftarrow{L}^A(s) = \{\omega \in \Sigma^* \mid s \in \delta(s_0, \omega)\}$.

Definition 2. *The* right language *of a state s in an automaton A is* $\overrightarrow{L}^A(s) = \{\omega \in \Sigma^* \mid \delta(s, \omega) \cap F \neq \varnothing\}$.

Note the generalized usage of δ. It represents a set of NFA states reachable by the string ω and any number of ε transitions preceding ω, following ω, and between any successive symbols in ω. Additionally, [7] prove the following proposition.

Proposition 1. *Two states, p and q, in an automaton A are mergeable if and only if* $\overleftarrow{L}^A(p) = \overleftarrow{L}^A(q)$ *or* $\overrightarrow{L}^A(p) = \overrightarrow{L}^A(q)$.

Unfortunately, testing the condition in Proposition 1 is NP-hard, requiring global knowledge [6]. However, because we build an NFA from a regular expression, which is a linear representation of the NFA, we do not need to deal with an arbitrary NFA. Based on this observation, we propose a sufficient mergeability condition, and design a novel heuristic algorithm to identify most mergeable states during a single-pass conversion from a regular expression to an NFA.

Proposition 2. *Two states, p and q, in an automaton* $A = (S, \Sigma, \delta, s_0, F)$ *are mergeable if* $\delta(p, \varepsilon) = \{q\}$ *and either (i)* $\bigcup_{c \in \Sigma} \delta(p, c) = \varnothing$ *or (ii)* $\{(s, c) \mid c \in \Sigma \cup \{\varepsilon\} \ \wedge q \in \delta(s, c)\} = \{(p, \varepsilon)\}$.

Proof. Suppose p and q satisfy the condition in the proposition. We prove (i) and (ii) separately.

(i) Because the epsilon transition is the only outgoing transition of p, we have $\delta(p, \omega) = \bigcup_{t \in \delta(p, \varepsilon)} \delta(t, \omega)$. Since $\delta(p, \varepsilon) = \{q\}$, the equation becomes $\delta(p, \omega) = \delta(q, \omega)$. Therefore, $\overrightarrow{L}^A(p) = \overrightarrow{L}^A(q)$.

(ii) Because the epsilon transition from p to q is the only incoming transition of q, for any ω such that $q \in \delta(s_0, \omega)$, we get $p \in \delta(s_0, \omega)$ by removing the last epsilon transition. Similarly, for any ω such that $p \in \delta(s_0, \omega)$, we get $q \in \delta(s_0, \omega)$ by adding another epsilon transition at the end. Therefore, $\overleftarrow{L}^A(p) = \overleftarrow{L}^A(q)$.

Thus, by Proposition 1, p and q are mergeable. □

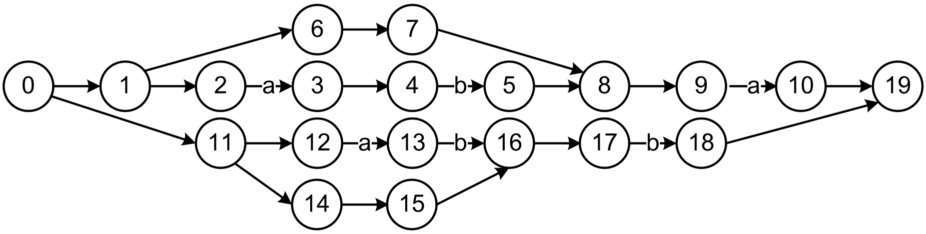


Fig. 2. NFA produced by the Thompson Algorithm for the RE ((ab|ε)a|(a|ε)b)

Note that to test conditions in Proposition 2, we only need the local knowledge of outgoing transitions for p and incoming transitions for q. To demonstrate the effectiveness of our state merging, we illustrate Thompson's NFA for $(ab|)a|(a|)b$ in Figure 2, and the result of state merging in Figure 1(a), which has significantly fewer active states than Thompson's NFA.

4.3 Building an Efficient NFA

After merging states, we will get an NFA with each state having multiple transitions, even for the same symbol, and multiple epsilon transitions (a multi-multi NFA, or MM). Such an NFA requires a dynamic data structure, such as a linked list, to maintain all outgoing transitions for each input symbol. A simpler way is to force each state to have at most one transition per symbol and at most one epsilon transition (a single-single NFA, or SS), and implement it as an array. Essentially, we wish to reduce an MM to an SS. Then the whole transition table of a state can be stored as a single entity with 257 elements (256 input symbols and one epsilon transition). To achieve at most one transition per symbol, we have to introduce new epsilon transitions to distribute multiple transitions for the same symbol over multiple states. In order to have room to add such new epsilon transitions, we first remove epsilon transitions from MM using Algorithm 1. For each epsilon transition from p to q followed by another transition from q to r, Algorithm 1 adds a shortcut from p to r, and eventually removes the epsilon transition.

After removing epsilon transitions, we serialize all outgoing transitions for the same symbol using new epsilon transitions. The pseudocode is shown in Algorithm 2. Note that every state on the chain of epsilon transitions created by this algorithm has exactly one incoming epsilon transition and one outgoing transition. In this way, we can guarantee that each state has at most one transition per input symbol and at most one epsilon transition. Given an MM NFA as shown in Figure 1(a), its SS counterpart is shown in Figure 1(c). We can use a similar algorithm to build an SM (a single-multi NFA), as shown in Figure 1(b). The only difference is to connect a new state n in Algorithm 2 to s directly using an epsilon transition instead of inserting it into the epsilon chain.

Overall, the algorithm to optimize an NFA created from a given set of regular expressions requires three steps: first, to create a compact NFA by merging states; second, to remove all epsilons from the NFA; and third, to force a single-transition per input symbol by creating an epsilon chain. This simplifies the memory representation of an NFA and its traversal algorithm, as we explain in Section 5.

Algorithm 1. Remove-Epsilons(State *s*)

```
if $s$ has been marked then
    return
end if
mark $s$ visited
for all $e$ such that $e$ is an epsilon transition out of $s$ do
    $d \leftarrow$ destination of $e$
    if $d$ has not yet been marked AND $d \neq s$ then
        add transitions in $d$ AND NOT in $s$ to the transitions of $s$
        if $d$ is accepting then
            set $s$ to accept
        end if
    end if
    remove $e$
end for
for all $t$ such that $t$ is a transition out of $s$ do
    $d \leftarrow$ destination of $t$
    Remove-Epsilons($d$)
end for
```

5 Efficient Layout and Traversal

In this section, we describe our architecture-friendly NFA layout and the efficient traversal algorithm. These optimizations are aimed at reducing the traversal cost and the resultant working set (the most frequently accessed data), thereby resulting in increased NFA processing throughput.

5.1 Motivation

NFA are typically stored using an adjacency list representation, with each state storing its outgoing *symbol* and ε-transitions. The NFA traversal algorithm maintains a list of *active states*, labeled `AS`, initialized to the `Start` state(s). For each input symbol α, the traversal consists of the following two steps:

Step 1: Computing the list of neighboring states (`NS`) for the symbol α for all states in `AS`.

Step 2: Computing the ε-closure[1] of all the states in `NS`, and assigning the resultant states to `AS`.

In the case where `AS` consists of the `End` state(s), a *match* is found, and the NFA execution may continue until `AS` contains no `End` state(s) (to find the longest sub-string match). On the other hand, in the case where all the input symbols are processed without ever reaching `End` state(s), the input stream does not match any regular expression in the set of regular expressions.

[1] By definition, ε-closure of any state *A* consists of the state *A* and all the states reachable using ε-transitions from ε-neighbors of *A*.

Algorithm 2. Force-Single-Transition-Per-Symbol (State s)

```
if $s$ has been marked then
    return
end if
mark $s$ visited
for all $c$ such that $c \in \Sigma$ do
    if there are multiple transitions for $c$ then
        for all $t$ such that $t$ is a transition $s \xrightarrow{c} d$ except the first do
            remove $t$
            insert a new state $n$ at the head of epsilon chain
            add a transition $n \xrightarrow{c} d$
        end for
    end if
end for
for all $t$ such that $t$ is a transition out of $s$ do
    $d \leftarrow$ destination of transition
    Force-Single-Transition-Per-Symbol($d$)
end for
```

The runtime of the traversal algorithm is dependent on the following 3 factors: (1) For each *active state* in AS, the time taken to lookup the neighboring states in Step 1. (2) The time taken to lookup the ε-neighbors (and the subsequent ε-closure) for each state in NS in Step 2. (3) Maintaining a unique list of states in AS and NS to avoid redundant computation.

Typical implementations use either linear or log-time complexity algorithms [2] to lookup neighbors and maintain unique lists. One primary reason has been the focus on reducing the total memory footprint and using compact representation at the expense of increased traversal cost. In contrast, our layout aims to exploit the existence of hardware caches and reduce the actual working set to make it fit in the cache, thereby minimizing the access cost and thus reducing the instructions required for the traversal. This layout makes our traversal algorithm compute-bound (rather than latency or bandwidth-bound), and also provides an opportunity to exploit the Single Instruction, Multiple Data (SIMD) execution units to further improve run-times.

5.2 Architecture-Friendly Layout

For the remainder of this section, let $\mathcal{S}$ denote the set of symbols, and $|\mathcal{S}|$ the cardinality of $\mathcal{S}$ (e.g. 256 for 8-bit input symbols—the most common case). Furthermore, let $\mathcal{M}$ denote the set of states (also referred to as nodes). We define the depth of any node as the shortest distance (in terms of number of edges) traversed from the Start state to that node. By definition, the Start state has a depth of *zero*.

On modern architectures, the data transfers (from main memory to caches, and within caches) are performed at the granularity of cache lines (typically 64 bytes or longer). In order to reduce memory accesses during traversal, it is important to reduce the number of accessed cache-lines, which implies storing temporally coherent data in proximity. Note that the NFA traversal involves neighborhood queries for all the active states for a

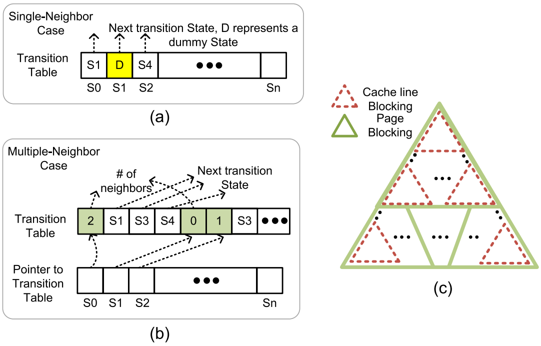


Fig. 3. (a) Layout for single-neighbor case. (b) Layout for multiple-neighbor case. (c) 2-Level hierarchical blocking of NFA nodes.

specific input symbol under consideration. Hence we need to store the transitions from various states for any given symbol close to each other. We therefore cluster all the transitions into ($|\mathcal{S}|$+1) groups ($|\mathcal{S}|$ for symbols and one for ε transitions). We now describe our layout for these transitions for any particular symbol. For ease of explanation, the remainder of the section uses the term symbol to include both input symbols and the ε transition.

We adopt different layout schemes dependent on whether we wish to construct a Single-neighbor (S) NFA where all the states have ≤ 1 neighbor or a Multiple-neighbor (M) NFA where states may have many neighbors. These NFA represent different design choices and are constructed as explained in Section 4.3. Consider the S case, and the symbol α. It is indeed possible for some states to have no transition for α. However, in the Single-neighbor NFA we must provide exactly one neighbor for every state in our layout in order to provide for a predictable data structure as illustrated in Figure 3(a). Thus, our layout employs a Dummy state for all non-existent transitions. While this increases the total number of states for the NFA by one, it offers an efficient method for determining no transitions.

Now consider the Multiple-neighbor (M) case. Different states may have varying number of neighbors, and hence we need to store a *count* for the number of neighbors. Furthermore, in order to reduce the number of accessed cache-lines, we need to store the *count* close to the state IDs. We therefore store the *count* followed subsequently by the neighboring IDs. This representation requires storing a pointer to the address of the memory location storing *count* (Figure 3(b)).

Our NFA layout described so far stores the states in increasing order, starting from state ID 0. However, the traversal pattern follows a specific order—for any given state, it accesses its neighboring (outgoing) state IDs. To improve cache locality, we need to store all the neighbors of a node close to each other. Doing so for all depths results in a breadth-first storage. However, this increases the storage distance between any node and its neighbors at larger depths. For input streams matching the given NFA, the depth of active states increases as we traverse through the inputs, thereby resulting in memory accesses that are separated by increasing distances. As described in Section 3, memory is laid out as pages and, for memory accesses offset by more than the page size, each access would result in a TLB miss which would increase memory access latency. In the worst case, we may incur a TLB miss for each symbol of the input stream. We propose a hierarchical blocking scheme that reorders the state IDs and reduces both the number of cache and TLB misses simultaneously.

Hierarchical 2-Level Blocking. The aim of hierarchical blocking is to *partition* the nodes into groups that fit entirely in a memory page (typical size of 4KB). Furthermore, we need to rearrange nodes within *every page* so that a node and its neighbors are stored close to each other—thereby fully exploiting a cache-line. We refer to this as our hierarchical blocking scheme. Assuming a 32-bit representation for each state, we can fit 16 states in a cache-line and 1K nodes in a 4KB memory page. We perform a global reordering of the nodes, and apply the same permutation to the states for all the symbols.

We start with the root node (`Start` state), and perform a depth-first traversal of the graph and assign depth values to all the nodes. Furthermore, we maintain a list of nodes that have not been assigned to any cluster—initialized to all the states in the NFA. We begin clustering by including the `Start` state and all its neighbors (irrespective of the symbol). We make progress by picking one of the unassigned nodes at the lowest depth and including it and its unassigned neighbors in the cluster. We continue the process until the threshold for the number of nodes in a page is reached. We then start clustering for the next page by either continuing with the neighbors of the node just being considered or starting with a new unassigned node. The process is terminated when all the NFA states have been assigned to any of the clusters.

We now describe our scheme for performing cache-line blocking within each cluster. We maintain a list of nodes that have not been assigned an index within the cluster. We start with the node at the lowest depth, and consider its neighboring states in the cluster. In the case where the number of unassigned neighbors is greater than 16 (the maximum number of neighbors within a cache-line), we assign the neighbors contiguously. We continue with the remaining neighbors, and assign them in a similar fashion. If the number of unassigned neighbors is less than 16, we fill up the cache-line partially and then continue with the process by selecting the unassigned node at the lowest depth. The process is carried out until all the nodes have been assigned an index (Figure 3(c)).

For any distribution of the symbols in the input stream, our hierarchical blocking scheme aims to reduce the average number of cache- and TLB-misses. In practice, the NFA traversal spends most of its time in the first few levels of the NFA, which are clustered together by our algorithm. Hence we have very few cache- and TLB-misses as shown in Section 6. Using our blocking scheme, we obtain a large performance improvement (of around 2.7×) for large NFA. Note that we reorder the nodes for each of the MM, SM and SS types of NFA as a pre-process step.

5.3 Traversal Algorithm

We first explain our technique to maintain unique lists in $O(1)$ time for our NFA layout, followed by the complete traversal algorithm for different NFA types. We maintain a time stamp (referred to as τ, and initialized to *zero*) for the simulation that gets updated for each step of the NFA traversal. Furthermore, we maintain an array (referred to as `TimeStamp`), that stores one time stamp value per state—representing the time stamp for the most recent access. This requires an additional two accesses per state (to check and potentially update its time stamp), but helps maintain unique states without the significant overhead due to sorting or linear searches. For each input symbol, τ gets incremented by one for the symbol-transitions and again incremented for the ε transitions. Depending on the granularity of the time stamp, we need to reset the `TimeStamp` array for all the states once the timer τ truncates to zero. Assuming a 32-bit timer, this reset process needs to be performed after traversing through 2 MB of input packets, rendering the amortized cost to be negligible.

We first describe the traversal algorithm for the Single-neighbor (`S`) case. Let α be the input symbol, and τ represent the current time step. Before starting the execution, `TimeStamp[Dummy]` $\leftarrow \tau$. Consider the Step 1 of the traversal (Section 5.1). The traversal consists of:

(a) Loading the next active state (S_i) in `AS`, followed by
(b) Looking up its α neighbor (say $S^{\alpha}{}_i$),
(c) Looking up the corresponding time stamp value (τ') of the neighbor (`TimeStamp`[$S^{\alpha}{}_i$]).
(d) Comparing τ' and τ, and
(e) In case $\tau' \neq \tau$, `TimeStamp`[$S^{\alpha}{}_i$] $\leftarrow \tau$ and append $S^{\alpha}{}_i$ to `NS` (since $\tau' = \tau$ implies that $S^{\alpha}{}_i \in$ `NS`).

The same process is carried out for Step 2 (for `S` case).

In the case of Multiple-neighbors (`M`), we replace the above steps by the following computations. We first (i) compute the address of the memory location containing the counter of the number of neighbors, and (ii) lookup the counter value. The rest of the process involves iterating over all the neighbors, and performing steps (a)–(e).

The `M` case clearly involves the added *overhead* of address computation and accessing the counter value prior to starting the efficient process of neighbor lookup and appending entries the list of unique active states. Further, it carries the additional *overhead* of loop computation and checking for termination. For a small number of neighbors this overhead contributes substantially to execution time, but gets amortized for large numbers of neighbors.

Improving ILP (Instruction-Level Parallelism). During the NFA traversal, a node may access cache-lines that are not resident in the cache. This scenario arises as we traverse deeper into the NFA graph. Such accesses may incur long latencies and stall the execution pipeline. In order to reduce the impact of such latencies on the execution time, we issue *software prefetches* in advance, before actually accessing such cache-lines. This reduces (and in most cases eliminates) the memory latency stalls. We modify our traversal algorithm as follows. During the execution of Step 1, we look at the subsequent input symbol (say β). As we identify and add states to the *NS*, we also issue software prefetches for all the memory locations storing the neighbors of these states (for symbol β and ε neighbors). This process is carried out during the execution of Step 2. Note that we can issue prefetches for all memory accesses, except for the ones that arise from the ε-transitions of the ε-neighbors of states in *NS*.

In addition to software prefetches, we also perform *loop unrolling* while iterating over the active states. Since steps (a), (b), and (c) above are completely independent for the various active states, these instructions increase the amount of parallel instructions available for the processor scheduler to improve the IPC (Instructions Per Cycle), and hence reduce the effective amount of execution cycles.

Exploiting Thread- and Data-Level Parallelism. We exploit the available multiple cores on current CPUs by dividing the input packets among the cores. This requires keeping a separate copy of the `TimeStamp`, `AS` and `NS` arrays. Since the cores perform *independent* traversals, we obtain near-linear scaling with number of cores.

In order to further reduce the executed instructions, we take advantage of the SIMD execution units available on modern computing units. We exploit SIMD by *operating on multiple active states* simultaneously. Modern CPUs have a SIMD width of 128-bits, and hence we operate on *four* active states simultaneously. Although there exist schemes that exploit SIMD by performing traversal on multiple input streams simultaneously, they do not achieve any speedup on CPUs [8].

For the Single-neighbor `S` case, performing step (a) in SIMD involves doing an *aligned vector load* from the memory into a vector register, while step (b) involves *gathering* the α neighbors for four different states (hence four memory addresses) into a register. Similarly step (c) is another *gather* operation from the `TimeStamp` array, while step(d) is a vector compare (equality between two registers). Finally step (e) needs to be performed only for the SIMD lanes that correspond to states which failed the equality test. Two operations are performed for the same—a masked *scatter* operation to update the `TimeStamp` array, and a *packed vector store* [22] to the `NS` array. Similarly for the `M` case, steps (i) and (ii) translate to vector gather instructions that gather the addresses of the memory locations storing the counter value, followed by another gather to obtain the count values themselves. The remaining operations can utilize SIMD in a similar fashion to `S`.

The speedup due to SIMD is governed by the following 3 factors: (i) efficient hardware support for load, compare, gather, scatter and packed-store vector instructions, (ii) the number of active states that are available for SIMDfication, and (iii) the results of comparison in step (d) that compute the number of elements that need to be scattered and pack-stored.

The current generation of CPUs do not have efficient support for gather, scatter and packed-store instructions. We therefore emulated them using scalar instructions. Hardware support for these instructions would have a greater impact on reduction in runtime. We provide performance numbers for SIMD speedup in Figure 4(a) in Section 6.1. Finally, we note that a kernel implementation reduces the overhead of context switches and user-space copies and serves to further improve performance. As such, we have implemented GPP-grep in the Linux 2.6 kernel.

5.4 Analytical Model

To predict performance we create an analytical model by computing the total number of ops[2] executed during the traversal (for an active state and the input symbol). The corresponding number of cycles depends on memory access patterns. Our layout is also

[2] 1 op implies 1 operation or 1 executed instruction.

optimized for efficient memory access and achieves close to maximal IPC (detailed results in Section 6.2). Thus with appropriate NFA layout optimizations, our analytical model may also project the number of executed cycles.

First consider the Single-neighbor ($\mathtt{S}$) case. For each active state the total cost of the Steps one and two, as outline in Section 5.1, is the sum of the sub-steps (a)–(e) and is termed $\text{cost}_{\text{step1/2}}$. However, step (e) is only executed with certain probability, dependent on the input stream and NFA characteristics. Let ρ denote the corresponding probability. Furthermore, we also issue *software prefetch* instructions to improve the IPC. This adds to the instruction overhead too. Let the cost of prefetches be denoted by $\text{cost}_{\text{pref}}$. Let the number of symbol neighbors be $\mathcal{N}_{\text{symbol}}$ and the ε-closure consist of $\mathcal{N}_{\varepsilon}$ elements. Therefore $\text{cost}_{\text{step1/2}} = \text{cost}_{\text{pref}} + \text{cost}_{(a)} + \text{cost}_{(b)} + \text{cost}_{(c)} + \text{cost}_{(d)} + \rho\text{cost}_{(e)}$, and hence $\text{cost}_{\mathtt{S}} = (\mathcal{N}_{\text{symbol}} + \mathcal{N}_{\varepsilon})\ \text{cost}_{\text{step1/2}}$. For the Multiple-neighbor ($\mathtt{M}$) case, we also need to include the overhead of steps (i) and (ii) (Section 5.3). Hence $\text{cost}_{\mathtt{M}} = (\mathcal{N}_{\text{symbol}} + \mathcal{N}_{\varepsilon})\ \text{cost}_{\text{step1/2}} + \text{cost}_{\text{step(i)}} + \text{cost}_{\text{step(ii)}}$.

For any given architecture, the costs for each of the above terms is known, and can be plugged in to get an estimate of the number of executed ops. We provide data and the projected results by our model in Figure 4(b) in Section 6. We believe the model serves as a metric to compare the performances of different architectures for the NFA traversal with different NFA representations.

6 Experimental Evaluation

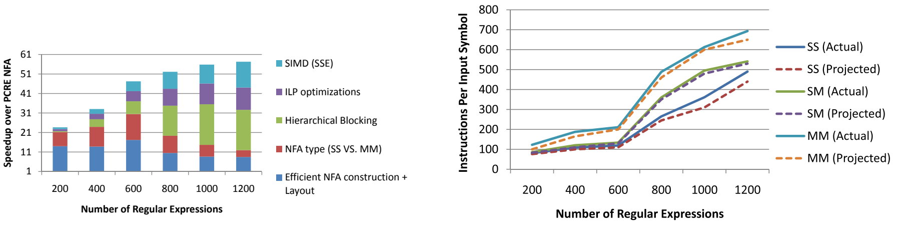
We now evaluate the performance of GPP-grep on the Intel Xeon DP Westmere-EP X5680 CPU. Our CPU platform has 2 sockets with a total of 12 cores running at 3.33 GHz. Our system has 12 GB RAM and runs SUSE Enterprise Edition Linux 11. The peak CPU compute power per socket is 150 Gops (300 Gops on 2 sockets) and achievable memory bandwidth of 44 GBps on 2 sockets.

We collected regular expressions from the Snort rule-sets as provided by the VRT [27]. We choose random subsets consisting of 200, 400, 600, 800, 1000, and 1200 regular expressions from the backdoor, exploit, spyware, web-activex, and web-client rule-sets as the basis for the NFA and our analysis. We employ a packet trace from the 1999 DARPA Intrusion Detection Evaluation Data Sets distributed by the MIT Lincoln Laboratory [9], and simulate a million packets as input data.

We first describe the impact of various algorithmic and architectural optimizations that we performed in GPP-grep. We then discuss the key static and runtime characteristics of NFA traversal and show that they correlate well to the performance model. Finally, we compare the performance of our regular expression matching against other state-of-the-art systems.

6.1 Impact of Optimizations

Figure 4(a) shows the speedup obtained over the baseline PCRE (v8) performance due to the various optimizations with the baseline PCRE performance normalized to 1. We parallelized PCRE for a fair comparison. The speedups obtained from each optimization are *multiplicative* on top of previous optimizations. The lowermost bar gives the



(a) Speedup over PCRE with various optimizations.

(b) Comparison of real instructions executed per input symbol to performance model predictions across MM, SM, and SS NFA.

Fig. 4. Speedup and fit to Analytical Model

impact of our efficient NFA construction that helps reduce the number of active states and efficient layout that enables a constant time neighbor lookup/addition to the active state list as described in Section 5.2. This provides a large benefit of 9–17× over PCRE. The effect of this optimization is primarily to reduce the number of instructions executed. In the absence of architectural optimizations, some of the impact of such instruction reduction weakens for large NFA due to increasing cache and TLB misses and consequently lower IPC. Thus we see a smaller benefit for large regular expression sets. This motivates the need for our architectural optimizations.

The next bar is the impact of picking NFA with multiple transitions per symbol and multiple epsilon transitions (MM) versus a single transition per symbol and epsilon transition (SS). This impact is consistently 1.5–1.8× on top of the previous optimizations. As before, the impact is primarily instruction reduction due to efficient address computation possible in the SS code (Section 5.3).

The next benefits come from an improvement in IPC. The impact of hierarchical NFA blocking increases with NFA size (larger number of regular expressions) up to a maximum of an additional multiplicative impact of 2.7×. As described in Section 5.2, this is due to a decrease in the number of cache and TLB misses. The next IPC optimization is the impact of ILP optimizations such as unrolling and software prefetching described in Section 5.3. This has an impact of 1.1–1.4×, and has more impact for larger NFA (similar to hierarchical blocking). This also includes the effect of SMT (Simultaneous Multi-Threading), which helps hide memory latency when software prefetching techniques do not succeed. This occurs, for instance, when fetching nodes that are in the ε-closure of an active state but not in the neighbor list of the state—we do not prefetch these nodes and rely on SMT.

Finally, we also obtain up to a 1.3× speedup due to the impact of using SIMD instructions. Since we use SIMD to process multiple active states at once, the impact of SIMD increases when we have a relatively large number of active states. This happens for larger NFA that correspond to more regular expressions. Hence our SIMD speedup increases with increasing numbers of regular expressions.

Table 1. Runtime characteristics of NFA traversal

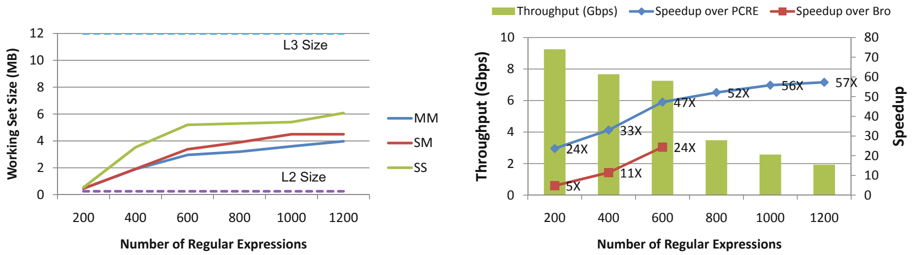
NFA	# REs	#	#	Avg. Active	Max Active	Avg.
Type		states	transitions	States	States	Depth
MM	200	15,883	1,922,997	4.2	24	3.5
	400	43,535	6,729,191	5.4	42	6.1
	600	62,188	9,462,802	7.0	46	8.9
	800	83,965	13,118,674	13.1	81	11.0
	1000	106,815	16,915,665	19.2	82	12.9
	1200	137,927	22,833,917	23.6	94	14.3
SM	200	16,291	1,923,405	4.3	24	3.7
	400	44,336	6,729,992	5.4	42	7.1
	600	63,377	9,463,991	6.9	46	10.2
	800	85,503	13,120,212	13.0	81	13.5
	1000	108,762	16,917,612	18.9	83	15.0
	1200	140,322	22,836,312	23.7	96	18.1
SS	200	16,252	1,957,905	4.3	24	3.7
	400	44,288	6,804,066	5.2	43	8.1
	600	63,313	9,580,679	6.8	47	15.1
	800	85,424	1,327,3108	13.1	82	17.6
	1000	108,731	17,124,534	19.0	84	20.5
	1200	140,491	23,088,684	23.6	98	23.2

6.2 Performance Analysis

Table 1 shows the salient static and runtime characteristics of the MM, SM, and SS NFA. Col. 5 shows average number of active states during traversal. The average number of active states is a good indicator of the number of instructions required to perform the traversal. Col. 7 shows the average depth of NFA states traversed, which gives us an indication of the true working set of traversal. This has implications on the number of cache misses during traversal. Table 1 also illustrates that the number of active states increases with the number of regular expressions. Since the number of active states directly impacts the number of instructions (and hence final performance), we expect to see a drop in performance for larger sets of regular expressions. Further, all three NFA types—MM, SM, and SS—have similar numbers of active states. However, the number of instructions executed for SS is the least because we can use the fact that there is only one neighbor per symbol and only one ε transition to simplify the address calculations during traversal.

Comparison with Performance Model. To compute the number of ops for various operations listed in Section 5.4, we analyzed the static assembly file and hand-counted the number of instructions. These numbers were then plugged in together with the average number of symbol neighbors and nodes in the ε-closure to compute the projected number of instructions. Our projected number matches closely to actual results, only slightly less (5–10%) than the real performance, and is illustrated in Figure 4(b). This is due to the register spills and fills that are not accounted for by our model. Furthermore, our optimized layout results in a per-core IPC of around 1.8 on 12 cores, and hence the resultant run-times are also within 8–12% of the projected run-times (obtained using the maximum per-core IPC of 2).

Working Set Analysis. We measured the working set (data size for which the number of cache hits is $\geq$ 90%), for our MM, SM, and SS NFA types with our input regular expression sets. The working set increases with increasing number of regular expressions



(a) Working sets for MM, SM, and SS NFA with a per-socket shared L3 cache of 12 MB and per-core private L2 cache of 256 KB.

(b) Throughput of GPP-grep and speedup over PCRE and Bro. For fair comparison, we parallelize PCRE and Bro. Bro exceeds memory capacity for >600 regular expressions.

Fig. 5. Working Set and Throughput Evaluation

and is highest for the SS type. Even for the SS type NFA for 1200 regular expressions (our largest input), the working set entirely fits in the L3 cache (12 MB per socket). However, the working set usually does not fit in the L2 caches of the individual cores. Finally, we do see an improvement in IPC using the software prefetches described in Section 5.3. The results of those improvements are illustrated in Figure 5(a).

6.3 Comparison with State of the Art Systems

Figure 5(b) demonstrates the throughput of GPP-grep using the best performing SS NFA. Also, Figure 5(b) shows the speedup of GPP-grep over PCRE and Bro (v.1.5.1) systems. After our optimizations, we obtain a final throughput between 2 and 9.3 Gbps, with a performance of **2 Gbps for 1200 regular expressions**. The absolute performance of GPP-grep drops with increases in the number of regular expressions. This results from increases in the number of active states as the number of regular expressions grows and is shown in Table 1.

We parallelized both PCRE and Bro. PCRE uses NFA with small working sets; these scale perfectly with the number of cores. GPP-grep is 24–57 × faster than parallel PCRE, depending on the number of regular expressions. The speedup improves with larger sets of regular expressions. This is because PCRE provide best performance when handling regular expressions one at a time; hence the runtime is proportional to the number of regular expressions. However, the number of active states in GPP-grep only increases by about 1.5 × from 200 to 600 regular expressions though there is a sudden increase in the number of active states from 600 onwards resulting in a stabilization of the speedup over PCRE. GPP-grep is also 5–24 × faster than Bro. Since the Bro system demonstrated poor results when run one regular expression at a time, we grouped all the regular expressions together. The Bro system adopts a DFA based approach for matching. For greater than 600 regular expressions, the DFA sizes generated by Bro were greater than our system memory of 12 GB. Hence we do not report the resultant performance numbers. Our experiments indicate that the Bro system rapidly becomes bandwidth bound for 400 regular expressions and beyond, since the working set of the DFA does not fit in the L3 cache. Bandwidth bound applications are unable to take advantage of the full computational capabilities of multi-core platforms; we only obtained a parallel scalability of about 3.6× on 12 cores for the 600 regular expression DFA. On the other hand, the working set of the NFA produced by GPP-grep fits in the L3 cache, even for large numbers of regular expressions—hence we are compute-bound and scale near-linearly with the 12 cores of a Xeon X5680. The difference in parallel runtime scaling results in an increase in speedup over Bro as the number of regular expressions increases.

We compare our performance with the FPGA based solution proposed by Mitra et al. [19]. We first note that our single-threaded PCRE performance for 200 regular expressions is similar to their PCRE performance (also for 200 Snort regular expressions) on a single 3.0 GHz Xeon; this indicates that the regular expressions used have similar complexity in terms of active states, making our performance comparisons fair. They report a speedup of 335× over single-threaded PCRE for 200 regular expressions when using FPGAs, while we are about 258× faster than single threaded PCRE on a single CPU (24× faster than PCRE running on 12 cores). Thus our CPU performance is about 1.3× off their FPGA performance using commodity processors. Further, our CPU implementation can scale to a much larger number of regular expressions, while FPGA are more resource limited in terms of on-board memory and do not scale as well to larger sets of regular expressions.

7 Discussion

Our fast regular expression matching algorithm is useful in contexts other than NIDS. In particular, XML queries expressed in XPath [29] often have to be matched against incoming documents in publish/subscribe systems [10] where efficient regular expression matching would prove a boon. Containment queries on trees and graphs wherein the task is to match a tree (or graph) against another tree (or graph), as illustrated by GraphDB [14], pose another potential arena for GPP-grep. Improving NFA traversal can be applied to the general graph traversals used in many contexts including graph searches and graph matching similar to the A* graph search which, in several forms, found use in graph database shortest path searches [14], as well as matches of subgraphs in protein databases, image databases, and software repositories [11].

The performance of our SIMD algorithm for graph traversal would further improve with hardware support for gathers/scatters and packed-store operations. The upcoming Intel MIC (Many Integrated Core) architecture will add such support and should improve SIMD utilization [22]. Coupling this with a kernel implementation is conducive to System-on-Chip (SoC) implementations where packet I/O is combined with matching on a single chip and which makes this approach applicable to current and future trends in regular expression processing. Finally, we note that our algorithm will benefit from any technique that helps further minimize the number of active states during traversal such as: XFA [25] and HFA [2]. Our techniques are complementary and should result in cumulative improvement.

8 Conclusion

We present GPP-grep, a fast regular expression processing engine on commodity general purpose processors. GPP-grep exploits thread-level and data-level parallelism, and employs an architecture-friendly layout and graph traversal scheme to improve efficiency. On a dual-socket commodity CPU system, GPP-grep attains a maximum throughput of 9.3 Gbps, and is up to 57× faster than traditional PCRE engines. In the future we hope to expand this engine to present a single tool for handling NIDS DPI processing of all criteria for any rule, fixed string or regular expression, in one pass. Ultimately, GPP-grep offers an economical solution, both financially and in terms of system resources, for high-speed regular expression matching.

References

1. Becchi, M., Cadambi, S.: Memory-efficient regular expression search using state merging. In: INFOCOM. IEEE (2007)
2. Becchi, M., Crowley, P.: A hybrid finite automaton for practical deep packet inspection. In: CoNEXT. ACM (2007)
3. Becchi, M., Crowley, P.: Extending finite automata to efficiently match perl-compatible regular expressions. In: CoNEXT. ACM (2008)
4. Becchi, M., Wiseman, C., Crowley, P.: Evaluating regular expression matching engines on network and general purpose processors. In: Architecture for Networking and Communications Systems. ACM (2009)
5. Cascarano, N., Rolando, P., Risso, F., Sisto, R.: iNFAnt: NFA pattern matching on GPGPU devices. SIGCOMM Comput. Commun. Rev. 40, 20–26 (2010)
6. Champarnaud, J.-M., Coulon, F.: NFA reduction algorithms by means of regular inequalities. Theoretical Computer Science 327(3), 241–253 (2004)
7. Champarnaud, J.-M., Coulon, F.: Erratum to NFA reduction algorithms by means of regular inequalities. Theoretical Computer Science 347(1-2), 437–440 (2005)
8. Chong, J., You, K., Yi, Y., Gonina, E., Hughes, C., Sung, W., Keutzer, K.: Scalable HMM-based inference engine in large vocabulary continuous speech recognition. In: International Conference on Multimedia and Expo. IEEE Press (2009)
9. Cunningham, R.K., Lippmann, R.P., Fried, D.J., Garfinkel, S.L., Graf, I., Kendall, K.R., Webster, S.E., Wyschogrod, D., Zissman, M.A.: Evaluating intrusion detection systems without attacking your friends: The 1998 DARPA intrusion detection evaluation. In: Intrusion Detection and Response (1999)
10. Diao, Y., Altinel, M., Franklin, M.J., Zhang, H., Fischer, P.M.: Path sharing and predicate evaluation for high-performance XML filtering. Trans. on Database Systems 28, 467–516 (2003)
11. Djoko, S., Cook, D.J., Holde, L.B.: An empirical study of domain knowledge and its benefits to substructure discovery. Trans. on Knowledge and Data Engineering 9, 575–586 (1997)
12. Dreger, H., Feldmann, A., Paxson, V., Sommer, R.: Operational experiences with high-volume network intrusion detection. In: Computer and Communications Security. ACM (2004)
13. Gramlich, G., Schnitger, G.: Minimizing NFA's and regular expressions. J. Comput. Syst. Sci. 73, 908–923 (2007)
14. Güting, R.H.: GraphDB: Modeling and querying graphs in databases. In: Very Large Data Bases. Morgan Kaufmann Publishers Inc. (1994)
15. Handley, M., Paxson, V., Kreibich, C.: Network intrusion detection: Evasion, traffic normalization, and end-to-end protocol semantics. In: USENIX Security. USENIX (2001)
16. Kumar, S., Dharmapurikar, S., Yu, F., Crowley, P., Turner, J.: Algorithms to accelerate multiple regular expressions matching for deep packet inspection. In: SIGCOMM. ACM (2006)

17. Kumar, S., Turner, J., Williams, J.: Advanced algorithms for fast and scalable deep packet inspection. In: Architecture for Networking and Communications Systems. ACM (2006)
18. Meiners, C.R., Patel, J., Norige, E., Torng, E., Liu, A.X.: Fast regular expression matching using small TCAMs for network intrusion detection and prevention systems. In: USENIX Security. USENIX (2010)
19. Mitra, A., Najjar, W., Bhuyan, L.: Compiling PCRE to FPGA for accelerating Snort IDS. In: Architecture for Networking and Communications Systems. ACM (2007)
20. Pasetto, D., Petrini, F., Agarwal, V.: Tools for very fast regular expression matching. IEEE Computer 43(3), 50–58 (2010)
21. Scarpazza, D.P., Russell, G.F.: High-performance R.E. scanning on the Cell/B.E. processor. In: International Conference on Supercomputing, pp. 14–25. ACM (2009)
22. Seiler, L., Carmean, D., Sprangle, E., Forsyth, T., Abrash, M., Dubey, P., Junkins, S., Lake, A., Sugerman, J., Cavin, R., Espasa, R., Grochowski, E., Juan, T., Hanrahan, P.: Larrabee: A Many-Core x86 Architecture for Visual Computing. ACM Trans. Graph. 27(3), 18:1–18:15 (2008)
23. Shenoy, G.S., Tubella, J., Gonzalez, A.: A performance and area efficient architecture for intrusion detection systems. In: Parallel & Distributed Processing Symposium. IEEE Computer Society (2011)
24. Smith, R., Estan, C., Jha, S.: XFA: Faster signature matching with extended automata. In: Security and Privacy. IEEE Computer Society (2008)
25. Smith, R., Estan, C., Jha, S., Kong, S.: Deflating the big bang: Fast and scalable deep packet inspection with extended finite automata. In: SIGCOMM. ACM (2008)
26. Smith, R., Goyal, N., Ormont, J., Sankaralingam, K., Estan, C.: Evaluating GPUs for network packet signature matching. In: Performance Analysis of Systems and Software. IEEE (2009)
27. Sourcefire Vulnerability Research Team: Sourcefire Vulnerability Research Team (VRT) Snort Rule-set, 2.9.0 edn. (August 2011), http://www.snort.org/vrt
28. Thompson, K.: Programming techniques: Regular expression search algorithm. Commun. ACM 11, 419–422 (1968)
29. XML path language (XPath) 2.0. W3C Recommendation (2007), http://www.w3.org/TR/xpath20/
30. Yang, L., Karim, R., Ganapathy, V., Smith, R.: Improving NFA-Based Signature Matching Using Ordered Binary Decision Diagrams. In: Jha, S., Sommer, R., Kreibich, C. (eds.) RAID 2010. LNCS, vol. 6307, pp. 58–78. Springer, Heidelberg (2010)
31. Yu, F., Chen, Z., Diao, Y., Lakshman, T.V., Katz, R.H.: Fast and memory-efficient regular expression matching for deep packet inspection. In: Architecture for Networking and Communications Systems. ACM (2006)

N-Gram against the Machine: On the Feasibility of the N-Gram Network Analysis for Binary Protocols

Dina Hadžiosmanović[1], Lorenzo Simionato[2,*], Damiano Bolzoni[1], Emmanuele Zambon[1], and Sandro Etalle[1,3]

[1] University of Twente, The Netherlands
[2] Ca' Foscari University of Venice, Italy
[3] Technical University of Eindhoven, The Netherlands

Abstract. In recent years we have witnessed several complex and high-impact attacks specifically targeting "binary" protocols (RPC, Samba and, more recently, RDP). These attacks could not be detected by current – signature-based – detection solutions, while – at least in theory – they could be detected by state-of-the-art *anomaly-based* systems. This raises once again the still unanswered question of how *effective* anomaly-based systems are in practice. To contribute to answering this question, in this paper we investigate the effectiveness of a widely studied category of network intrusion detection systems: anomaly-based algorithms using *n-gram analysis* for payload inspection. Specifically, we present a thorough analysis and evaluation of several detection algorithms using variants of n-gram analysis on real-life environments. Our tests show that the analyzed systems, in presence of data with high variability, cannot deliver high detection and low false positive rates at the same time.

1 Introduction

While most of the current commercial network intrusion detection systems (NIDS) are *signature-based*, i.e., they recognize an attack when it *matches* a previously defined signature, there is a large body of literature on *anomaly-based* detection. An anomaly-based NIDS raises an alert when the observed input *does not match* the behavior that was previously observed; the underlying assumption being that attack payloads "look different" than normal network traffic. In principle, anomaly-based NIDS have *one great advantage* over signature-based ones: they can detect threats for which there exists no signature yet, including zero-day and targeted attacks. Targeted attacks are so complex and evasive that *by definition* cannot be detected by signature-based systems (false negative problem). One famous example of targeted attack is Stuxnet [14], a malware designed to hit specific embedded systems used in Iranian installations for uranium enrichment, discovered in the late 2010. The subsequent analysis revealed that the malware

[*] The author carried out part of this work while he was a visiting student at the University of Twente. Current affiliation: Google Inc.

D. Balzarotti, S. Stolfo, and M. Cova (Eds.): RAID 2012, LNCS 7462, pp. 354–373, 2012.

exploits two previously unknown vulnerabilities in network services. Thus, while the final target was a component typical of industrial control systems, (some of) the vulnerabilities aimed at infecting local computers. Hence, those vulnerabilities could have been used to attack “regular” business and home systems. Other attacks specifically designed to target Industrial Control Systems (ICS - which includes of nuclear power plants, oil and gas extraction and distribution facilities) have been disclosed recently [3].

Given that signature-based systems are ineffective against targeted and zero-day attacks and that most likely there exists no signature yet for the great majority of the attacks that one can buy on the black market, an effective anomaly-based NIDS would be the silver bullet thousands of enterprises and governments are looking for.

Problem & Contribution. Although the field of anomaly detection is well established in research, to date there are only few actual deployments of anomaly-based NIDS worldwide. A common reason used for explaining this is that such systems show poor performance with respect to false positive rate in real-life environments. More generally, Sommer and Paxson [29] argue that many machine learning approaches (which are typically used in anomaly-based IDS) are not effective enough for real-life deployments.

With this paper we want to shed a new light on the detection capabilities of anomaly-based NIDS for payload inspection. We do so by focusing on systems employing a form of n-gram-analysis as anomaly detection engine. To perform the analysis, we apply selected algorithms to environments that widely utilize binary protocols. Specifically,

- We perform thorough benchmarks using real-life data from binary-based protocols, which have been lately targeted by high-impact cyber attacks,
- We include in the analysis a protocol that is specific for Industrial Control Systems (also known as “SCADA”),
- We analyze and discuss the reasons why certain attack instances are (not) detected by the chosen approaches,
- We discuss the feasibility of deploying such approaches in real-life environments, in particular w.r.t. the false positive rate, an issue that is seldom discussed by authors in their work.

Our experiments show that n-gram analysis cannot be indiscriminately applied to the whole network stream data, and that data with high variability are difficult to model and analyze, confirming the conclusion of some earlier work on the matter (see Section 6 for a more detailed discussion).

2 Preliminaries

In this section we introduce the concepts and the terminology that will be used in the remaining of the paper.

2.1 Anomaly-Based Network Intrusion Detection Systems

A detection system can use different sources to extract data features, namely network traffic or system/application activities. In this embodiment we focus on network-based approaches. These approaches monitor the network traffic in a transparent way, without affecting the host performance and are thus often preferred over host-based approaches. There are two types of anomaly-based NIDS: (1) systems that analyse network flows and (2) systems that analyse the actual payload. A flow-based approach takes into consideration features such as the number of sent/received bytes, the duration of the connection and the layer-4 protocol used. A payload-based approach considers features of individual packet or communication payloads. Despite being complementary, the payload-based approaches offer higher chances of detecting a broader set of threats and, unlikely flow-based, these approaches can capture "semantic attacks". Semantic attacks exploit "a specific feature or implementation bug of some protocol or application installed at the victim" [25]. In addition, most of the top security risks (such as in the "OWASP Top 10" [33]) require the injection of some data to exploit the vulnerability. Thus, we focus on payload-based approaches.

2.2 Binary Protocols

In contrast to text-based network protocols (such as HTTP, POP and SMTP), binary protocols are designed to be processed by a computer rather than a human. Such protocols are largely used in network services, such as distributed file systems, databases, etc. In practical terms, the network payload of a binary protocol is more compact if compared to text protocols, often unreadable by a human and may resemble to attack payloads (since malware packets often consists of binary fragments too). Due to these reasons the challenge of detecting attacks in binary-based data is typically greater than in text-based data.

2.3 N-Gram Analysis

N-gram analysis is a common technique for capturing features of data content. This technique is used in various areas, such as monitoring system calls [16], text analysis [10], packet payload analysis [35]. In the context of network payload analysis, the current approaches use the concept of n-grams in different ways. In particular, we distinguish two aspects:

1. The way an n-gram builds feature space - The extracted n-grams can be used for building different feature spaces [12]: (a) count embedding (count the number of different n-grams to describe the payload), (b) frequency embedding (use relative frequency of byte values of an n-gram to describe the payload, e.g. [1,6,36]) and (c) binary embedding (use the presence/absence of specific n-grams to describe the payload, e.g., [35]).
2. The accuracy of payload representation - N-grams can represent the payload in the following ways: (a) as an exact payload description (n-grams represent

continuous sequences of bytes, e.g. in [6,35,36]) and (b) as an approximated payload description (n-grams represent a compression or a reduction of the exact payload, e.g., [17,28]).

Also, various systems employ different architectures and combinations of approaches to analyze n-grams (e.g., Markov models in [1], Self-Organizing Maps in [6], hashing in [17]).

For performing our benchmarks we choose algorithms that are conceptually different in the way the n-gram analysis is performed. Unfortunately, our choice is also limited to the availability of implementations and the level of details in algorithm descriptions.

2.4 Description of Analysed Systems

In the remaining of the section we introduce four algorithms that we select for testing: PAYL, POSEIDON, Anagram and McPAD. These algorithms are 1) general-purpose enough to be used with multiple application-level protocols, 2) proposed by often cited papers in the IDS community or 3) claiming to improve over the previous ones. Each algorithm requires as an input only the incoming network traffic, and does not perform any correlation between different packets.

PAYL. Wang and Stolfo in [36] present their 1-gram-based payload anomaly detector (PAYL). The system detects anomalies by combining 1-gram analysis algorithm with a classification method based on clustering of packet payload data length. The system employs a set of *models*: a model stores incrementally the resulting values of the 1-gram analysis for packet payloads of length l, thus each payload length has a different model. Each model stores two data series: mean byte frequency (i.e., relative byte frequencies span across several payloads of length l) and byte frequency standard deviation for each byte value (i.e., how relative byte frequencies change across payloads). During the detection phase, the same values are computed for incoming packets and then compared to model values: a significant difference from the model parameters produces an alert.

When PAYL fails to detect an attack. Fogla et al. [15] show that PAYL's detection can be evaded by mimicry attacks. PAYL is vulnerable to mimicry attacks since it models only 1-gram byte distributions. By carefully crafting an attack payload, an attacker is able to deceive the algorithm with additional bytes, which are useless to carry on the attack, but match the statistics of normal models.

POSEIDON. Bolzoni et al. present POSEIDON [6], a system built upon a modified PAYL architecture. PAYL uses data length field for choosing the right model. By contrast, POSEIDON employs a neural network to classify packets (and thus choose the most similar model) during the preprocessing phase. The authors use Self-Organizing Maps (SOMs) [19] to implement the unsupervised clustering. First, the full packet payload is analyzed by the SOM, which returns the value of the most similar neuron. That neuron model is then used for the calculation of byte frequency and standard deviation values, as in PAYL.

When POSEIDON fails to detect an attack. Differently from PAYL, POSEIDON is more resilient to mimicry attacks due to the combination of SOM and PAYL. The SOM analyzes the input by taking into consideration byte value at i-th position within the whole payload. Thus, extra bytes inserted by the attacker would be taken into consideration as well, resulting in a different classification than normal traffic. However, the granularity of the classification done by the SOM is coarse. Thus, if the attack portion of the sample payload is small enough, then the sample could be assigned to one of the clusters containing models of regular traffic, and may go unnoticed because of a similar byte frequency distribution.

Anagram. Wang et al. [35] present Anagram. The basic idea behind Anagram is that the usage of higher-order n-grams (i.e., n-grams where $n > 1$) helps to perform a more precise analysis. However, the memory needed to store average and standard deviation values for each n-gram grows exponentially (256^n, where n is the n-gram order). For instance, 640GB of memory would be needed to store 5-grams statistics. To solve this issue, the authors propose to use a binary-based n-gram analysis and store the occurred n-grams efficiently in a Bloom filters [5]. The binary-based approach implies a simple recording of the presence of distinct n-grams during training. Since less information is stored in the memory, it becomes possible to effectively use higher-order n-grams for the analysis. Authors show that this approach is more precise than the frequency-based analysis (e.g., used in PAYL) in the context of network data analysis. This is because higher-order n-grams are more sparse than low-order n-grams, and gathering accurate byte-frequency statistics becomes more difficult as the n-gram order increases.

When in detection mode, the current input is ranked using the number of previously unseen n-grams.

When Anagram fails to detect an attack. There are two main reasons why Anagram may fail to detect attack attempt. Firstly, the Bloom filter could saturate during training. This is because the user may underestimate the number of unique n-grams and allocates a small Bloom filter, during testing any n-gram would be considered as normal. Secondly, Anagram will likely miss the detection if the attack leverages a sequence of n-grams that have been observed during testing.

McPAD. Perdisci et al. present "Multiple-classifier Payload-based Anomaly Detector" (McPAD) [28] with a specific goal of an accurate detection of shellcode attacks. The authors use a modified version of the 2-gram analysis, combined with a group of one-class Support Vector Machine (SVM) classifiers [34]. The 2-gram analysis is performed by calculating the frequency of bytes that are

ν positions apart from each other. By contrast, a typical 2-gram analysis measures the frequency of 2 consecutive bytes. By varying the parameter ν, McPAD constructs several representation of the payload in different feature spaces. For example, for ν=*0..m*, McPAD builds m different representations of the packet payload. When in testing mode, a packet is flagged as anomalous if a combination (e.g., majority) of SVM outputs acknowledge the payload as anomalous.

When McPAD fails to detect an attack. By design, McPAD tries to give a wide representation of the payload (i.e. add more context by constructing byte pairs that are several positions apart). This may represent a difficulty in two cases. First, this is an approximate representation and that may imply a poorly described payload in case of slight differences between the training sample and an attack [1]. This may lead to a high false positive rate and a low detection rate. Secondly, McPAD uses different classifiers that have to come into an "agreement" to decide if a particular packet is anomalous or not. A problem may arise when, due to an approximate payload representation, several classifiers are misled by the byte pair representation and result in outvoting "correct" classifiers. In such case, the system might miss the detection.

3 Approach

We believe that one of the main reasons for poor performance of anomaly-based NIDS lies in the intrinsic limitation of commonly applied algorithm for content analysis: *n-gram analysis*. Since performing a comprehensive test to verify the ability of an IDS of identifying (all) attacks and to spot its weaknesses is unfeasible [22], we proceed to experimentally address our claim. We present a comparative analysis and evaluate the effectiveness of anomaly-based algorithms that analyse network payloads by using some form of n-gram analysis.

To verify the effectiveness of different algorithms we execute a number of steps: 1) collect network and attack data, 2) obtain a working implementation of each algorithm, 3) run the algorithms and analyse the results.

Obtaining the data. We acknowledge that optimal conditions for evaluating the performance of an IDS consist of running tests on unprocessed data from real networks [2]. Thus we first collect real-life data from different network environments, which are currently being operated. The past research is typically focused on benchmarking the algorithms with the HTTP protocol, although the authors do not explicitly restrict the scope of their algorithms to this protocol. We focus on the analysis of binary protocols. In particular, we analyse an example of a binary protocol found in a typical LAN (such as a Windows-based network service) and an example of a binary protocol typically found in an ICS.

Windows is the most used OS in the world, and every instance runs by default certain network services that are often used within LANs. For instance, the SMB/CIFS protocols [20] are used to exchange files between two computers, while other services (e.g. RPC) run on the top of it to provide additional functionalities. Although Windows systems are usually secured against abuses of such service from the Internet, corporate users take advantage of this feature quite often. An attacker that would develop an exploit for a zero-day vulnerability leveraging this protocol could potentially affect a large number of systems. In the last decade several malware [8,14] exploited SMB/CIFS to operate botnets and carry on other malicious activities.

As described in the introduction, ICS have lately become a valuable target for cyber attackers. Considering the sensitive character of such environments, the detection of cyber attacks plays a crucial role, sometimes even in homeland security. Lately, we have witnessed an increasing number of vulnerabilities discovered in software used in critical facilities, mainly due to the poor software development cycles that several vendors adopted, and the "security by obscurity" paradigm used to "protect" legacy devices.

We collect attacks in two different datasets. Our focus is on data injection attacks that have a high impact (see [27]).

Obtaining the implementations. Secondly, to carry out the benchmarks, we need working prototypes of all the algorithms we want to test. We could obtain an implementation of POSEIDON and McPAD from the authors. For the other two algorithms, we write our own implementation[1] based on the description found in the papers. To be sure that our implementation is correct, we need to verify that our results resemble the ones shown in the benchmarks of the respective original papers.

Analysing the results. The last step of our evaluation is the analysis of results with a focus on the identification of reasons for (un)successful detection.

3.1 Evaluation Criteria

The effectiveness of an IDS is mainly determined by the detection and false positive rates. The detection rate indicates the number of attack instances correctly identified by the IDS (true positives), w.r.t. the total number of attack instances. The false positive rate indicates the amount of samples that the IDS flags as attacks when they are actually not. False positives are a major limiting factor in this domain because, differently from other classification problems, their cost is high [4].

Detection rate. To provide a detailed overview of the detection capabilities of each algorithm, we consider both the number of correctly detected packets in the attack set and the number of detected attack instances. In fact, not all attacks show malicious payload within one single packet. Although an algorithm that exhibits a high per-packet detection rate has a higher chance of detecting attack instance, we do not argue that a low per-packet detection rate implies an equivalently low per-instance detection. In summary, we consider an alarm

[1] We intend to disclose the implementations in near future.

as a true positive if the algorithm is able to trigger at least one alert packet per attack instance.

False positive rate. The usual approach to document the performance of an IDS is to relate the false positive rate with the detection rate. This is done by drawing so called Receiver Operating Characteristic (ROC) curves. The benchmarks from the original papers of the proposed algorithms express the false positive rate as a percentage. However, such number has little meaning to the final users. A better way to express the false positive rate is in terms of the number of false positives per time unit. We establish two different thresholds: 10 false positives per day and 1 false positive per minute. The former value is proposed in [21] as a reasonable number for a user to maintain trust in the system. The latter is, in our opinion, the highest rate at which a human can verify alerts generated by an IDS. It is worth noting that anomaly-based IDSes, unlikely signature-based ones, do not provide information regarding the attack classification. Thus, the user might require additional time to investigate whether the alert is a true or a false positive. For each data set we compute these two thresholds based on the number of actual packets included in the verification sub data set after having split the original data set.

Since we do not make the data sets attack-free before hand, and thus some "noise" could have been collected as well, we need a way to verify that the alerts generated while processing the verification sub data set are actual false positives. To do that, we use a signature-based IDS (the popular open-source Snort), which is automatically fed with the network stream for which an alert was triggered during verification.

Commonly, in IDS evaluation papers the authors stop their analysis by reporting on the true and false positive rates. We believe that inspecting which attacks are detected, which are not detected and why, would provide useful information to fully understand when an algorithm could perform better than others (and for which threats). This kind of analysis can provide insights for future improvements. Finally we aim at evaluating the effectiveness of combining diverse algorithms to boost the detection rate.

4 Description of Network Data

In this section we describe in detail the background data set and attack data sets that will be used for benchmarking the detection algorithms. The chosen data sets comprise network traffic taken from two environments. We use the publicly available vulnerabilities and high impact exploits to run the tests.

4.1 Web Data Set

The following data sets are a collection of network traces of web traffic (in particular, the HTTP protocol).

DS_{DARPA}. The DARPA 1999 data set [21] is a standard data set used as reference by a number of researchers. Despite being anachronistic (and criticized in several works [23]), three of the algorithms we test have used this data set to compare their performance to previous works. Thus, we use the DARPA data set to verify that our own implementations of PAYL and Anagram offer comparable detection and false alert rates with the tests reported in the research papers of the detection algorithms.

AS_{HTTP}. This attack set is presented by Ingham and Inoue in [18], and has been used also by the authors of McPAD for their benchmarks. It comprises 66 diverse attacks, including 11 shellcodes, which were collected from public attack archives. The attacks are instances of buffer overows, input validation errors (other than buffer overflows), signed interpretation of unsigned values and URL decoding errors.

4.2 LAN Data Set

DS_{SMB} This data set includes network traces from a large University network. Samples have been collected through a week of observations. The average data rate of incoming and outgoing packets is ∼40Mbps.

In particular, we focus on the SMB/CIFS protocol, and even more on SMB/ CIFS messages which encapsulate RPC messages (see Section 5.2). The average packet rate for this traffic is ∼22/sec. Based on this we calculate the false positive rate threshold for obtaining 10 alerts per day as 0.0005% and the one for obtaining 1 alert per minute as 0.073%.

AS_{SMB}. This attack data set is made of seven attack instances which exploit four different vulnerabilities in the Microsoft SMB/CIFS protocol: *ms04-011*, *ms06-040*, *ms08-067* and *ms10-061* [7].

ms04-011 is a vulnerability of certain Active Directory service functions in LSASRV.DLL of the Local Security Authority Subsystem Service (LSASS) of several Microsoft Windows versions. We select this vulnerability because it is used by the worm Sasser [26]. We collect two different attack instances for this vulnerability. One trace is downloaded from a public repository of network traces [13] where the attack payload is split in three fragments and contains a shellcode of 3320 bytes. The shellcode is made of a number of NOP instructions (byte value `0x90`), followed by valid x86 instructions and a sequence of the ASCII character '1'. The second instance is generated through the Metasploit framework [24]. The attack payload is split into three fragments and contains a shellcode of 8204 bytes to remotely launch a command shell in the victim host.

m06-040 is a vulnerability of the Microsoft Server RPC service. In particular, the vulnerability allows a stack overflow during the canonicalization of a network resource path. The specified path can be crafted to execute arbitrary code after the exploitation. We collect the attack instance from a public repository of network traces [13].

ms08-067 is a vulnerability of the Microsoft Server RPC service which exploits a similar weakness as the one described in *m06-040*, with the same effects. We select this vulnerability because it is used by Conficker [8] and Stuxnet, two high-impact pieces of malware. We collect two different attack instances for this vulnerability. One instance was downloaded from a public repository [13], while the other one was generated by us using the metasploit framework. In the first instance the payload contains a shellcode of 684 bytes, while in the second instance the shellcode is 305 bytes long only.

ms10-061 is a vulnerability of the PrintSpooler RPC service. When printer sharing is enabled, the PrintSpooler service does not properly validate spooler access permissions. Remote attackers can create files in a system directory, and consequently execute arbitrary code, by sending a crafted print request over RPC. We select this vulnerability because it was used by Stuxnet to successfully propagate in both regular backoffice LANs as well as in industrial control system environments. We collect two different attack instances for this vulnerability, both of them are generated through the metasploit framework. In one instance the attack payload is a binary file (the meterpreter executable), which accounts for 69832 bytes spanned over 18 fragments, while in the other instance the payload consists of a DLL file, which accounts for 1735 bytes spanned over three fragments.

4.3 ICS Data Set

DS_{Modbus} To test the anomaly detection algorithms on ICS networks we collect a data set of traces from the industrial control network of a real-world plant over 30 days of observation. The average throughput on this network is ∼800Kbps.

This data set includes network traces of one of the most common protocols used in such environments, Modbus/TCP [32]. Modbus was developed more than 30 years ago initially as a protocol used in serial channels, while the TCP/IP variant was introduced approximately 15 years ago to allow the serial protocol to be used in TCP/IP networks. Modbus/TCP features basic instructions and functions. Its structure is relatively simple, and operators of critical infrastructures usually repeat a limited set of operations, thus reducing the variability of the transmitted data. In fact, the maximum size of a Modbus/TCP message is 256 bytes. Thus, a Modbus/TCP message is always contained in one single TCP segment. Observations on DS_{Modbus} reveal that the average size of Modbus/TCP messages is 12.02 bytes (in DS_{SMB} it is 535.5 bytes). In the observed DS_{Modbus}, the number of duplicated TCP segment payloads is high (96.08%), in contrast to the other data set (e.g., DS_{SMB} has 31.37%). In other words, more than 9 TCP segments out of 10 carry a Modbus/TCP message that is a perfect duplicate of some other message already observed. The average packet rate for Modbus incoming traffic is ∼96/sec. Based on this observation, we calculate the false positive rate threshold for generating 10 alerts per day as 0.00012% and the threshold for generating 1 alert per minute as 0.017%.

AS_{Modbus}. The attack data set is made of 163 attack instances, which exploit diverse vulnerabilities of the Modbus/TCP implementations. There are fewer publicly known attacks against Modbus/TCP devices than SMB/CIFS attacks. Network traces for a good deal of these attacks can be downloaded from the website of an ICS security firm [11]. The exploited vulnerabilities can be categorised in two large families: *unauthorised use* and *protocol errors*.

Unauthorized use consist of two attack types: (a) "weird" clients talking to the Modbus server and (b) messages used only for diagnostics and special maintenance, which are thus seldom seen in the network traffic. By issuing these special messages, the attacker is often able to achieve a complete take over of the device.

Protocol errors are mainly fuzzing attempts against a device. For instance, these attacks are carried out by sending data not compliant with the protocol specifications (e.g. a too short protocol data unit). The outcome of such attacks can range from the unavailability of the device up to the control of the execution flow (see Cui and Stolfo [9] for a more detailed discussion).

5 Benchmarks

In this section we show the results of our benchmarks and compare the performances of the algorithms for each data set.

Setting up and tuning the algorithms. For each dataset we split the background traffic into two sets, one for training and one for verification. The splitting is performed randomly, by sampling the network streams. Each split sub data set accounts for nearly 50% of the original data. The training sub data set is used to build the detection profiles for each algorithm, while the verification one is used to evaluate the number of false positives raised by the algorithm. Finally, we run the algorithms on the attack data set.

For performing the benchmarks we need to set up several starting parameters for each algorithm.

PAYL. As introduced in the original paper, the size of the PAYL model can be reduced by merging profiles when their number becomes too large. For each data set we perform several runs using different values of the merging parameter. Finally, for the DS_{DARPA} data set we set the parameter value to 0.12. For the remaining data sets we do not merge profiles, as the total number of profiles remains low (up to 150, compared to 480 in DS_{DARPA}). As the "smoothing factor", we use the same value (0.001) suggested by Ingham and Inoue in [18]. We apply the algorithm to individual TCP segment payloads.

Anagram. For each data set we run tests with different n-gram sizes (n size of 3, 5, 7, 9 and 12). Since we obtain the best results with the 3-grams, we set this as the standard n-gram size. As in the original paper, we set the size of the Bloom filter to 2MB. We do not use the "bad content model" proposed by the authors because it would be ineffective as they build it with virus samples, and our attack data sets do not include viruses.

POSEIDON. For all tests we use a SOM with fixed number of neurones (96). Also, we set the number of instances for training the SOM at 10000.

McPAD. For all tests we use the best performing parameters as proposed in [28]. Those are: number of clusters $k = 160$, desired false positive rate for each SVM classifier set to 1% and *maximum probability* as combination rule for the output of the SVM classifiers.

For each algorithm we vary the value of the threshold to observe how the false positive and detection rates change.

5.1 Implementation Verification

To verify the correctness of our implementations we run initial tests using DS_{DARPA} data set (since that was the only common data set used in 3 original algorithm benchmarks). Instead of using DARPA attack dataset, we use the AS_{HTTP} for testing. There are two main reasons for this: (1) the original attack instances of the DARPA data set do not reflect at all modern attacks, and (2) not all the algorithms have been benchmarked against the attack set of DARPA (Anagram is the exception). Thus, it would be impossible to faithfully reproduce the previous experiments. In Table 1 we summarize the results of this first round of benchmarks: for each algorithm we report the highest detection rate we achieve, and the corresponding false positive rate.

Table 1. Test results on DS_{DARPA} and testing with AS_{HTTP}

	FPR (packet-based)	DR (packet-based)
PAYL	0.00%	90.73%
POSEIDON	0.004%	92.00%
Anagram	**0.00%**	**100.00%**
McPAD	0.33%	87.80%

All the algorithms show high detection and low false positive rates. When compared to the original papers, these results match our expectations, thus we can be reasonably sure that our re-implementations are not (too) dissimilar from the original ones in terms of completeness and accuracy.

5.2 Tests with LAN Data Set

We first perform the test on DS_{SMB} by using all the SMB/CIFS packets directed to the TCP ports 139 or 445. However, none of the algorithm can perform well enough under these conditions. For example, the Bloom filter used by Anagram saturates with 3-grams during the training phase. Consequently, no attack instance is further detected, even with the lowest threshold. Increasing either the size of n-grams or the size of the Bloom filter cannot solve the issue of undetected attacks. In the former case, the Bloom filter saturates even with fewer training packets. In the latter case, even with a filter size of 20MB (10 times bigger than the size suggested by the authors) which does not cause full saturation, no attack is detected with false positives rates lower than 4%. Other algorithms exhibit similar detection problems, with at least one attack instance detected only with false positive rates higher than 1%.

This result alone would imply a complete failure for this protocol. We believe the reason why the algorithms poorly perform on SMB/CIFS is because of the high variability of the analyzed payload. In fact, SMB/CIFS is mainly used to transfer files between Windows computers. Such files are contained in the request messages processed by the algorithms and can be of any type, from simple text files to compressed binary archives or even encrypted data.

We acknowledge that all attack instances publicly available exploit vulnerabilities of the Windows RPC service by leveraging SMB/CIFS, which can encapsulate RPC messages. Thus, we re-run the test on a filtered data set. In particular, we extract only SMB/CIFS messages that carry RPC data (∼1% of the original SMB/CIFS traffic). By doing this, we are implicitly providing a semantical hint to the algorithms. We expect this to improve both the detection and false positive rates.

The results of this round of tests are summarized on Table 2. Anagram achieves a 0.00% false positive rate while still being able to detect 3 attack instances. Anagram also achieves the lowest false positive rate among the tested algorithms when detecting all of the attack instances, a rate lower than the adjusted false positive of 1 alert per minute. McPAD generates the highest false positive rate, and it is impossible to lower that no matter which combination of parameters we choose. We believe that the main reason for this lies in the fact that McPAD implements the approximate payload representation, that, in such variable conditions, provides poor payload description.

There is no need to evaluate how a combined approach, i.e., using all the algorithms simultaneously, would perform since Anagram is already outperforming the other algorithms.

Finally, we process all false positives with SNORT to verify that none of them is actually a true positive.

Analysis of detected and undetected attacks. By considering which attack instance is detected the most, we observe that all the algorithms can detect an attack instance exploiting the *ms04-011* vulnerability. In particular, of the three segments composing the attack payload, only one is always identified as anomalous. Here we report a fragment of it:

```
0230   59 35 1c 59 ec 60 c8 cb cf ca 66 4b c3 c0 32 7b  Y5.Y.‘....fK..2{
0240   77 aa 59 5a 71 76 67 66 66 de fc ed c9 eb f6 fa  w.YZqvgff.......
0250   d8 fd fd eb fc ea ea 99 da eb fc f8 ed fc c9 eb  ................
0260   f6 fa fc ea ea d8 99 dc e1 f0 ed cd f1 eb fc f8  ................
0270   fd 99 d5 f6 f8 fd d5 f0 fb eb f8 eb e0 d8 99 ee  ................
0280   ea ab c6 aa ab 99 ce ca d8 ca f6 fa f2 fc ed d8  ................
0290   99 fb f0 f7 fd 99 f5 f0 ea ed fc f7 99 f8 fa fa  ................
02a0   fc e9 ed 99 fa f5 f6 ea fc ea f6 fa f2 fc ed 99  ................
02b0   90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90  ................
02c0   90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90  ................
02d0   90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90  ................
02e0   90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90  ................
02f0   90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90  ................
0300   90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90  ................
...
05a0   90 90 90 90 90 90 90 90                          ........
```

Table 2. Test results on DS_{SMB} and testing with AS_{SMB}

	FPR (packet-based)	DR (packet-based)	DR (instance-based)
PAYL	0.004%	1.43%	2/7
	0.007%	3.35%	3/7
	0.01%	6.65%	4/7
	0.05%	23.92%	5/7
	4.41%	66.51%	6/7
	11.05%	85.02%	7/7
POSEIDON	0.007%	6.22%	2/7
	0.007%	10.04%	3/7
	1.27%	37.32%	4/7
	2.28%	50.23%	6/7
	3.32%	58.37%	6/7
	5.39%	66.98%	7/7
Anagram	0.00%	0.95%	2/7
	0.00%	22.48%	3/7
	0.02%	37.32%	7/7
	2.34%	55.50%	7/7
	8.39%	63.64%	7/7
McPAD	19.02%	60.38%	7/7
	20.62%	60.86%	7/7
	22.31%	61.35%	7/7
	27.61%	65.21%	7/7
	97.39%	90.00%	7/7

We verify that this particular segment is flagged as anomalous by all the algorithms because of the long sequence of the byte value `0x90`, which corresponds to the NOP instruction of the shellcode. Although there are individual bytes with value `0x90` in the training set, we verify that there is never a sequence of three bytes with this value. This explains why Anagram can identify as anomalous a vast majority of the 3-grams in the aforementioned payload. Also PAYL and POSEIDON will identify an anomalous byte frequency distribution, in which the byte value `0x90` peaks above all the others. Similarly, the fact that the payload consists of continuous `0x90` bytes implies that McPAD classifiers will be able to recognize the peak in the frequency of these 2-grams.

We also observe that both PAYL and POSEIDON fail to detect one attack instance exploiting the *ms06-040* vulnerability, when the false positive rate is below 2%. A fragment of the payload of such attack instance is reported below:

```
0170   52 df 47 2c 0c 86 de fe fe b9 f6 68 ae 23 4f a5  R.G,.......h.#O.
0180   81 53 79 43 fc fc 31 58 af ad 6e 30 29 f7 50 8a  .SyC..1X..n0).P.
0190   4a e1 78 43 30 6a 55 75 58 72 6e 4f 42 48 4c 42  J.xC0jUuXrnOBHLB
01a0   36 34 33 4a 51 38 69 42 52 37 39 46 59 49 79 71  643JQ8iBR79FYIyq
01b0   7a 62 38 48 4e 47 68 48 7a 52 59 6e 38 76 55 78  zb8HNGhHzRYn8vUx
01c0   41 4d 57 61 57 66 68 30 48 4c 30 61 76 73 61 6b  AMWaWfh0HL0avsak
01d0   7a 56 65 4d 32 42 76 64 64 43 64 41 45 75 75 53  zVeM2BvddCdAEuuS
01e0   4f 7a 41 4f 70 6b 30 37 4c 45 70 66 73 44 73 49  OzAOpk07LEpfsDsI
01f0   66 57 39 65 31 59 45 6e 43 38 52 62 76 57 65 50  fW9e1YEnC8RbvWeP
0200   59 63 54 77 7a 63 32 4f 50 4f 52 6b 71 4c 33 4b  YcTwzc2OPORkqL3K
0210   65 7a 69 62 72 57 4e 6d 58 33 4b 56 70 50 6c 45  ezibrWNmX3KVpPlE
```

This fragment contains a large majority of printable characters, thus one would expect that, since RPC over SMB/CIFS messages are mostly binary, a detection algorithm based on byte frequency distribution would be able to detect such payload. However, RPC over SMB/CIFS is also used to feed remote print spoolers with files to print. For example, in the filtered data set used for training we can identify the following fragment which is part of a PDF file sent to the spooler:

```
0200   09 2f 40 6f 70 53 74 61 63 6b 4c 65 76 65 6c 20  ./@opStackLevel
0210   40 6f 70 53 74 61 63 6b 4c 65 76 65 6c 20 31 20  @opStackLevel 1
0220   61 64 64 20 64 65 66 0d 0a 09 09 63 6f 75 6e 74  add def....count
0230   64 69 63 74 73 74 61 63 6b 20 31 20 73 75 62 0d  dictstack 1 sub.
0240   0a 09 09 40 64 69 63 74 53 74 61 63 6b 43 6f 75  ...@dictStackCou
0250   6e 74 42 79 4c 65 76 65 6c 20 65 78 63 68 20 40  ntByLevel exch @
0260   64 69 63 74 53 74 61 63 6b 4c 65 76 65 6c 20 65  dictStackLevel e
0270   78 63 68 20 70 75 74 0d 0a 09 09 2f 40 64 69 63  xch put..../@dic
0280   74 53 74 61 63 6b 4c 65 76 65 6c 20 40 64 69 63  tStackLevel @dic
0290   74 53 74 61 63 6b 4c 65 76 65 6c 20 31 20 61 64  tStackLevel 1 ad
```

Similar to the previous fragment, this fragment also contains a vast majority of printable characters. Due to the high variability of data, the threshold for detecting such fragment had to be set in such a way that many normal samples were classified as anomalous.

5.3 Tests with ICS Data Set

The results of this round of tests are summarized on Table 3. Anagram shows outstanding results in this test, and this does not come unexpected. The messages in this data set are rather short and the number of duplicates is also high (more than 95%). This is a perfect combination for Anagram and its binary-based approach. Anagram detects most of attack instances with a false positive rate that is lower than the adjusted false positive rate of 10 alerts per day. When detecting all of the attack instances, the false positive rate is still one order of magnitude lower than the adjusted false positive rate of 1 alert per minute.

McPAD also performs well with respect to the false positive. This is expected because the analysed Modbus traffic is expressed in messages of fixed length structure and with a limited range of values in used bytes. This results in McPAD accurately modeling relationships in the message structure.

PAYL seems to have a better packet-rate detection rate than POSEIDON. However, POSEIDON always performs better with respect to the instance-based detection rate as well as lower false positive. When the two algorithms are tuned to detect all of the attack instances, they both generate an overwhelming number of false positives, triggering on almost every packet.

With respect to the false positives generated during the verification phase, no raised alert turned out to be a true positive when processed with Snort. This is largely expected due to 1) the small number of available signatures for the Modbus protocol, and 2) the fact that the industrial control network from which we collected traffic from is highly isolated from other networks, with only a fixed number of hosts communicating. Thus, the chance of "noise" is substantially low.

Similarly to the previous test, there is no reason to test how a combination of algorithms would perform, because Anagram outperforms all the other algorithms in terms of detection and false positive rates.

Table 3. Test results on DS_{Modbus} and testing with AS_{Modbus}

	FPR (packet-based)	DR (packet-based)	DR (instance-based)
PAYL	0.00%	62.57%	101/163
	0.00%	92.63%	150/163
	9.25%	95.70%	155/163
	95.00%	100.00%	163/163
POSEIDON	0.04%	3.07%	4/163
	0.07%	77.30%	125/163
	0.37%	95.09%	154/163
	7.65%	97.54%	158/163
	97.81%	100.00%	163/163
Anagram	0.00%	3.68%	5/163
	0.00005%	10.42%	16/163
	0.00007%	18.40%	29/163
	0.00007%	96.93%	157/163
	0.002%	100.00%	163/163
McPAD	0.041	6.31%	10/163
	0.044	96.93%	157/163
	0.045	96.93%	157/163
	0.046	96.93%	157/163

Analysis of detected and undetected attacks. To understand why Anagram works so well with Modbus traffic consider the following two Modbus messages. The first one is a valid read request (identified by the 8th byte with value `0x03` which corresponds to the request "function code"):

```
35 ae 00 00 00 06 00 03 0c 7f 00 64
```

The following fragment is an attack instance attempting to corrupt the PLC memory by invoking a vulnerable diagnostic function (byte value `0x08`) with invalid data (bytes `0x00 0x04 0x00 0x00`):

```
00 00 00 00 00 06 0a 08 00 04 00 00
```

We first observe that the anomalous value in this payload are the byte value of the function code, and the subsequent four bytes (never observed in the training set on that same positions). There are 6 3-grams over 10 (60%) which are not present in the valid request. The number of distinct 3-grams observed during training is not much bigger than the one observable in the aforementioned request, due to the large number of duplicated payloads. Thus, with such a small packet size, even a few bytes with unusual value can make a big difference.

On the other hand, from the results in Table 3 we see that for all the algorithms there is always a significant increase in the amount of false positives raised when the threshold is adjusted to detect all attack instances. We observe that the attack instances that do not get detected before the threshold is adjusted are similar to the one in the following example:

```
00 00 00 00 00 02 0a 11
```

This 8 bytes long message is the smallest possible Modbus message allowed by the protocol specification. We acknowledge that this request is not unusual only because of its size, but also because the function code value `0x02` (corresponding to the request "report slave ID") was never observed during training. We have verified that the only 3-gram in this payload not observed during the training is the last one `0x02 0x0a 0x11`. Thus, in spite of the small size of this payload, the threshold has to be lowered in order to detect it. Detecting such anomalous packet (with only one anomalous n-gram) in a bigger message would be much more difficult.

6 Related Work

To the best of our knowledge, Ingham and Inoue describe the most recent framework for testing the performance of IDS algorithms [18]. The authors focus on HTTP traffic. The framework is based on the general principle that testing different IDS algorithms on the same network environment and with the same network and attack data allows a better comparison of the algorithms' performance, which would be impossible by re-using the results of the unrelated, individual tests run by the algorithm developers. They collect background web traffic from four different websites and create a publicly available set of network traces. The traces contain instances of web attacks generated by running exploitation tools downloaded from popular vulnerability repositories (e.g., BugTraq, SecurityFocus and the Open Source Vulnerability Database). There are other IDS evaluation frameworks, as reported in [18], but are all quite dated.

In [31] Song et al. show that polymorphic behaviour in shellcodes is too greatly spread to be modeled effectively. The authors note that it is difficult to model data with high variability, especially in case the adversary is able to inject some "normal" looking n-grams into the attack payload to make it look legitimate. Our experiments with SMB traffic confirm that, not only this observation is true for polymorphic shellcode traffic, but even for regular attacks that do not leverage any evasion techniques.

7 Conclusion

In this paper we present a thorough analysis of several n-gram-based algorithms for network-based anomaly detection. We investigate the performance of state-of-the-art detection algorithms when analyzing network traffic from two binary protocols. We believe our analysis allows us to draw interesting observations and conclusions.

First, despite the fact that the attack instances on the SMB/CIFS protocols are correctly detected, all studied algorithms incur in a high penalty in terms of false positives they raise. Concretely, it would be expensive to deploy them independently in a real environment. On the other hand, if we restrict the field to the Modbus protocol alone, Anagram detects almost every attack instance with a rate of false positives lower than the 10 alerts per day threshold.

We believe that with such performance the algorithm could be deployed in a real environment.

Second, all studied algorithms trigger on the exploitation payload. We can observe this by selecting two different attack instances that exploit the same vulnerability, but using two different attack payloads. In several cases, while one instance is detected even with a low threshold, to detect both attacks one needs to increase the threshold significantly. The previously missed attack instance usually contains a small-size attack payload, and thus "blends" more easily with the normal payload data, thereby avoiding detection.

Third, there is no absolute best algorithm among the ones we studied. Anagram performs slightly better than the rest when analyzing the filtered SMB/CIFS and the Modbus protocols, but it is also the one performing worst when the filter is not applied. Technically, this is due to the fact that the unfiltered SMB/CIFS traffic contains several n-grams that are also present in the attack payloads. This supports the intuition that variability of the network traffic has a great impact on the performance of these systems. Indeed, *every* studied algorithm is affected by this, allowing us to conclude that, rather than the single implementation, it is the underlying principle of capturing regularity in the unstructured packet payload that does not hold true. Our results show that n-gram analysis quickly becomes incapable of capturing relevant content features when analysing moderately variable traffic. This problem could be partly alleviated by deploying the detection system in combination with some other sensor that will verify the correctness of alerts [35]. We believe that a more promising approach is the one focusing on identifying chunks of payload (that represent some kind of semantic units) and applying the n-gram analysis on those. For example, several authors propose to exploit the syntactical knowledge of the HTTP protocol to improve the overall performance of anomaly-based systems, e.g. in [30]. We foresee that a similar approach could be applied to binary protocols as well. Another issue that remains still open is how to "measure" traffic variability without having to run several empirical experiments.

Acknowledgements. We thank Davide Ariu for providing the McPAD source code and supporting during tests. The research leading to these results has been supported by the Ministry of Security and Justice of the Kingdom of the Netherlands through projects Hermes, Castor and Midas and by the European Commission through project FP7-SEC-285477-CRISALIS funded by the 7th Framework Program.

References

1. Ariu, D., Tronci, R., Giacinto, G.: HMMPayl: An intrusion detection system based on Hidden Markov Models. Computers and Security 30(4), 221–241 (2011)
2. Athanasiades, N., Abler, R., Levine, J., Owen, H., Riley, G.: Intrusion Detection Testing and Benchmarking Methodologies. In: IWIA 2003: Proc. 1st IEEE International Workshop on Information Assurance, pp. 63–72. IEEE Computer Society Press (2003)

3. Auriemma, L.: Advisories (March 2011), http://aluigi.altervista.org/ (accessed March 2012)
4. Axelsson, S.: The base-rate fallacy and the difficulty of intrusion detection. ACM Transactions on Information and System Security 3(3), 186–205 (2000)
5. Bloom, B.H.: Space/time trade-offs in hash coding with allowable errors. Communications of the ACM 13(7), 422–426 (1970)
6. Bolzoni, D., Zambon, E., Etalle, S., Hartel, P.H.: POSEIDON: a 2-tier Anomaly-based Network Intrusion Detection System. In: IWIA 2006: Proc. 4th IEEE International Workshop on Information Assurance, pp. 144–156. IEEE Computer Society Press (2006)
7. Microsoft Security Response Center. Microsoft Security Bulletin, http://technet.microsoft.com/en-us/security/bulletin/ (accessed March 2012)
8. Microsoft Security Response Center. Conficker Worm: Help Protect Windows from Conficker (April 2009), http://technet.microsoft.com/en-us/security/dd452420.aspx (accessed March 2012)
9. Cui, A., Stolfo, S.J.: Defending Embedded Systems with Software Symbiotes. In: Sommer, R., Balzarotti, D., Maier, G. (eds.) RAID 2011. LNCS, vol. 6961, pp. 358–377. Springer, Heidelberg (2011)
10. Damashek, M.: Gauging similarity with n-grams: Language-independent categorization of text. Science 267(5199), 843–848 (1995)
11. Digital Bond, Inc. QuickDraw SCADA IDS, http://www.digitalbond.com/tools/quickdraw/ (accessed March 2012)
12. Dussel, P., Gehl, C., Laskov, P., Busser, J., Störmann, C., Kästner, J.: Cyber-Critical Infrastructure Protection Using Real-Time Payload-Based Anomaly Detection. In: Rome, E., Bloomfield, R. (eds.) CRITIS 2009. LNCS, vol. 6027, pp. 85–97. Springer, Heidelberg (2010)
13. Mu Dynamics. pcapr, http://pcapr.net (accessed March 2012)
14. Falliere, N., Murchu, L.O., Chien, E.: W32.Stuxnet Dossier. Technical report, Symantec (September 2010)
15. Fogla, P., Sharif, M., Perdisci, R., Kolesnikov, O., Lee, W.: Polymorphic blending attacks. In: Proc. 15th USENIX Security Symposium, pp. 241–256. USENIX Association (2006)
16. Forrest, S., Hofmeyr, S.A.: A Sense of Self for Unix Processes. In: S&P 1996: Proc. 17th IEEE Symposium on Security and Privacy, pp. 120–128. IEEE Computer Society Press (2002)
17. Gu, G., Porras, P., Yegneswaran, V., Fong, M., Lee, W.: BotHunter: Detecting Malware Infection Through IDS-Driven Dialog Correlation. In: Proc. 16th USENIX Security Symposium (Security 2007). USENIX Association (2007)
18. Ingham, K.L., Inoue, H.: Comparing Anomaly Detection Techniques for HTTP. In: Kruegel, C., Lippmann, R., Clark, A. (eds.) RAID 2007. LNCS, vol. 4637, pp. 42–62. Springer, Heidelberg (2007)
19. Kohonen, T.: Self-Organizing Maps, Second Extended Edition. Springer Series in Information Sciences, vol. 30. Springer (1995)
20. MSDN Library. [MS-CIFS]: Common Internet File System (CIFS) Protocol Specification, http://msdn.microsoft.com/en-us/library/ee442092v=prot.13.aspx (accessed March 2012)
21. Lippmann, R.P., Haines, J.W., Fried, D.J., Korba, J., Das, K.: The 1999 DARPA off-line intrusion detection evaluation. Computer Networks: The International Journal of Computer and Telecommunications Networking 34(4), 579–595 (2000)

22. Loscocco, P.A., Smalley, S.D., Muckelbauer, P.A., Taylor, R.C., Turner, S.J., Farrell, J.F.: The Inevitability of Failure: The Flawed Assumption of Security in Modern Computing Environments. In: NISSC 1998: Proc. 21st National Information Systems Security Conference, pp. 303–314 (1998)
23. Mahoney, M.V., Chan, P.K.: An Analysis of the 1999 DARPA/Lincoln Laboratory Evaluation Data for Network Anomaly Detection. In: Vigna, G., Kruegel, C., Jonsson, E. (eds.) RAID 2003. LNCS, vol. 2820, pp. 220–237. Springer, Heidelberg (2003)
24. Metasploit Penetration Testing Software, http://metasploit.com/ (accessed March 2012)
25. Mirkovic, J., Reiher, P.: A taxonomy of DDoS attack and DDoS defense mechanisms. SIGCOMM Comput. Commun. Rev. 34, 39–53 (2004)
26. Nakayama, T.: W32.Sasser.Worm. Technical report, Symantec (April 2004)
27. NIST: National Institute of Standards and Technologies. National Vulnerability Database, http://nvd.nist.gov (accessed March 2012)
28. Perdisci, R., Ariu, D., Fogla, P., Giacinto, G., Lee, W.: McPAD: A multiple classifier system for accurate payload-based anomaly detection. Computer Networks 53(6), 864–881 (2009)
29. Sommer, R., Paxson, V.: Outside the Closed World: On Using Machine Learning for Network Intrusion Detection. In: S&P 2010: Proc. 31st IEEE Symposium on Security and Privacy, pp. 305–316. IEEE Computer Society (2010)
30. Song, Y., Stolfo, S.J., Keromytis, A.D.: Spectrogram: A Mixture-of-Markov-Chains Model for Anomaly Detection in Web Traffic. In: NDSS 2009: Proc. 16th ISOC Symposium on Network and Distributed Systems Security. The Internet Society (2009)
31. Song, Y., Locasto, M.E., Stavrou, A., Keromytis, A.D., Stolfo, S.J.: On the infeasibility of modeling polymorphic shellcode. In: Proceedings of the 14th ACM Conference on Computer and Communications Security, CCS 2007, pp. 541–551. ACM, New York (2007)
32. Swales, A.: Open MODBUS/TCP Specification (March 1999)
33. The OWASP Foundation. OWASP: The Open Source Web Application Security Project, https://www.owasp.org (accessed March 2012)
34. Vapnik, V.N., Lerner, A.: Pattern recognition using generalized portrait method. Automation and Remote Control 24 (1963)
35. Wang, K., Parekh, J.J., Stolfo, S.J.: Anagram: A Content Anomaly Detector Resistant to Mimicry Attack. In: Zamboni, D., Kruegel, C. (eds.) RAID 2006. LNCS, vol. 4219, pp. 226–248. Springer, Heidelberg (2006)
36. Wang, K., Stolfo, S.J.: Anomalous Payload-Based Network Intrusion Detection. In: Jonsson, E., Valdes, A., Almgren, M. (eds.) RAID 2004. LNCS, vol. 3224, pp. 203–222. Springer, Heidelberg (2004)

Online Social Networks, a Criminals Multipurpose Toolbox (Poster Abstract)

Shah Mahmood and Yvo Desmedt

Department of Computer Science, University College London, United Kingdom
{shah.mahmood,y.desmedt}@cs.ucl.ac.uk

In this paper, we discuss how online social networks can be used by conventional (physical) criminals as a toolbox to (1) find incentives for a crime (who to rob and why?), (2) plan the crime's execution (how and when to rob them?), and (3) make an escape plan (how to avoid getting caught?).

Text Updates and Photo Uploads. Textual updates similar to the ones shown in Table 1 can provide criminals with incentives. 154000 users on Facebook have publicly expressed the possession of a diamond while 2190 expressed losing their keys at a certain place. Any overlap between the two sets, within the geographical reach of the criminal, could give then an easy access to a valuable item. Table 1 contains limited publicly shared results. A criminal may widen their search by embedding support for their local languages and using a larger set of phrases. Similarly, uploading a photo of valuable items on social networks can also incentivize crime. Moreover, sleeping and drinking patterns identified from uploads, can help criminals in planning their crime execution hours.

Crime Promotion Using YouTube. YouTube can be used by criminals for planning their crime and to gain the necessary skills to execute it, e.g. a criminal may learn how to pick locks from one of the 23300 videos teaching it on YouTube (statistics given in Table 2).

LinkedIn Helps Find Professionals. LinkedIn can be used by criminal groups to locate people with specific skills, e.g. LinkedIn's "CodeBreakers" group can help with finding cryptologists. Similarly they may use LinkedIn's "search like a pro" feature to short list candidates for specific tasks. Coworker information by LinkedIn could be used to coerce users to do a task.

Table 1. Theft incentivizing phrases on Facebook and Twitter (May 16, 2012)

Phrase	Shared publicly on Facebook	Shard publicly on Twitter
"I bought" OR "I got" "a diamond"	154,000	12,900
"I got a diamond necklace"	24	18
"I bought" OR "I got" "Rolex"	23,000	2,140
"I lost my keys in"	2,190	1,850

Facebook Applications as Information Goldmines. Facebook applications are mostly owned and regulated by third parties. All top ten Facebook

D. Balzarotti, S. Stolfo, and M. Cova (Eds.): RAID 2012, LNCS 7462, pp. 374–375, 2012.

applications using statistics from appdata.com, record a user's: user name, profile picture, gender, list of friends, email address, userID and other information publicly shared. Most of these applications also access the user's home address and contact numbers. All this information, if leaked to criminals, can be used by them for targeted crimes, e.g. a user's name, profile picture and gender can provide criminals with an estimate of the level of resistance they may face, if they attempt to break into a house and find the person being there.

Social Network Events May Define Optimal Times for Theft. Facebook and Meetup events can be used to invite users to events. For public events anyone can confirm, decline, comment on event page, or check who else has been invited. Criminals can also check the event pages to see if their potential victims are attending and plan their crime accordingly. On Facebook any user's attendance option for a public event page is publicly visible.

Impact of Family and Friends on Crime. Having friends and family working for law enforcement agencies can act as a deterrent or an attractor for criminals. Petty thieves will prefer to target someone with limited access to means of tracing the criminal. Large criminal organizations may want to take revenge to set an example. Mexican drug cartels have killed several U.S. federal agents as an act of revenge. Moreover, Information about family members could help criminals kidnap them for ransom or to coerce their main target to fulfill a task.

Friends as Your Weak Link. Social network friends can provide criminals with access to victims photos, postings, etc., helping them with their attack.

Facebook Real Name Policy can Help Theft. With an estimate of the age of the victim and the full name, a criminal can use websites like peopletracer.co.uk or 123people.com to find information including full address, property value, phone number and neighbors details. All these bits and pieces of information can make a previously impossible crime fairly easy.

Organize Crime and Target Competitors Using Social Networks. Criminals like any other group of users of social networks can plan and organize their events through social networks, e.g. Twitter was used in London riots to organize riots and loots. With billions of photos and videos being shared, some can be used by criminals for passing steganographic messages about their next project, avoiding eavesdropping. Criminals can run social network analysis to find their competitors and leak their information to the law enforcement agencies.

Future Work. We will observe some users who upload crime incentivizing information to identify if their risk of being robbed increases. Also, as a defense tool we will write a tool which will warn users when they upload crime incentivizing information.

Table 2. YouTube videos that may help theft (May 16, 2012)

Phrase	Total Videos	Maximum views (single video)
pick lock	23300	7788708
RFID cloning	237	48199
hack CCTV	733	51910

The Triple-Channel Model: Toward Robust and Efficient Advanced Botnets (Poster Abstract)

Cui Xiang[1], Shi Jinqiao[2], Liao Peng[1], and Liu Chaoge[1]

[1] Institute of Computing Technology, Chinese Academy of Sciences
[2] Institute of Information Engineering, Chinese Academy of Sciences
cuixiang@ict.ac.cn

Abstract. Botnet robustness and efficiency are two contradictory features from a general point of view. To achieve them simultaneously, we design a command and control (C&C) channel division scheme and then propose a *Botnet Triple-channel Model (BTM).* BTM divides a C&C channel into three independent sub-channels, denoting as *Command Download Channel*, *Registration Channel* and *Data Upload Channel,* respectively. Botnets based on BTM will promise to be as robust as P2P botnets and as efficient as centralized botnets.

Keywords: botnet, Triple-Channel, BTM, C&C.

1 Background and Problem Analysis

Most of current botnets could not achieve robustness and efficiency at the same time. For example, Rustock, Mariposa, Coreflood, and Waledac/Kelihos have paid much attention to efficiency, while being shut down due to C&C protocol vulnerabilities. On the other hand, Conficker, which constructs an extremely robust C&C channel, is ineffective in the aspect of monitoring the botnet and retrieving the collected data.

The internal cause of the contradictory between robustness and efficiency probably lies in the fact that current botnets always rely on only **one** C&C protocol to accomplish **all** tasks, however, it is impossible for any existing C&C protocol to satisfy all requirements solely. Fox example, the robust P2P/URL Flux [1] protocols have no upstream channels; the recoverable Domain-flux protocol is limited by performance and vulnerable to sinkhole attack. In a word, each C&C protocol has its particular advantages as well as corresponding limitations. BTM aims at solving the problem to some degree.

2 The Proposed Botnet Triple-Channel Model

BTM (shown in Fig.1) includes three independent C&C sub-channels. Each sub-channel, determined by its characteristic, is responsible for particular task.

Command Download Channel (CDC). CDC is only responsible for commands distribution. CDC must be extremely robust, recoverable and high-performance to defend against worldwide coordinated countermeasures; while the uploading capability is not indispensable. Therefore, URL Flux/P2P style protocols are suitable; although they are unidirectional (data can and can only transfer from servers to bots).

D. Balzarotti, S. Stolfo, and M. Cova (Eds.): RAID 2012, LNCS 7462, pp. 376–377, 2012.

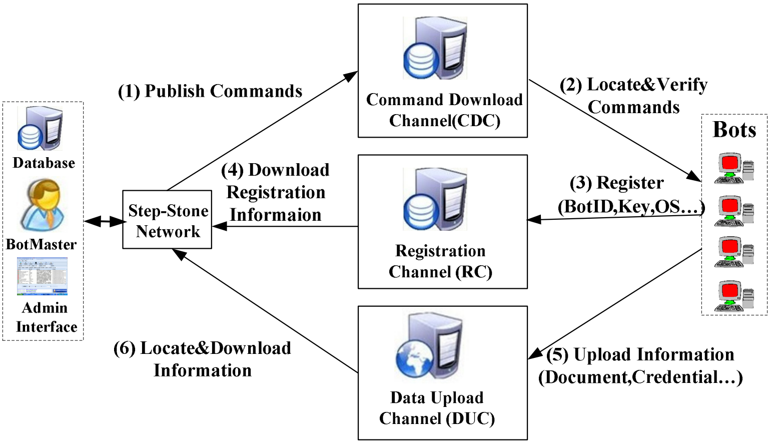

Database
BotMaster
Admin Interface
Step-Stone Network
(1) Publish Commands
Command Download Channel(CDC)
(2) Locate&Verify Commands
Bots
(3) Register (BotID,Key,OS...)
(4) Download Registration Informaion
Registration Channel (RC)
(5) Upload Information (Document,Credential...)
Data Upload Channel (DUC)
(6) Locate&Download Information

Fig. 1. Botnet Triple-Channel Model

Registration Channel (RC). RC is only responsible for fundamental information (i.e., the automatically generated BotID and individualized symmetric key, IP and OS etc.) collection. The information will be used for monitoring botnet size, encrypting data, etc. RC must be bidirectional, encrypted and recoverable to enable uploading activities and defend against monitoring, sinkhole and host-level forensics; while the robustness and high-performance features are not indispensable. Therefore, Domain-flux style protocol is suitable, although it is vulnerable to DDoS and sinkhole attack.

Data Upload Channel (DUC). DUC is only responsible for collecting and storing the uploaded information. DUC requires high-performance and mass storage space to enable massive data uploading in parallel by large-scale botnets. Furthermore, it must ensure the uploaded information can and can only be located and decrypted by botmasters who own bots' BotIDs and individualized symmetric keys. However, DUC is not necessary to be recoverable. Thus, those free cloud-based file hosting services [2] could be exploited. To go a step further, we can combine cloud and URL Shortening Services supporting custom alias (i.e., tinyurl.com) together to enable botmaster to locate the uploaded file automatically, since bots could make the file downloading URLs computable and predictable by constructing custom shorten URLs which end with BotID and current date (i.e., http://tinyurl.com/BotID_20120605_001).

In future works, we will prove the completeness and independence of the triple sub-channels and invests more research on how to fight against BTM-based botnets.

References

1. Xiang, C., Binxing, F., Lihua, Y., Xiaoyi, L.: Andbot: Towards Advanced Mobile Botnets. In: Proceedings of the 4th USENIX Workshop on Large-scale Exploits and Emergent Threats, LEET 2011 (2011)
2. Paz, R.D.: Malware Uses Sendspace to Store Stolen Documents (February 2012), `http://tinyurl.com/use-Cloud-but-no-ShortenURL`

Network Security Analysis Method Taking into Account the Usage Information (Poster Abstract)

Wu Jinyu[1,2], Yin Lihua[2], and Fang Binxing[1]

[1] Beijing University of Posts and Telecommunications, Beijing, China
[2] Institute of Information Engineering, Chinese Academy of Sciences, Beijing, China
eyoudian19@gmail.com, yinlihua@iie.ac.cn, fangbx@bupt.edu.cn

Abstract. Existing network security analysis methods such as using tools like attack graphs or attack trees to compute risk probabilities did not consider the concrete running environment of the target network, which may make the obtained results deviate from the true situation. In this paper, we propose a network security analysis method taking into account the usage information of the target network. We design usage sensors in each host to get the usage information in the network. Combining with attack graph generation tool which gets all the vulnerabilities in the network in the graph form, we evaluation the network using the usage information and the vulnerabilities information, and get more accurate evaluation results.

1 Introduction

Network security analysis is the vital step in risk management. Many Models, such as attack graphs, attack tree and Petri Net, have been proposed. However, existing methods [1], [2] based on these models only consider the vulnerabilities in the network, while the running environment information also matters. For example, let's consider two running circumstances in the network shown in Fig. 1.

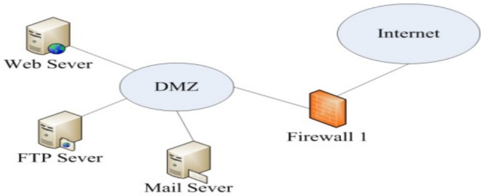


Fig. 1. An example network

(1) The Web Sever in the network offers the major service and has been heavily used, while the FTP Server and Mail Server are rarely used; (2) The FTP Server in the network offers the major service and has been heavily used, while the Web Server and Mail Server are rarely used. The vulnerabilities are the same in both of the two circumstances. We can easily judge that the Web Server in the Circumstance 1 is more risky than the Web Server in the Circumstance 2 and the FTP Server in the Circumstance 2 is more risky than the FTP Server in the Circumstance 1.

D. Balzarotti, S. Stolfo, and M. Cova (Eds.): RAID 2012, LNCS 7462, pp. 378–379, 2012.

Thus, to make the evaluation result more accurate, we have to consider the running environment of the network.

2 The Analysis Method

First, we use scan tool such as Nessus to get all the vulnerabilities in the network. And then, we use the off-the-shell attack graph generation tool such as the MulVAL attack graph toolkit [3] to generate the attack graph. To get the usage information in the network, we design usage sensors and place them into the hosts in the network. The usage sensors will count the traffic to the host in the setting time intervals.

There are two kinds of nodes in the attack graph: the condition node and the exploit node. We calculate the risk probabilities of them using different computing methods. Let the variable *Total* be the sum of traffic amount counting in all hosts in the network.

(1) For each of the condition nodes c that has no income edge in the attack graph, $\mathrm{P[c]} = \frac{\mathrm{F[c]}}{\mathrm{Total}}$. Where P[c] represents the risk probability of c, and F[c] is the counting traffic amount of c.

(2) For each of the exploit nodes e in the attack graph, $\mathrm{P[e]} = \frac{\mathrm{F[e]}}{\mathrm{Total}} * (1 - \mathrm{D[e]}) * \prod_{\mathrm{c \in pre[e]}} \mathrm{P[c]}$. Where P[e] represents the risk probability of e, F[e] is the counting traffic amount of e, D[e] is the difficulty factor of the exploit node computed by the Common Vulnerability Scoring System (CVSS for short, provided by the National Vulnerability Database), and pre[e] is the pre-condition node set of e.

(3) For each of condition nodes c that has income edges in the attack graph, $\mathrm{P[c]} = \frac{\mathrm{F[c]}}{\mathrm{Total}} * (1 - \prod_{\mathrm{e \in pre[c]}}(1 - \mathrm{P[e]}))$. Where P[c] represents the risk probability of c, F[c] is the counting traffic amount of c, and pre[c] is the pre-condition node set of c.

We calculate the risk probabilities of the nodes in the attack graph in topological order. And to avoid zero multiplication, we replace each zero result with a small positive value in the computing process.

3 Preliminary Results and Future Works

We have implemented our method to evaluation a small network. And experiment results show that our method get more accurate result due to considering the running environment of the target network. In the future work, we will develop more efficient method for real time security evaluation.

References

1. Jha, S., Sheyner, O., Wing, J.: Two formal analyses of attack graphs. In: Proc. of the 15th IEEE Computer Security Foundations Workshop, pp. 49–63. IEEE Computer Society, Cape Breton (2002)
2. Ye, Y., Xu, X.-S., Jia, Y., Qi, Z.-C.: An Attack Graph-Based Probabilistic Computing Approach of Network Security. Chinese Journal of Computers 33(10), 1987–1996 (2010)
3. Ou, X.M., Boyer, W.F., McQueen, M.A.: A scalable approach to attack graph generation. In: Proc. of the 13th ACM Conf. on Computer and Communications Security, pp. 336–345. ACM Press, Alexandria (2006)

Automatic Covert Channel Detection in Asbestos System (Poster Abstract)

Shuyuan Jin[1], Zhi Yang[2], and Xiang Cui[1]

[1]Institute of Information Engineering, Chinese Academy of Sciences
[2] The PLA Information Engineering University
jinshuyuan@ict.ac.cn

Abstract. The detection of covert channels in an information flow model is a practical problem in determining whether the security guarantees of the operation system have been achieved. Asbestos system is a typical information confidentiality protection system. This poster introduces a formal approach to automatically detect covert channels in Asbestos systems. The approach innovatively generalizes a CSP (Communicating Sequential Process) based formal description of Asbestos system and utilizes Ray's noninterference Equivalence in the detection of covert channels. The covert channels are automatically detected by employing a CSP based model checking tool FDR2.

Keywords: Covert Channel, CSP, Noninterference.

1 Introduction

The Information Flow Control models (IFCs) usually refer to such class of computer security models, which require access of all subjects and objects under their control on a system wide basis. Recently developed Asbestos system [1] extends information confidentiality models (classical IFCs), resulting in its real application in practice.

However, as Asbestos system becomes more flexible, it gets more susceptible to covert channel attacks than classical IFCs. Few works discuss automatic detection of covert channels in Asbestos system. This poster contributes the field a formal model in CSP notions and realizes automatic covert channel detection in Asbestos system.

2 Formal Detection Model

The formal detection model is defined as follows:

$$\textit{Detection Scheme} =< \mathbf{CR}, \mathbf{GIFS}, \mathbf{ASSERT} >$$

Where

CR represents the Control Rules of Asbestos system.

GIFS represents the Generalized Information Flow System, More precisely, $GIFS =< S, O, CONSTRAIN, IPC, IO, STATE >$, where S is the set of subjects, O is the set of objects, *CONSTRAIN* is the control engine which controls the information flow in the system according to **CR**, *IPC* is the set of events among

D. Balzarotti, S. Stolfo, and M. Cova (Eds.): RAID 2012, LNCS 7462, pp. 380–381, 2012.

subjects. For each *ipc* in $IPC, ipc \in \{send, recv\}$. *IO* is the set of events among subjects and objects. For each *io* in *IO*, $io \in \{create, created, delete, deleted, read, isread, write, written, execute, executed\}$. *STATE* is the internal channels that *CONSTRAIN* can use to acquire or modify the labels of subjects and objects in Asbestos system.

ASSERT is an assertion of "the Asbestos system has no covert channels".

We use CSP notions to represent the *Detection Scheme DS* as follows:

$$DS = (OBJECTS \underset{io,ipc,born,exit}{\|} CONSTRAIN) \setminus (aCONSTRAIN \bigcup aOBJECT)$$

where $aCONSTRAIN = \{state, random, lock, unlock, pid, cr\}$

$$aOBJECT = \{io_{i,j,L}.command \mid i \in P, j \in Q, command \in \{created, deleted, isread, written\}\}$$

That is, the *Detection Scheme* is the synchronization of the *OBJECTS* (representing all activities of subjects and objects) and the *CONSTRAIN* (representing information flow controls), with internal activities $(aCONSTRAIN \bigcup aOBJECT)$ hidden.

The **ASSERT** in the detection schema is based on Ryan's noninterference Equivalence [2] in CSP. Ryan's noninterference Equivalence states that the set of failures of a system with all HIGH events shielded is equivalent to the set of failures of this system preventing all HIGH events. That is: $(DS \underset{aHIGH}{\|} STOP) \setminus aHIGH \equiv_{failures} DS \setminus aHIGH$ where *aHIGH* represents all high level events or all security sensitive events. That $(DS \underset{aHIGH}{\|} STOP)$ represents putting STOP in parallel with *DS* synchronized over all *HIGH* events (*HIGH* events can be regarded as high level events or high sensitive events), which means shielding all high level events. $DS \setminus aHIGH$ means preventing all high level events. If Asbestos system satisfies Ryan's noninterference Equivalence, then Asbestos system is noninterference, and we can assert that Asbestos system has no covert channels. Otherwise, if Ryan's noninterference Equivalence can not be verified, we will assert that Asbestos system has convert channels. We utilize a CSP model checker FDR2 in automatic detection.

3 Detection

The detection of covert channels in Asbestos system is an automatically formal verification process. The **CR** in an Asbestos system can be represented as follows:

$CR = cr.requery?(flow, sender_s, sender_{r,} receive_{s,} receive_r) \rightarrow if\ \ sender_s \le receive_r$

then $cr.res!(true, \max(sender_s, receive_s)) \rightarrow CR\ else\ cr.res!(false, receive_s) \rightarrow CR$

FDR2 spent 42 seconds to complete a verification example. It gives out a counter example, which shows Asbestos has covert channels.

References

1. Efstathopoulos, P., Krohn, M., VanDeBogart, S., Frey, C., Ziegler, D., Kohler, E., Mazieres, D., Kaashoek, F., Robert, M.: Labels and event processes in the asbestos operating system. In: Proc. of the 20th ACM Symposium on Operating Systems Principles (2005)
2. Ryan, P.A., Schneider, S.A.: Process algebra and noninterference. Journal of Computer Security 9(1-2), 75–103 (2001)

EFA for Efficient Regular Expression Matching in NIDS[*] (Poster Abstract)

Dengke Qiao[1,3], Tingwen Liu[2,3], Yong Sun[1], and Li Guo[1]

[1] Institute of Information Engineering, Chinese Academy of Sciences, Beijing, China
[2] Institute of Computing Technology, Chinese Academy of Sciences, Beijing, China
[3] Graduate University of Chinese Academy of Sciences, Beijing, China
{qiaodengke,liutingwen}@nelmail.iie.ac.cn, sunyong@iie.ac.cn

Abstract. Regular Expression (RegEx) matching has been widely used in many network security systems. Despite much effort on this important problem, it remains a fundamentally difficult problem. DFA-based solutions are efficient in time but inefficient in memory, while NFA-based solutions are memory-efficient but time-inefficient. This poster provides a new solution named EFA (Excl-deterministic Finite Automata) to address the problem by excluding cancerogenic states from active state sets. The cancerogenic states are identified based on conflict relations. We make an evaluation of EFA with real RegExes and traffic traces. Experimental results show that EFA can dramatically reduce DFA state size at the cost of limited matching performance.

1 Introduction and Our Work

How to perform RegEx matching at line rate is a crucial issue in a variety of network security applications, such as network intrusion detection systems (NIDS). Comparing with NFA, DFA is the preferred representation of RegExes in gigabit backbones because of its high matching efficiency. DFA, which is full-deterministic from NFA, may experience state explosion during the process of transformation that creates a DFA state for each possible *A*ctive *S*tate *Set* of NFA (ASSet). We find that some states in a NFA are the key reason that leads to state explosion, we call them cancerogenic states.

Y.H.E. Yang et al. [1] define a string set for each NFA state, and then they define four relationships between states based on the corresponding string sets. They further prove that a NFA without conflict relations between its any two states will not generate an explosive DFA. However, their method to calculate the relations by comparing string sets is inefficient and inaccurate.

This poster first introduces an accurate method named ROBAS to calculate relations between states based on ASSets but not string sets. We prove that the relation between states x and y can be exactly calculated by the following three

[*] Supported by the National High-Tech Research and Development Plan of China under Grant No. 2011AA010705 and the Strategic Priority Research Program of the Chinese Academy of Sciences under Grant No.XDA06030200.

D. Balzarotti, S. Stolfo, and M. Cova (Eds.): RAID 2012, LNCS 7462, pp. 382–383, 2012.

Table 1. Comparison among NFA, DFA and EFA on state size and construction time

RegEx	NFA		DFA		EFA $(k = 3)$		EFA $(k = 15)$	
set	states	times(s)	states	times(s)	states	times(s)	states	times(s)
snort24	749	0.08	13886	178.50	2394	26.28	592	16
bro217	2645	0.18	6533	260.63	3300	34.39	2659	35.33

Table 2. Comparison among NFA, DFA and EFA on the size of ASSet during matching

RegEx	NFA		DFA		EFA $(k = 3)$		EFA $(k = 15)$	
set	max	average	max	average	max	average	max	average
snort24	10	1.144	1	1	1	1	1	1
bro217	39	9.33	1	1	2	1.004	3	1.056

conditions: $x \cap y$, $\overline{x} \cap y$ and $x \cap \overline{y}$. The condition $x \cap y$ implies x and y are both active in some ASSets. States x and y are conflict if the three conditions are all true. Based on the insight, we can get that the first condition for any two states in the same ASSet is true, and the second condition for the case that one state in an ASSet and the other state not in the ASSet is true. Thus, for a given NFA, we can get its all relations exactly after scanning its ASSets in one time.

For a given NFA, transforming it into a DFA can obtain all ASSets. The DFA transition matrix is needless, thus abandoning it can largely reduce memory consumption. Even so, the modified transformation is still slow. To accelerate the process, this poster proposes an algorithm named ASS-SUBSET. As an ASSet (a DFA state) has many common transitions, it is wasteful to obtain all ASSets on 256 characters. In ASS-SUBSET, each ASSet only transfers on these necessary characters got by combining the selected characters of NFA states in the ASSet.

We regards these NFA states that have many conflict relations as cancerogenic states. Specifically, we introduce a threshold k, and exclude the k states that have the most conflict relations from ASSets. Then we can transform NFA to EFA similarly as the transformation from NFA to DFA.

2 Evaluation

We make use of real RegExes from Snort system and Bro system, and real traffic traces captured from backbones to evaluate our work. As shown in Table 1 and Table 2, our experimental results show that EFA can reduce states by several to dozens of times while at the cost of one percent loss in matching speed comparing with DFA. Moreover, the time used to construct EFA from RegExes is much less than that used to construct DFA. This implies that EFA is more suitable to perform large-scale RegEx matching in high-speed networks.

Reference

1. Yang, Y.H.E., Prasanna, V.K.: Space-Time Tradeoff in Regular Expression Matching with Semi-Deterministic Finite Automata. In: INFOCOM, pp. 1853–1861 (2011)

Distress Detection (Poster Abstract)

Mark Vella, Sotirios Terzis, and Marc Roper

Department of Computer and Information Sciences, University of Strathclyde,
Livingstone Tower, 26 Richmond Street, G1 1XH Glasgow, United Kingdom
{mark.vella,sotirios.terzis,marc.roper}@strath.ac.uk

Web attacks are a major security concern as novel attacks can be easily created by exploiting different vulnerabilities, using different attack payloads, and/or encodings (obfuscation). Intrusion detection systems (IDS) aim to correctly detect attacks. There are two main approaches to intrusion detection: misuse and anomaly detection. Despite the difference in approach, they both fail to offer adequate resilience to novel attacks due to the difficulty in generalizing beyond known attack or normal behavior [1].

Distress Detection. The aim of distress detection (DD) is to address this problem and to provide resilience to novel attacks by generalizing beyond known attacks while controlling the false positives (FP) rate. In order to achieve this DD combines attack generalization based on attacker objectives, dynamic analysis techniques for the definition of suspicious behavior signatures, and feature-based correlation of suspicious HTTP requests and system events.

Attacker objectives are the intended end results of attacks. Whilst attackers can produce a large number of attacks to achieve their objective, there are certain system events that are necessary for each particular objective [2]. By focusing on the objectives, DD can generalize beyond specific instances of known attacks to ones that have the same objective.

Within a specific objective, HTTP requests are suspicious if they can potentially launch an attack with this objective, while attack symptoms are those system events that are produced when the attack succeeds. Both suspect HTTP requests and attack symptoms are identified by signatures that are resistant to obfuscation and aim to maximize detection effectiveness. In contrast to misuse detection, they do not have to be exclusive to attacks.

The responsibility of associating attack symptoms with suspect HTTP requests in a manner that suppresses false alerts is placed on a separate alert correlation process. Our premise is that attacks must be launched by suspicious HTTP requests and their successful execution must generate symptoms with similar features. We do not relate alerts raised against specific attack techniques but rather ones that indicate generic suspicious behavior.

Example Distress Detector. In order to demonstrate how DD can be applied we developed an example detector. The detector focuses on web server tampering threats, like spawning a reverse shell, bootstrapping the installation of a bot, or planting a web-based backdoor. These threats can be grouped into the broader *'malicious remote control'* attacker objective. The signatures for the detector are based on the heuristic that in order to achieve malicious remote control, an

D. Balzarotti, S. Stolfo, and M. Cova (Eds.): RAID 2012, LNCS 7462, pp. 384–385, 2012.

attacker requires the establishment of a remote connection to the target server. Only after a connection is established can the attacker proceed, e.g. to execute shell commands to search for sensitive information, or use the victim as an attack bot.

A malicious remote connection can be established either through a new connection over an unused server port, or a new connection to an already listened to web server port. Establishing a connection to an already listened to web server port actually allows clients to connect to the web server and use its services. However, an attacker could leverage it to first maliciously extend the code-base of the web application and to subsequently activate the code through its URL. In this context, all events indicating the establishment of a connection or a listening port by web application server processes or their child processes, and those indicating the addition of web application code, are considered attack symptoms.

Any HTTP request that contains executable content, intended either for dynamic in-memory injection into request processing code or for static injection into the application's code on-disk, is regarded as suspicious. Either type of injection is necessary to establish a network connection or extend the application code-base. The first alert correlation condition associates suspects alerts raised against executable content intended for dynamic injection with network connection symptom alerts by matching the IP address-port in the system call traces generated by the executable content, with those of the subsequent network connection event. The second correlation condition associates suspect alerts raised for executable content intended for static injection with the code-base extension symptom alert by matching the corresponding blocks of code.

Results and Conclusions. We implemented the detector for a LAMP deployment and we evaluated its resilience to novel attacks by assessing its ability to effectively detect attacks targeting a vulnerable `phpBB3` installation. Fourteen attacks were used employing different vulnerability exploits (heap-overflow, command and code injection, and unrestricted file upload), different payloads (reverse-shell, botzilla PHP-based IRC bot download and execute, and c99 backdoor installation), and different obfuscation techniques (XOR, base-64 and PHP obfuscation) within the detection scope of the detector. Attacks were mixed with traffic based on our departmental `phpBB` server. Despite large numbers of suspect requests and symptoms, the detector was able to detect all fourteen attacks with no FP. Building on these promising results we are now developing additional detectors to further explore the potential of distress detection.

References

1. Li, Z., Das, A., Zhou, J.: Model Generalization and Its Implications on Intrusion Detection. In: Ioannidis, J., Keromytis, A.D., Yung, M. (eds.) ACNS 2005. LNCS, vol. 3531, pp. 222–237. Springer, Heidelberg (2005)
2. Peisert, S., Bishop, M., Karin, S., Marzullo, K.: Towards models for forensic analysis. In: Proceedings of the Second International Workshop on Systematic Approaches to Digital Forensic Engineering (2007)

Trie Data Structure to Compare Traffic Payload in a Supervised Anomaly Detection System (Poster Abstract)

Jenny Andrea Pinto Sánchez and Luis Javier García Villalba

Group of Analysis, Security and Systems (GASS)
Department of Software Engineering and Artificial Intelligence (DISIA)
School of Computer Science, Universidad Complutense de Madrid (UCM)
Calle Profesor José García Santesmases s/n
Ciudad Universitaria, 28040 Madrid, Spain
jepinto@estumail.ucm.es, javiergv@fdi.ucm.es

1 Extended Abstract

Through an Anomaly Detection System, unknown attacks could be detected using a model of normal network behavior to distinguish between usual and unusual activities. Collecting representative data of normal activity and properly train the system are the deciding factors in a Supervised Intrusion Detection System. This work aims to propose a supervised anomaly detection system to detect unknown intrusions using the packet payload in the network, implementing its detection algorithm as a "dynamic pre-processor" of Snort. Critical infrastructures are exposed to a several threats which demand computer network protection. An Intrusion Detection System (IDS) provides adequate protection of process control networks. IDSs are usually classified into misuse/signature detection and anomaly detection. Signature-based IDS typically exhibit high detection accuracy because it identifies attacks based on known attack characteristics. Anomaly detection is the alternative approach to detect novel attacks tagging suspicious events. Learning a model of normal traffic and report deviations from the normal behavior is the main strength of anomaly based detection system. The major weakness is that it is susceptible to false positive alarms.

Supervised technique consists of two different phases: training and detection. Training phase is the critical one because the system needs to collect representative data from network connections to build an appropriate model which enable to set up thresholds for intrusion alerts. As such, we focus this work on training phase which use comparison of connection payloads based on *n-grams* to facilitate bytes sequences analysis and tries to store different **n-grams** occurrences and simplify searching and comparing sequences.

Considering that Snort is a signature-based IDS and unknown attacks are beyond their reach, we propose a supervised anomaly detection system that works as a Snort dynamic preprocessor in order to have a highly effective detection within known and unknown attacks. In general, the proposed system takes into account byte payload sequences represented by fixed-length *n-grams* and the

D. Balzarotti, S. Stolfo, and M. Cova (Eds.): RAID 2012, LNCS 7462, pp. 386–387, 2012.

correlations among *n-gram* arranged in a *trie* data structure. The aim is that these components enable a system with high performance at training phase and facilitate more and novel attack detection. Several proposals that make depth analysis of the payload to detect malicious code are explicated. In this work, *n-grams* are used to change bytes from connection payload to sequences of fixed length n, in order to reduce storage and analysis range. We believe that the system could store larger *n-gram* values to build the correct nor-mal network model if a trie data structure is used. This approach will increase the detection of unknown attacks. The classical scheme for storing and comparing *n-gram* models utilizes several data structures such as hash table, sorted arrays and suffix trees. However a better alternative for storing and comparing *n-grams* is a trie data structure. A trie is an *N-ary* tree, whose nodes are *N-place* vectors with components corresponding to digits or characters. Tries are considered as an optimal data structure over *n-grams* with higher **n** values applied over NIDS [1].

A payload could be embedded using a trie of *n-grams*; new *n-grams* are represent-ed and the others must be taken into account in the leaves that store the number of sequence concurrencies. One-way branching with no common prefix are removed to take advantage of redundancy in the stored sequences. A trie data structure is not only used efficiently in computation of similarity measures, but also in searching and comparing sequences. The main advantages in the way of storing and retrieving in-formation are: shorter access time, elimination of n-gram redundancies and inherent symbolic addressing [2]. Supervised system involves model selection from normal and abnormal datasets. Normal datasets are used to found patterns in the network and then these patterns are applied to dataset with attacks to establish rules for detection phase. Particularly, tries data structures are needed to compare current package *n-grams* with a previous one to find out a successful network model.

The idea is to perform some tests to determine if the speed in the basic training phase improves and if it is possible to store larger size of *n-grams*. Testing is performed taking into account a NIDS based on payload which use Bloom Filter as a data structure. The Bloom Filter represents different *n-grams* using arrays, each position denote a sequence of bits with the amount of occurrences, while tries represents each *n-gram* as a string of characters. A trie-node denotes a char in a sequence.

Acknowledgments. This work was supported by the MITyC (Spain) through the Project TSI-020100-2011-165 and by AECID (Spain) through the Project A1/037528/11.

References

1. Rieck, K., Laskov, P.: Detecting Unknown Network Attacks Using Language Models. In: Büschkes, R., Laskov, P. (eds.) DIMVA 2006. LNCS, vol. 4064, pp. 74–90. Springer, Heidelberg (2006)
2. Fredkin, E.: Trie memory. Communications of ACM 3(9), 490–499 (1960)

Towards Automated Forensic Event Reconstruction of Malicious Code (Poster Abstract)

Ahmed F. Shosha, Joshua I. James, Chen-Ching Liu, and Pavel Gladyshev

University College Dublin, Ireland
Ahmed.Shosha@ucdconnect.ie,
{Joshua.James,Liu,Pavel.Gladyshev}@ucd.ie

Abstract. A call for formalizing digital forensic investigations has been proposed by academics and practitioners alike [1, 2]. Many currently proposed methods of malware analysis for forensic investigation purposes, however, are derived based on the investigators' practical experience. This paper presents a formal approach for reconstructing the activities of a malicious executable found in a victim's system during a post-mortem analysis. The behavior of a suspect executable is modeled as a finite state automaton where each state represents behavior that results in an observable modification to the victim's system. The derived model of the malicious code allows for accurate reasoning and deduction of the occurrence of malicious activities even when anti-forensic methods are employed to disrupt the investigation process.

Keywords: Formal Models, Event Reconstruction, Model Checking and Automated Static Malware Analysis.

Introduction: This work introduces a formal model for automated malware investigation based on the modeling of malicious executables. In the proposed approach, malicious code is analyzed using automated static analysis methods [3-5]. The malicious code's control flow graph is then formally modeled as a finite state automaton (FSA). The formalized model of the malicious code behavior is processed by an extension of the event reconstruction algorithms proposed in [2, 6], which computes the set of all possible explanations for the state of the victim's system in the context of the malicious code where the observed state of the victim's system and malware trace creation states intersect. The result is a reduced state-space where malicious actions agree with the observed state of the system. Furthermore, the modeled FSA allows for the inference of the occurrence of actions that do not leave an observable trace.

***Modeling Investigated Malicious Code*:** Malicious executable *IE* is formally defined as a sequence of instructions$\langle I_1, I_2 \dots . I_n\rangle$. The behavior of *IE* is represented in a finite state automata $M = (Q, \sum, \delta, q)$, where Q is a finite set of all possible instructions in *IE* and δ represents transition function that determines the next instruction I_m for every possible combination of event and instruction state I_q, such that, $\delta: \sum \times Q \rightarrow Q$. A transition is the process of instruction execution. An execution path $p = (s_0, s_1 \dots s_q)$ is a run of finite computations consisting of a sequence of instructions that lead executable *IE* to the final state q.

D. Balzarotti, S. Stolfo, and M. Cova (Eds.): RAID 2012, LNCS 7462, pp. 388–389, 2012.

***Malicious Events Reconstruction*:** is the process of determining all possible execution paths that are consistent with observable evidence. In this approach, we extend and improve a formal model for automated reasoning of evidential statements and reconstruction of digital events proposed in [2]. The extended formal model is based on back-tracing execution paths that hold the observation O_x. The proposed back-tracing technique over all possible execution paths is based on the finite computation $c_j = \langle c_j^{\sigma}, I_j^{q} \rangle$, where, $c_j^{\sigma} \in \Sigma$ is an event and $I_j^{q} \in Q$ is a state. Any two instructions $I_k$and $I_{k-1}$are related via the transition function for a given instruction $I_j^{q} = \delta\left(c_{k-1}^{\sigma}, I_{k-1}^{q}\right)$. The notation of back-tracing an execution path is formalized in Equation 1, where ψ^{-1} traces back all finite computations representing the execution paths in the malicious executable *IE*.

$$\psi^{-1}(Q) = \bigcup^{\forall I \in Q} \psi^{-1}(I) \quad (1) \quad O = (P, min, opt, pr_c) \quad (2) \quad AG\, AF\, \mu \Rightarrow \psi^{-1}(p) \quad (3)$$

***Formalizing Malicious Code Observations*:** Evidence is described as an observable property, *O,* of a victim's system that denotes the execution of a malicious payload. The formalization of an observation is defined in Equation 2, where *P* is a set of all instructions in *IE* that have the observed property *pr*. *min* and *opt* are positive integers specifying the duration of the observation and pr_c is the set of characteristics of the observed property *pr*. An execution path *p* is said to contribute to *O* if a set of sequence of instructions in *p* possesses the observed property *pr*.

***Observation Consistency Checking*:** Anti-forensic techniques are formally encoded in a CTL specification model [7] μ. Using the proposed model checking algorithm, the model of a suspect executable *IE* is checked against the encoded techniques μ in the context of malicious code execution to identify tampered observations. The model checking algorithm takes a formula μ and executable model *IE* and verifies all states $s \in IE$ where μ holds. The notation of the model checking algorithm is formalized in Equation 3, where *A* is a quantifier over all paths *p* that contribute to the observation *o*, and *G/F* are a path specific quantifiers that check if μ holds over all states *s* and possess *o*.

References

1. Stephenson, P.: Using a Formalized Approach to Digital Investigation. In: Computer Fraud & Security (2003)
2. Gladyshev, P., Patel, A.: Finite state machine approach to digital event reconstruction. Digital Investigation (2004)
3. Christodorescu, M., Jha, S.: Static analysis of executables to detect malicious patterns. In: USENIX Security Symposium (2003)
4. Christodorescu, M., Jha, S., Kruegel, C.: Mining specifications of malicious behavior. In: ESEC-FSE (2007)
5. Kinder, J., Katzenbeisser, S., Schallhart, C., Veith, H.: Detecting Malicious Code by Model Checking. In: Julisch, K., Kruegel, C. (eds.) DIMVA 2005. LNCS, vol. 3548, pp. 174–187. Springer, Heidelberg (2005)
6. James, J., et al.: Analysis of Evidence Using Formal Event Reconstruction. Digital Forensics and Cyber Crime (2010)
7. Emerson, E.A.: Temporal and modal logic. In: van Jan, L. (ed.) Handbook of Theoretical Computer Science, vol. B (1990)

Accurate Recovery of Functions in a Retargetable Decompiler[⋆](Poster Abstract)

Lukáš Ďurfina, Jakub Křoustek, Petr Zemek, and Břetislav Kábele

Faculty of Information Technology, IT4Innovations Centre of Excellence,
Brno University of Technology, Božetěchova 1/2, 612 66 Brno, Czech Republic
{idurfina,ikroustek,izemek}@fit.vutbr.cz, xkabel03@stud.fit.vutbr.cz

Introduction. Decompilation is used for translation of executable files into a high-level language (HLL) representation. It is an important mechanism for information forensics and malware analysis. Retargetable decompilation represents a very difficult task because it must handle all the specific features of the target platform. Nevertheless, a retargetable decompiler can be used for any particular target platform and the resulting code is represented in a uniform way.

One of the essential features of each decompiler is a detection of functions. They need to be properly recovered and emitted in the output HLL representation. The quality of this process directly impacts the final decompilation results.

Retargetable function detection and recovery is quite hard because machine code differs dramatically based on the used language, its compiler, and the target platform. In the past, several methods of function detection and recovery have been proposed with varying results [1,2,3]. Some of them are hard-coded for a particular processor architecture (e.g. Intel x86) or not implemented at all. Furthermore, only a few methods aim at the detection of function arguments.

In this poster, we present a new, platform-independent method of function detection and recovery. This method was successfully adopted within an existing retargetable decompiler [4]. It is very briefly described in the following text.

Function Detection in the Retargetable Decompiler. At the beginning of the decompilation process, we try to locate the address of the `main` function. This is done by an entry point analysis. The following step is realized by a control-flow analysis and its aim is a detection of basic blocks. The detection is realized over the internal intermediate code. At this point, we need to find all branch instructions and their targets. We process all instructions and store addresses of every instruction which jumps, modifies the program counter, or changes the control flow by other ways. In some cases it can be more difficult, e.g. for architectures which use indirect jumps. We solve this problem by static-code interpretation (i.e. tracking register or memory values).

[⋆] This work was supported by the project TA ČR TA01010667 System for Support of Platform Independent Malware Analysis in Executable Files, BUT FIT grant FIT-S-11-2, and by the European Regional Development Fund in the IT4Innovations Centre of Excellence project (CZ.1.05/1.1.00/02.0070).

D. Balzarotti, S. Stolfo, and M. Cova (Eds.): RAID 2012, LNCS 7462, pp. 390–392, 2012.

The function analysis itself consists of several parts. The first part is function detection that is done by detectors utilizing the top-down approach [1], bottom-up approach [2], or debug information. Next comes a generation of a call graph. After that, we use a data-flow analysis to detect arguments, return values, and return addresses. The top-down detector tries to split a function containing the whole program into smaller ones by the given instructions. It works in iterations. Each iteration consists of finding targets of jumps and operation split. The bottom-up detector joins blocks together to create functions. In our modification, we have already created basic blocks and we can join them to functions. This detector is also used to improve the top-down approach, after every iteration in the top-down detector, it is called to make the bottom-up analysis on every detected function. Several other detectors are utilized to achieve the most accurate results (e.g. detectors of malware obfuscation of call conventions).

Experimental Results. The implemented method for recovering functions was tested on the MIPS and ARM architectures. We used our own tests that were compiled by different compilers at all available optimization levels. In total, 97 test cases were evaluated. The final results are shown in Table 1.

Table 1. Evaluation of overall results

	Functions [%]	Arguments [%]	Return Values [%]
Correctly detected	89	87,4	81,4
Wrongly detected	3,8	9,2	14,0
Undetected	7,2	3,4	4,6

The retargetable function detection achieves quite precise results. However, the number of false positives in the recovery of arguments and return values has to be reduced in future research. Their visualization is presented in Figure 1.

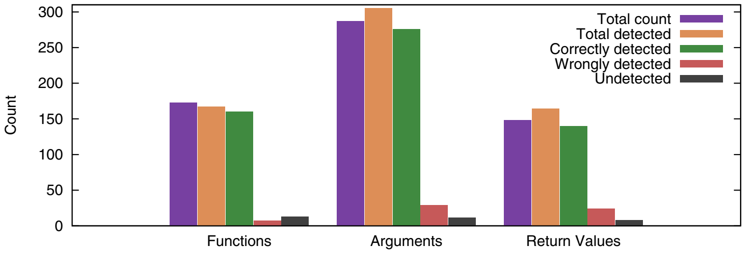


Fig. 1. Results of decompilation—detection of functions, arguments, and return values

Future Research. We propose two major areas for future research—(1) detection of inlined functions and (2) function reconstruction after obfuscation which is necessary for accurate decompilation of malware.

References

1. Kästner, D., Wilhelm, S.: Generic control flow reconstruction from assembly code. ACM SIGPLAN Notices 37(7) (July 2002)
2. Theiling, H.: Extracting safe and precise control flow from binaries. In: Proceedings of the 7th Conference on Real-Tim Computing Systems and Applications (2000)
3. Balakrishnan, G., Reps, T.: Analyzing Memory Accesses in x86 Executables. In: Duesterwald, E. (ed.) CC 2004. LNCS, vol. 2985, pp. 5–23. Springer, Heidelberg (2004)
4. Ďurfina, L., Křoustek, J., Zemek, P., Kolář, D., Masařík, K., Hruška, T., Meduna, A.: Design of a retargetable decompiler for a static platform-independent malware analysis. International Journal of Security and Its Applications 5(4), 91–106 (2011)

Improvement of an Anagram Based NIDS by Reducing the Storage Space of Bloom Filters (Poster Abstract)

Hugo Villanúa Vega, Jorge Maestre Vidal,
Jaime Daniel Mejía Castro, and Luis Javier García Villalba

Group of Analysis, Security and Systems (GASS)
Department of Software Engineering and Artificial Intelligence (DISIA)
School of Computer Science, Universidad Complutense de Madrid (UCM)
Calle Profesor José García Santesmases s/n
Ciudad Universitaria, 28040 Madrid, Spain
{hvillanua,jmaestre}@estumail.ucm.es, {j.mejia,javiergv}@fdi.ucm.es

1 Extended Abstract

When optimizing our NIDS APAP [1] we started focusing our efforts on ensuring that it would work on real-time network traffic. This effort, was penalized by the excessive cost of storage of various data structures needed to meet its goals satisfactorily.

APAP is based on Anagram [2] and initially worked with small size *N-gram*. This allowed us to detect more attacks at the expense of a higher rate of false positives. But when we wanted to test the results obtained with larger *N-gram* sizes, we found that the cost of storage of the Bloom filter structures that we generated to analyze the payload of the traffic was too large.

A Bloom filter [3] is a probabilistic data structure used to determine if a data belongs to a dataset. In our NIDS the Bloom filter is intended to store information of the appearance of N-grams in the package. When representing the Bloom filter, we considered it in theory as an array of bits on which a value of 0 indicates the non-appearance of that *N-gram* on normal network traffic and a value of 1 indicates its presence.

In practice, we used the Bloom filter as an array of bits on which a value of 0 indicates the non-appearance of that N-gram on normal network traffic and an n indicates its number of appearances. By using bits to represent n-grams less memory is required to store information of legitimate traffic.

Each N-gram found during training will have a different position in the Bloom filter, establishing a direct correspondence between each N-gram and the structure.

To get an idea of the cost to implement this type of structure, we will assume the hypothetical implementation of the Bloom filter as an array of integers in C. In the Table 1 we can see their progression.

To reduce this cost, we designed a simple data compression algorithm that can reconstruct the original structure from the file without penalizing the performance of the computer.

D. Balzarotti, S. Stolfo, and M. Cova (Eds.): RAID 2012, LNCS 7462, pp. 393–394, 2012.

Table 1. Chart:Increased storage space

N-gram	**Size (bytes)**
1-GRAM	1024 bytes
2-GRAM	262144 bytes
3-GRAM	64 Mbytes
4-GRAM	16 Gbytes

The idea is based on the high number of occurrences of most elements in heterogeneous networks. To reduce the amount of used memory to store this data, we could generate a file where the data did not have a regular structure such as 32-bit integer, but instead to use only the amount of bits required for each number, and saving that amount in a new file which would be read in parallel when we loading the structure.

As a first step we optimize the quantities by calculating the distance between adjacent positions of the array of appearances. This is accomplished by holding the first position of the array intact, and subtracting from every position value the previous position value, beginning from the end, to fill the resulting structure. In addition, an auxiliary array with the same size will indicate the number of bytes required to read these values of the resulting file.

To load the structure is only necessary to read the number of bits required to read each original number, read that amount in the file and calculate the new positions as the sum of the content of the previous position plus the content of the new read value. We are currently working on the implementation of that structure.

Acknowledgments. This work was supported by the Ministerio de Industria, Turismo y Comercio (MITyC, Spain) through the Project Avanza Competitividad I+D+I TSI-020100-2011-165 and the Agencia Española de Cooperación Internacional para el Desarrollo (AECID, Spain) through Acción Integrada MAEC-AECID MEDITERRÁNEO A1/037528/11.

References

1. García-Villalba, L.J., Mejía-Castro, J.D., Sandoval-Orozco, A.L., Martínez-Puentes, J.: Malware Detection System by Payload Analysis of Network Traffic. In: Proceedings of the 15th International Symposium on Research in Attacks, Intrusions and Defenses (September 2012)
2. Wang, K., Parekh, J.J., Stolfo, S.J.: Anagram: A Content Anomaly Detector Resistant to Mimicry Attack. In: Zamboni, D., Kruegel, C. (eds.) RAID 2006. LNCS, vol. 4219, pp. 226–248. Springer, Heidelberg (2006)
3. Broder, A., Mitzenmacher, M.: Network applications of bloom filters: A survey. In: Internet Mathematics, pp. 636–646 (2002)

Concurrency Optimization for NIDS (Poster Abstract)

Jorge Maestre Vidal, Hugo Villanúa Vega,
Jaime Daniel Mejía Castro, and Luis Javier García Villalba

Group of Analysis, Security and Systems (GASS)
Department of Software Engineering and Artificial Intelligence (DISIA)
School of Computer Science, Universidad Complutense de Madrid (UCM)
Calle Profesor José García Santesmases s/n
Ciudad Universitaria, 28040 Madrid, Spain
{jmaestre,hvillanua}@estumail.ucm.es, {j.mejia,javiergv}@fdi.ucm.es

1 Extended Abstract

The current demand of high network speed has led NIDS to process increasing amounts of information in less time. Consequently, most part of manufacturers have opted for hardware design implementation, which in most cases increased the price of these products. The aim of this paper focus the optimization of the performance of our NIDS APAP, based on different concurrency techniques. This upgrade increases amount of traffic per unit of time that is being processed by the system without relying on a hardware implementation. It is important to clarify that despite these measures can make our NIDS perform in real time on fast networks, it cannot achieve the same performance as a hardware implementation. As the first step it is interesting to briefly highlight some of the most important features of our initial prototype of IDS, APAP [1], with the purpose of getting into context. This system was developed as a hybrid NIDS combining signature and anomaly based detection. The system simultaneously executes Snort along with its preprocessors and an anomaly based detector whose design is based on Anagram [2]. We chose to work on CPU level parallelism using OpenMP libraries. These libraries provide an API that allows us to add concurrency to the application by means of shared memory parallelism. It is based on the creation of parallel execution threads that share variables from their parent process. OpenMP consists of a set of compiler directives, library routines, and environment variables that influence run-time behavior. The first thing to take into consideration is the degree of parallelization of the algorithm. Because the optimization could be in jeopardy if the threads context changes do not take place. Therefore, we created four testing suites corresponding to four different parallelization criteria. The first suite is a total parallelization of the algorithm, the other three are relaxations of the first by means of no parallelization of: fixed loop iterations, variable loop iterations and loops iterating to a concrete variable of the code, respectively. Notice that each suite includes the relaxations made on the previous ones. Figure 1 illustrates the time it took to run the algorithm depending on the number of threads for each level respect of the execution on

D. Balzarotti, S. Stolfo, and M. Cova (Eds.): RAID 2012, LNCS 7462, pp. 395–396, 2012.

a single thread. This analysis was done using a Core 2 DUO CPU processor, meaning a powerful performance may be achieved using more powerful processors. The trace used for the tests was ceded by the Computer Center of the Universidad Complutense de Madrid.

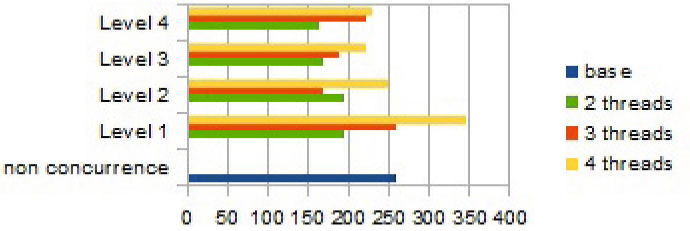


Fig. 1. Run Time of the Algorithm

We can clearly see how a 57% speed-up has been reached on the fourth suite using two threads. Motivated by the results, we have seen that the use of software NIDS on small to medium size networks can be feasible if concurrent techniques are correctly applied on its implementation, giving an economic and versatile approach as opposite to hardware NIDS. We are currently working on optimizing our NIDS by means of GPU level parallelism. Most part of the original code is already reconstructed using OpenCL libraries, which will execute the analysis on the Anagram segments from the package payload. We hope to obtain results soon enough.

Acknowledgments. This work was supported by the Ministerio de Industria, Turismo y Comercio (MITyC, Spain) through the Project Avanza Competitividad I+D+I TSI-020100-2011-165 and the Agencia Española de Cooperación Internacional para el Desarrollo (AECID, Spain) through Acción Integrada MAEC-AECID MEDITERRÁNEO A1/037528/11.

References

1. García-Villalba, L.J., Mejía-Castro, J.D., Sandoval-Orozco, A.L., Martínez-Puentes, J.: Malware Detection System by Payload Analysis of Network Traffic. In: Proceedings of the 15th International Symposium on Research in Attacks, Intrusions and Defenses (September 2012)
2. Wang, K., Parekh, J.J., Stolfo, S.J.: Anagram: A Content Anomaly Detector Resistant to Mimicry Attack. In: Zamboni, D., Kruegel, C. (eds.) RAID 2006. LNCS, vol. 4219, pp. 226–248. Springer, Heidelberg (2006)

Malware Detection System by Payload Analysis of Network Traffic (Poster Abstract)

Luis Javier García Villalba,
Jaime Daniel Mejía Castro, Ana Lucila Sandoval Orozco,
and Javier Martínez Puentes

Group of Analysis, Security and Systems (GASS)
Department of Software Engineering and Artificial Intelligence (DISIA)
School of Computer Science, Universidad Complutense de Madrid (UCM)
Calle Profesor José García Santesmases s/n
Ciudad Universitaria, 28040 Madrid, Spain
{javiergv,j.mejia,asandoval,javiermp}@fdi.ucm.es

1 Extended Abstract

NIDS based on Payload Analysis detect the malicious code by analyzing the payload of packets flowing through the network. Typically consist of a training phase and another one of detection. The training phase is done with clean traffic so that it represents statistically the usual traffic of the system. Thus, a pattern of such traffic is established. On the other hand, during the detection, traffic analysis is modeled and compared these patterns to determine if it can be classified as dangerous. Then, various proposals that make analysis of the payload to detect malicious code are explicated. In general, all are variants of PAYL [1], one of the first proposals that used this technique successfully. PAYL system classifies traffic based on three characteristics: the port, packet size and flow direction (input or output). Using these three parameters, payloads are classified creating a series of patterns to define what would be normal behavior within each class. Poseidon [2] was developed to correct the errors that arise in building models in PAYL when clustering about the size of packets is applied. The combination of multiple classifiers of a class, also based on PAYL, was developed to eliminate the original system's vulnerability in the face of mimicry attacks. PCNAD [3] appears to correct the defect PAYL that could not process large packets on fast networks with enough speed. Anagram is another evolution of PAYL, developed by the same authors to correct the deficiencies that had the original system. As in the PAYL, the system is based on n-grams to process the packets and create patterns of behavior. However, it employed Bloom Filters to divide the packets in n-grams of sizes larger than one without the cost in space and system performance will be injured.

Our proposal, *Advanced Payload Analyzer Preprocessor* (APAP), is an intrusion detection system by analysis of Payload from network traffic to look for some kind of Malware. The APAP system implements its detection algorithm as "dynamic pre-processor" of Snort. By working together of Snort and APAP

D. Balzarotti, S. Stolfo, and M. Cova (Eds.): RAID 2012, LNCS 7462, pp. 397–398, 2012.

someone can affirm that a highly effective system to known attacks (by passing Snort rules) and equally effective against new and unknown attacks (which was the main objective of the designed system) is obtained. To summarize the APAP working would suffice to say that, like most such systems, it consists of two phases: an initial training phase and a second phase of detection. During the training phase a statistical model of legitimate network traffic through the techniques Bloom Filters and n-grams is created. Then the results obtained by analyzing a dataset of attacks with this model is compared and thus a set of rules that will be able to determine whether a payload analyzed corresponds to some kind of Malware or otherwise be classified as legitimate traffic is obtained. During the detection phase the traffic for analyzing is passed by the Bloom Filter which it is created in the training phase and the obtained results with the rules that occurred during the training phase are compared. Due to the system relies on the creation of a normal traffic pattern of the network to defend, it is necessary to release a series of previous training steps before analysis, to detect the Malware that can circulate through the network. Thus it is noteworthy that for such training have to be available two datasets: a set of datasets consisting of traffic as clean as possible to be representative of the network. Thus, the system with the traffic "should" have regularly is training, and another set of datasets consisting of attacks to get the rules which will determine later whether a particular package corresponds to some kind of Malware or otherwise can be considered legitimate. Training phase creates a statistical model of network legitimate traffic using n-gram and BloomFilter techniques. This phase is divided into four stages: i) Initialization: the system is reset by creating the structures that need. ii) Basic-Training: the model of network legitimate traffic is created. iii) Reference-Training: it provides support to the heuristic used for the next stage. iv) Determine-K-Training: it creates the rules that determine which values will be launched alerts.

Acknowledgments. This work was supported by the Ministerio de Industria, Turismo y Comercio (MITyC, Spain) through the Project Avanza Competitividad I+D+I TSI-020100-2011-165 and the Agencia Española de Cooperación Internacional para el Desarrollo (AECID, Spain) through Acción Integrada MAEC-AECID MEDITERRÁNEO A1/037528/11.

References

1. Almeida, P.S., Baquero, C., Preguica, N., Hutchison, D.: Scalable bloom filters. Information Processing Letters 101(6), 255–261 (2007)
2. Zhang, Y., Li, T., Sun, J., Qin, R.: An FSM-Based Approach for Malicious Code Detection Using the Self-Relocation Gene. In: Huang, D.-S., Wunsch II, D.C., Levine, D.S., Jo, K.-H. (eds.) ICIC 2008. LNCS, vol. 5226, pp. 364–371. Springer, Heidelberg (2008)
3. Shafiq, M.Z., Tabish, S.M., Mirza, F., Farooq, M.: PE-Miner: Mining Structural Information to Detect Malicious Executables in Realtime. In: Balzarotti, D. (ed.) RAID 2009. LNCS, vol. 5758, pp. 121–141. Springer, Heidelberg (2009)

Author Index

Ahn, Dongkyun 107
Amann, Johanna 314

Bilge, Leyla 64
Binxing, Fang 378
Bolzoni, Damiano 354
Bos, Herbert 86

Carbone, Martim 22
Cárdenas, Alvaro A. 210
Castro, Jaime Daniel Mejía 393, 395, 397
Cavallaro, Lorenzo 86
Chaoge, Liu 376
Chen, Xin 147
Chhugani, Jatin 334
Chu, Jie 294
Conover, Matthew 22
Cui, Xiang 380

Desmedt, Yvo 374
Dubey, Pradeep 334
Ďurfina, Lukáš 390
dutt-Sharma, Nitish 86

Engel, Thomas 190
England, Paul 1
Etalle, Sandro 354

François, Jérôme 190

Ge, Zihui 294
Giffin, Jonathon 1
Gladyshev, Pavel 388
Gu, Guofei 230
Guo, Li 382

Hadžiosmanović, Dina 354
Hall, Seth 314
Hsu, Hungyuan 147
Huber, Richard 294
Hurson, Ali R. 147

James, Joshua I. 388
Ji, Ping 294
Jin, Shuyuan 380
Jinqiao, Shi 376
Jinyu, Wu 378
Johns, Martin 254

Kábele, Břetislav 390
Kaeli, David 127
Kiernan, Seán 64
Kim, Changkyu 334
Kim, Min Sik 334
Kirda, Engin 169
Křoustek, Jakub 390
Kruegel, Christopher 274

Lauinger, Tobias 169
Lee, Gyungho 107
Lee, Martin 64
Lee, Wenke 22
Lekies, Sebastian 254
Lihua, Yin 378
Liu, Chen-Ching 388
Liu, Peng 42
Liu, Tingwen 382

Mahmood, Shah 374
Mankin, Jennifer 127
Marchal, Samuel 190
Mashima, Daisuke 210
Michiardi, Pietro 169
Montague, Bruce 22

Nikiforakis, Nick 254

O'Gorman, Gavin 64
Orozco, Ana Lucila Sandoval 397

Peng, Liao 376
Piessens, Frank 254
Puentes, Javier Martínez 397

Qiao, Dengke 382

Raj, Himanshu 1
Roper, Marc 384

Sánchez, Jenny Andrea Pinto 386
Satish, Nadathur 334
Sharma, Aashish 314
Shosha, Ahmed F. 388
Simionato, Lorenzo 354
Sommer, Robin 314
Srivastava, Abhinav 1
State, Radu 190
Sun, Yan 334
Sun, Yong 382

Terzis, Sotirios 384
Thonnard, Olivier 64
Tighzert, Walter 254

Valgenti, Victor C. 334
van der Veen, Victor 86
Van Overveldt, Timon 274
Vega, Hugo Villanúa 393, 395
Vella, Mark 384
Vidal, Jorge Maestre 393, 395
Vigna, Giovanni 274
Villalba, Luis Javier García 386, 393, 395, 397

Xiang, Cui 376
Xu, Zhaoyan 230
Xu, Zhi 147

Yang, Chao 230
Yang, Zhi 380
Yates, Jennifer 294
Yu, Yung-Chao 294

Zambon, Emmanuele 354
Zemek, Petr 390
Zhang, Jialong 230
Zhang, Shengzhi 42
Zhu, Sencun 147