Richard Lippmann
Engin Kirda
Ari Trachtenberg (Eds.)

LNCS 5230

Recent Advances in Intrusion Detection

11th International Symposium, RAID 2008
Cambridge, MA, USA, September 2008
Proceedings

Springer

Richard Lippmann Engin Kirda
Ari Trachtenberg (Eds.)

Recent Advances in Intrusion Detection

11th International Symposium, RAID 2008
Cambridge, MA, USA, September 15-17, 2008
Proceedings

Springer

Volume Editors

Richard Lippmann
Lincoln Laboratory
Massachusetts Institute of Technology
Lexington, MA, USA
E-mail: lippmann@ll.mit.edu

Engin Kirda
Institut Eurecom
Sophia-Antipolis, France
E-mail: engin.kirda@eurecom.fr

Ari Trachtenberg
Boston University
Boston, MA, USA
E-mail: trachten@bu.edu

Library of Congress Control Number: 2008934305

CR Subject Classification (1998): K.6.5, K.4, E.3, C.2, D.4.6

LNCS Sublibrary: SL 4 – Security and Cryptology

ISSN	0302-9743
ISBN-10	3-540-87402-X Springer Berlin Heidelberg New York
ISBN-13	978-3-540-87402-7 Springer Berlin Heidelberg New York

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India
Printed on acid-free paper SPIN: 12511846 06/3180 5 4 3 2 1 0

Preface

On behalf of the Program Committee, it is our pleasure to present the proceedings of the 11th International Symposium on Recent Advances in Intrusion Detection (RAID 2008), which took place in Cambridge, Massachusetts, USA on September 15–17.

The symposium brought together leading researchers and practitioners from academia, government and industry to discuss intrusion detection research and practice. There were six main sessions presenting full-fledged research papers (rootkit prevention, malware detection and prevention, high performance intrusion and evasion, web application testing and evasion, alert correlation and worm detection, and anomaly detection and network traffic analysis), a session of posters on emerging research areas and case studies, and two panel discussions ("Government Investments: Successes, Failures and the Future" and "Life after Antivirus - What Does the Future Hold?").

The RAID 2008 Program Committee received 80 paper submissions from all over the world. All submissions were carefully reviewed by at least three independent reviewers on the basis of space, topic, technical assessment, and overall balance. Final selection took place at the Program Committee meeting on May 23rd in Cambridge, MA. Twenty papers were selected for presentation and publication in the conference proceedings, and four papers were recommended for resubmission as poster presentations.

As a new feature this year, the symposium accepted submissions for poster presentations, which have been published as extended abstracts, reporting early-stage research, demonstration of applications, or case studies. Thirty-nine posters were submitted for a numerical review by an independent, three-person subcommittee of the Program Committee based on novelty, description, and evaluation. The subcommittee chose to recommend the acceptance of 16 of these posters for presentation and publication.

The success of RAID 2008 depended on the joint effort of many people. We would like to thank all the authors who submitted papers, whether accepted or not. We would also like to thank the Program Committee members and additional reviewers, who volunteered their time to carefully evaluate the numerous submissions. In addition, we would like to thank the General Chair, Rob Cunningham, for handling the conference arrangements, Ari Trachtenberg for handling publication, Jon Giffin for publicizing the conference, Anup Ghosh for finding sponsors for the conference, and MIT Lincoln Lab for maintaining the conference website. Finally, we extend our thanks to The Institute for Information Infrastructure Protection (I3P), Symantec Corporation, IBM, and MIT Lincoln Laboratory for their sponsorship of student scholarships.

June 2008

Richard Lippmann
Engin Kirda

Organization

RAID 2008 was organized by MIT Lincoln Laboratory and held in conjunction with VIZSEC 2008.

Conference Chairs

Conference Chair	Robert Cunningham (MIT Lincoln Laboratory)
Program Chair	Richard Lippmann (MIT Lincoln Laboratory)
Program Co-chair	Engin Kirda (Eurecom / Technical University of Vienna)
Publications Chair	Ari Trachtenberg (Boston University)
Publicity Chair	Jon Giffin (Georgia Tech)
Sponsorship Chair	Anup Ghosh (George Mason University)

Program Committee

Michael Bailey	University of Michigan
Michael Behringer	Cisco
Herbert Bos	Vrije Universiteit
David Brumley	Carnegie Mellon University
Tzi-cker Chiueh	State University of New York at Stony Brook
Andrew Clark	Queensland University of Technology
Robert Cunningham	MIT Lincoln Lab
Ulrich Flegel	SAP Research
Debin Gao	Singapore Management University
Anup Ghosh	George Mason University
Jonathon Giffin	Georgia Institute of Technology
Thorsten Holz	University of Mannheim
Jaeyeon Jung	Intel
Engin Kirda	Institute Eurecom
Kwok-Yan Lam	Tsinghua University
Zhuowei Li	Microsoft
Richard Lippmann	MIT Lincoln Laboratory
Raffael Marty	Splunk
Benjamin Morin	Supélec
Rei Safavi-Naini	University of Calgary
R. Sekar	State University of New York at Stony Brook
Robin Sommer	ICSI and LBNL
Salvatore Stolfo	Columbia University
Toshihiro Tabata	Okayama University
Ari Trachtenberg	Boston University

Vijay Varadharajan	Macquarie University
Andreas Wespi	IBM Zurich Research Laboratory
Diego Zamboni	IBM Zurich Research Laboratory
Jianying Zhou	Institute for Infocomm Research

Steering Committee

Marc Dacier (Chair)	EURECOM, France
Hervé Debar	France Télécom R&D, France
Deborah Frincke	Pacific Northwest National Lab, USA
Ming-Yuh Huang	The Boeing Company, USA
Erland Jonsson	Chalmers, Sweden
Wenke Lee	Georgia Tech, USA
Ludovic Mé	Supélec, France
Alfonso Valdes	SRI International, USA
Giovanni Vigna	University of California, Santa Barbara, USA
Andreas Wespi	IBM Research, Switzerland
S. Felix Wu	UC Davis, USA
Diego Zamboni	IBM Research, Switzerland
Christopher Kruegel	University of California, Santa Barbara, USA / Technical University of Vienna, Austria

Additional Reviewers

Hirotake Abe	Toyohashi University of Technology
Manos Antonakakis	Georgia Tech
Venkat Balakrishnan	Macquarie University
Ulrich Bayer	Technical University of Vienna
Leyla Bilge	Technical University of Vienna
Damiano Bolzoni	University of Twente
Gabriela Cretu	Columbia University
Italo Dacosta	Georgia Tech
Loic Duflot	DCSSI
Thomas Dullien	Zynamics
Jose M. Fernandez	École Polytechnique de Montréal
Vanessa Frias-Martinez	Columbia University
Jochen Haller	SAP Research
Philip Hendrix	Harvard University
Yoshiaki Hori	Kyushu University
Kyle Ingols	MIT Lincoln Laboratory
Florian Kerschbaum	SAP Research
Hyung Chan Kim	Columbia University
Andreas Lang	University of Magdeburg
Pavel Laskov	Fraunhofer FIRST & University of Tuebingen
Timothy Leek	MIT Lincoln Laboratory

Sponsoring Institutions

The Institute for Information Infrastructure Protection (I3P)
Symantec Corporation
IBM
MIT Lincoln Laboratory

Table of Contents

Posters

Guest-Transparent Prevention of Kernel Rootkits with VMM-Based Memory Shadowing

Ryan Riley[1], Xuxian Jiang[2], and Dongyan Xu[1]

[1] CERIAS and Department of Computer Science, Purdue University
{rileyrd,dxu}@cs.purdue.edu
[2] Department of Computer Science, North Carolina State University
jiang@cs.ncsu.edu

Abstract. Kernel rootkits pose a significant threat to computer systems as they run at the highest privilege level and have unrestricted access to the resources of their victims. Many current efforts in kernel rootkit defense focus on the *detection* of kernel rootkits – after a rootkit attack has taken place, while the smaller number of efforts in kernel rootkit *prevention* exhibit limitations in their capability or deployability. In this paper we present a kernel rootkit prevention system called NICKLE which addresses a common, fundamental characteristic of most kernel rootkits: the need for executing their own kernel code. NICKLE is a lightweight, virtual machine monitor (VMM) based system that transparently prevents unauthorized kernel code execution for unmodified commodity (guest) OSes. NICKLE is based on a new scheme called *memory shadowing*, wherein the trusted VMM maintains a shadow physical memory for a running VM and performs real-time kernel code authentication so that only authenticated kernel code will be stored in the shadow memory. Further, NICKLE transparently routes guest kernel instruction fetches to the shadow memory at runtime. By doing so, NICKLE guarantees that only the authenticated kernel code will be executed, foiling the kernel rootkit's attempt to strike in the first place. We have implemented NICKLE in three VMM platforms: QEMU+KQEMU, VirtualBox, and VMware Workstation. Our experiments with 23 real-world kernel rootkits targeting the Linux or Windows OSes demonstrate NICKLE's effectiveness. Furthermore, our performance evaluation shows that NICKLE introduces small overhead to the VMM platform.

1 Introduction

Kernel-level rootkits have proven to be a formidable threat to computer systems: By subverting the operating system (OS) kernel, a kernel rootkit embeds itself into the compromised kernel and stealthily inflicts damages with full, unrestricted access to the system's resources. Effectively omnipotent in the compromised systems, kernel rootkits have increasingly been used by attackers to hide their presence and prolong their control over their victims.

There have been a number of recent efforts in mitigating the threat of kernel rootkits and they can mainly be classified into two categories: (1) detecting the presence of kernel rootkits in a system [1, 2, 3, 4, 5] and (2) preventing the compromise of OS kernel integrity [6, 7]. In the first category, Copilot [4] proposes the use of a separate PCI card to periodically grab the memory image of a running OS kernel and analyze it to determine if the kernel has been compromised. The work which follows up Copilot [2] further extends that capability by detecting the violation of kernel integrity using semantic specifications of static and dynamic kernel data. SBCFI [3] reports violations of the kernel's control flow integrity using the kernel's control-flow graph. One common attribute of approaches in this category is the *detection* of a kernel rootkit's presence based on certain symptoms exhibited by the kernel *after* the kernel rootkit has already struck. As a result, these approaches are, by design, not capable of *preventing kernel rootkit execution in the first place.*

In the second category, Livewire [6], based on a virtual machine monitor (VMM), aims at protecting the guest OS kernel code and critical kernel data structures from being modified. However, without modifying the original kernel code, an attacker may choose to load malicious rootkit code into the kernel space by either exploiting kernel vulnerabilities or leveraging certain kernel features (e.g., loadable kernel module support in modern OSes). More recently, SecVisor [7] is proposed as a hypervisor-based solution to enforce the W$\oplus$X property of memory pages of the guest machine, with the goal of preventing unauthorized code from running with kernel-level privileges. SecVisor requires modifying kernel source code and needs the latest hardware-based virtualization support and thus does not support closed-source OSes or legacy hardware platforms. Moreover, SecVisor is not able to function if the OS kernel has *mixed* pages that contain both code and data. Unfortunately, such mixed kernel pages do exist in modern OSes (e.g., Linux and Windows as shown in Section 2.2).

To complement the existing approaches, we present NICKLE ("No Instruction Creeping into Kernel Level Executed")[1], a lightweight, VMM-based system that provides an important guarantee in kernel rootkit prevention: *No unauthorized code can be executed at the kernel level.* NICKLE achieves this guarantee on top of legacy hardware and without requiring guest OS kernel modification. As such, NICKLE is readily deployable to protect unmodified guest OSes (e.g., Fedora Core 3/4/5 and Windows 2K/XP) against kernel rootkits. NICKLE is based on observing a common, fundamental characteristic of most modern kernel rootkits: their ability to execute unauthorized instructions at the kernel level. By removing this ability, NICKLE significantly raises the bar for successfully launching kernel rootkit attacks.

To achieve the "NICKLE" guarantee, we first observe that a kernel rootkit is able to access the entire physical address space of the victim machine. This observation inspires us to impose restricted access to the instructions in the kernel space: only *authenticated* kernel instructions can be fetched for execution. Obviously, such a restriction cannot be enforced by the OS kernel itself. Instead,

[1] With a slight abuse of terms, we use NICKLE to denote both the system itself and the guarantee achieved by the system – when used in quotation marks.

a natural strategy is to enforce such memory access restriction using the VMM, which is at a privilege level higher than that of the (guest) OS kernel.

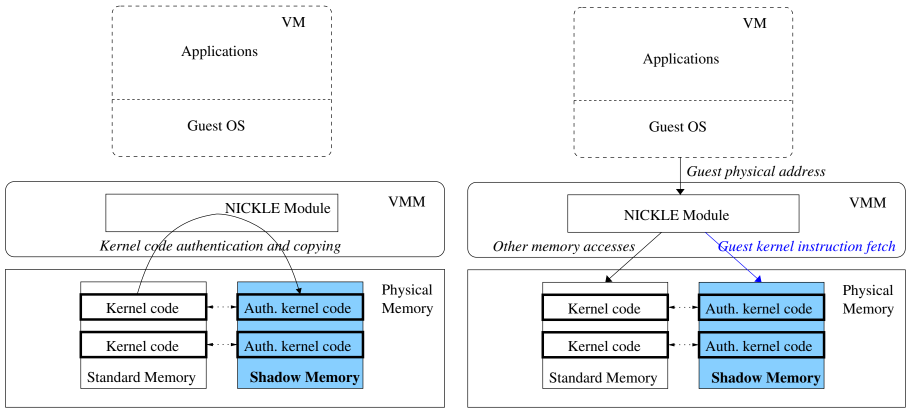
Our main challenge is to realize the above VMM-level kernel instruction fetch restriction in a guest-transparent, real-time, and efficient manner. An intuitive approach would be to impose W⊕X on kernel memory pages to protect existing kernel code and prevent the execution of injected kernel code. However, due to the existence of mixed kernel pages in commodity OSes, this approach is not viable for guest-transparent protection. To address that, we propose a VMM-based *memory shadowing* scheme for NICKLE that will work in the face of mixed kernel pages. More specifically, for a virtual machine (VM), the VMM creates two distinct physical memory regions: a *standard memory* and a *shadow memory*. The VMM enforces that the guest OS kernel cannot access the shadow memory. Upon the VM's startup, the VMM performs kernel code authentication and dynamically copies authenticated kernel instructions from the standard memory to the shadow memory. At runtime, any instruction executed in the kernel space must be fetched from the shadow memory instead of from the standard memory. To enforce this while maintaining guest transparency, a lightweight *guest memory access indirection* mechanism is added to the VMM. As such, a kernel rootkit will never be able to execute any of its own code as the code injected into the kernel space will not be able to reach the shadow memory.

We have implemented NICKLE in three VMMs: QEMU[8] with the KQEMU accelerator, VirtualBox [9], and VMware Workstation. Our evaluation results show that NICKLE incurs a reasonable impact on the VMM platform (e.g., 1.01% on QEMU+KQEMU and 5.45% on VirtualBox when running UnixBench). NICKLE is shown capable of transparently protecting a variety of commodity OSes, including RedHat 8.0 (Linux 2.4.18 kernel), Fedora Core 3 (Linux 2.6.15 kernel), Windows 2000, and Windows XP. Our results show that NICKLE is able to prevent and gracefully respond to 23 real-world kernel rootkits targeting the above OSes, without requiring details of rootkit attack vectors. Finally, our porting experience indicates that the NICKLE design is generic and realizable in a variety of VMMs.

2 NICKLE Design

2.1 Design Goals and Threat Model

Goals and Challenges. NICKLE has the following three main design goals: *First*, as its name indicates, NICKLE should prevent any unauthorized code from being executed in the kernel space of the protected VM. The challenges of realizing this goal come from the real-time requirement of prevention as well as from the requirement that the guest OS kernel should not be trusted to initiate any task of the prevention – the latter requirement is justified by the kernel rootkit's highest privilege level inside the VM and the possible existence of zero-day vulnerabilities inside the guest OS kernel. NICKLE overcomes these challenges using the VMM-based memory shadowing scheme (Section 2.2). We



(a) Kernel code authorization and copying (b) Guest physical address redirection

Fig. 1. Memory shadowing scheme in NICKLE

note that the scope of NICKLE is focused on preventing unauthorized kernel code execution. The prevention of other types of attacks (e.g., data-only attacks) is a non-goal and related solutions will be discussed in Section 5.

Second, NICKLE should not require modifications to the guest OS kernel. This allows commodity OSes to be supported "as is" without recompilation and reinstallation. Correspondingly, the challenge in realizing this goal is to make the memory shadowing scheme transparent to the VM with respect to both the VM's function and performance.

Third, the design of NICKLE should be generically portable to a range of VMMs. Given this, the challenge is to ensure that NICKLE has a small footprint within the VMM and remains lightweight with respect to performance impact. In this paper we focus on supporting NICKLE in software VMMs. However, we expect that the exploitation of recent hardware-based virtualization extensions [10, 11] will improve NICKLE's performance even further.

In addition, it is also desirable that NICKLE facilitate various flexible response mechanisms to be activated upon the detection of an unauthorized kernel code execution attempt. A flexible response, for example, is to cause only the offending process to fail without stopping the rest of the OS. The challenge in realizing this is to initiate flexible responses entirely from outside the protected VM and minimize the side-effects on the running OS.

Threat Model and System Assumption. We assume the following adversary model when designing NICKLE: (1) The kernel rootkit has the highest privilege level inside the victim VM (e.g., the *root* privilege in a UNIX system); (2) The kernel rootkit has full access to the VM's memory space (e.g., through `/dev/mem` in Linux); (3) The kernel rootkit aims at stealthily maintaining and hiding its presence in the VM and to do so, the rootkit will need to execute its own (malicious) code in the kernel space. We note that such a need exists in most kernel rootkits today, and we will discuss possible exceptions in Section 5.

Meanwhile, we assume a trusted VMM that provides VM isolation. This assumption is shared by many other VMM-based security research efforts [1, 6, 12, 13, 14, 15]. We will discuss possible attacks (e.g., VM fingerprinting) in Section 5. With this assumption, we consider the threat from DMA attacks launched from physical hosts outside of the scope of this work.[2]

2.2 Enabling Scheme and Techniques

Memory Shadowing. The memory shadowing scheme enforces the "NICKLE" property: For a VM, apart from its standard physical memory space, the VMM also allocates a separate physical memory region as the VM's *shadow memory* (Figure 1) which is transparent to the VM and controlled by the VMM. Upon the startup of the VM's OS, all known-good, authenticated guest kernel instructions will be copied from the VM's standard memory to the shadow memory (Figure 1(a)). At runtime, when the VM is about to execute a kernel instruction, the VMM will transparently redirect the kernel instruction fetch to the shadow memory (Figure 1(b)). All other memory accesses (to user code, user data, and kernel data) will proceed unhindered in the standard memory.

The memory shadowing scheme is motivated by the observation that modern computers define a single memory space for all code – both kernel code and user code – and data. With the VMM running at a higher privilege level, we can now "shadow" the guest kernel code space with elevated (VMM-level) privileges to ensure that the guest OS kernel itself cannot access the shadowed kernel code space containg the authenticated kernel instructions. By doing so, even if a kernel rootkit is able to inject its own code into the VM's standard memory, the VMM will ensure that the malicious code never gets copied over to the shadow memory. Moreover, an attempt to execute the malicious code can be caught immediately due to the inconsistency between the standard and shadow memory contents.

The astute reader may be asking "How is NICKLE functionally different from W⊕X?" In essence, W⊕X is a scheme that enforces the property, "A given memory page will never be both writable and executable at the same time." The basic premise behind this scheme is that if a page cannot be written to and later executed from, code injection becomes impossible. There are two main reasons why this scheme is not adequate for stopping kernel level rootkits:

First, W⊕X is not able to protect mixed kernel pages with both code and data, which do exist in current OSes. As a specific example, in a Fedora Core 3 VM (with the 32-bit 2.6.15 kernel and the NX protection), the Linux kernel stores the main static kernel text in memory range $[0xc0100000, 0xc02dea50]$ and keeps the system call table starting from virtual address $0xc02e04a0$. Notice that the Linux kernel uses a large page size ($2MB$) to manage the physical memory,[3] which means that the first two kernel pages cover memory ranges

[2] There exists another type of DMA attack that is initiated from within a guest VM. However, since the VMM itself virtualizes or mediates the guest DMA operations, NICKLE can be easily extended to intercede and block them.

[3] If the NX protection is disabled, those kernel pages containing static kernel text will be of $4MB$ in size.

$[0xc0000000, 0xc0200000)$ and $[0xc0200000, 0xc0400000)$, respectively. As a result, the second kernel page contains both code and data, and thus must be marked both writable and executable – This conflicts with the W⊕X scheme. Mixed pages also exist for accommodating the code and data of Linux loadable kernel modules (LKMs) – an example will be shown in Section 4.1. For the Windows XP kernel (with SP2), our investigation has confirmed the existence of mixed pages as well [16]. On the other hand, NICKLE is able to protect mixed pages.[4]

Second, W⊕X assumes only one execution privilege level while kernel rootkit prevention requires further distinction between user and kernel code pages. For example, a page may be set executable in user mode but non-executable in kernel mode. In other words, the sort of permission desired is not W⊕X, but W⊕KX (i.e. not writable and kernel-executable at the same time.) Still, we point out that the enforcement of W⊕KX is *not* effective for mixed kernel pages and, regardless, not obvious to construct on current processors that do not allow such fine-grained memory permissions.

Another question that may be asked is, "Why adopt memory shadowing when one could simply guard kernel code by keeping track of the ranges of valid kernel code addresses ?" Indeed, NICKLE is guided by the principle of kernel code guarding, but does so differently from the brute-force approach of tracking/checking kernel code address ranges – mainly for performance reasons. More specifically, the brute-force approach could store the address ranges of valid kernel code in a data structure (e.g., tree) with $O(logN)$ search time. On the other hand, memory shadowing allows us to locate the valid kernel instruction in the shadow memory in $O(1)$ time thus significantly reducing the processing overhead. In addition, memory shadowing makes it convenient to compare the instructions in the shadow memory to those in the standard memory. If they differ (indicating malicious kernel code injection or modification), a number of response actions can be implemented based on the difference (details in Section 3).

Guest Memory Access Indirection. To realize the guest memory shadowing scheme, two issues need to be resolved. First, how does NICKLE fill up the guest shadow memory with authenticated kernel code? Second, how does NICKLE fetch authenticated kernel instructions for execution while detecting and preventing any attempt to execute unauthorized code in the kernel space? We note that our solutions have to be transparent to the guest OS (and thus to the kernel rootkits). We now present the guest memory access indirection technique to address these issues.

[4] We also considered the option of eliminating mixed kernel pages. However, doing so would require kernel source code modification, which conflicts with our second design goal. Even given source code access, mixed page elimination is still a complex task (more than just page-aligning data). In fact, a kernel configuration option with a similar purpose exists in the latest Linux kernel (version 2.6.23). But after we enabled the option, we still found more than 700 mixed kernel pages. NICKLE instead simply avoids such complexity and works even with mixed kernel pages.

Guest memory access indirection is performed between the VM and its memory (standard and shadow) by a thin NICKLE module inside the VMM. It has two main functions, kernel code authentication and copying at VM startup and upon kernel module loading as well as guest physical address redirection at runtime (Figure 1).

Kernel Code Authentication and Copying. To fill up the shadow memory with authenticated kernel instructions, the NICKLE module inside the VMM needs to first determine the accurate timing for kernel code authentication and copying. To better articulate the problem, we will use the Linux kernel as an example. There are two specific situations throughout the VM's lifetime when kernel code needs to be authorized and shadowed: One at the VM's startup and one upon the loading/unloading of loadable kernel modules (LKMs). When the VM is starting up, the guest's shadow memory is empty. The kernel bootstrap code then decompresses the kernel. Right after the decompression and before any processes are executed, NICKLE will use a cryptographic hash to verify the integrity of the kernel code (this is very similar to level 4 in the secure bootstrap procedure [17]) and then copy the authenticated kernel code from the standard memory into the shadow memory (Figure 1(a)). As such, the protected VM will start with a known clean kernel.

The LKM support in modern OSes complicates our design. From NICKLE's perspective, LKMs are considered injected kernel code and thus need to be authenticated and shadowed before their execution. The challenge for NICKLE is to *externally* monitor the guest OS and detect the kernel module loading/unloading events in real-time. NICKLE achieves this by leveraging our earlier work on non-intrusive VM monitoring and semantic event reconstruction [1, 14]. When NICKLE detects the loading of a new kernel module, it intercepts the VM's execution and performs kernel module code authentication and shadowing. The authentication is performed by taking a cryptographic hash of the kernel module's code segment and comparing it with a known correct value, which is computed a priori off-line and provided by the administrator or distribution maintainer.[5] If the hash values don't match, the kernel module's code will not be copied to the shadow memory.

Through kernel code authentication and copying, only authenticated kernel code will be loaded into the shadow memory, thus blocking the copying of malicious kernel rootkit code or any other code injected by exploiting kernel vulnerabilities, including zero-day vulnerabilities. It is important to note that neither kernel startup hashing nor kernel module hashing assumes trust in the guest OS. Should the guest OS fail to cooperate, *no* code will be copied to the shadow memory, and any execution attempts from that code will be detected and refused.

Guest Physical Address Redirection. At runtime, the NICKLE module inside the VMM intercepts the memory accesses of the VM *after* the "guest virtual address $\rightarrow$ guest physical address" translation. As such, NICKLE does not interfere

[5] We have developed an off-line kernel module profiler that, given a legitimate kernel module, will compute the corresponding hash value (Section 3.1).

with – and is therefore transparent to – the guest OS's memory access handling procedure and virtual memory mappings. Instead, it takes the guest physical address, determines the type of the memory access (kernel, user; code, data; etc.), and routes it to either the standard or shadow memory (Figure 1(b)).

We point out that the interception of VM memory accesses can be provided by existing VMMs (e.g., QEMU+KQEMU, VirtualBox, and VMware). NICKLE builds on this interception capability by adding the guest physical address redirection logic. First, using a simple method to check the current privilege level of the processor, NICKLE determines whether the current instruction fetch is for kernel code or for user code: If the processor is in supervisor mode (CPL=0 on x86), we infer that the fetch is for kernel code and NICKLE will verify and route the instruction fetch to the shadow memory. Otherwise, the processor is in user mode and NICKLE will route the instruction fetch to the standard memory. Data accesses of either type are always routed to the standard memory.

One might object that an attacker may strive to ensure that his injected kernel code will run when the processor is in user mode. However, this creates a significant challenge wherein the attacker would have to fundamentally change a running kernel to operate in both supervisor and user mode *without changing any existing kernel code*. The authors do not consider such a rootkit to be a possibility without a severe loss of rootkit functionality.

Flexible Responses to Unauthorized Kernel Code Execution Attempts If an unauthorized execution attempt is detected, a natural follow-up question is, "How should NICKLE respond to an attempt to execute an unauthenticated kernel instruction?" Given that NICKLE sits between the VM and its memory and has a higher privilege level than the guest OS, it possesses a wide range of options and capabilities to respond. We describe two response modes facilitated by the current NICKLE system.

Rewrite mode: NICKLE will dynamically rewrite the malicious kernel code with code of its own. The response code can range from OS-specific error handling code to a well-crafted payload designed to clean up the impact of a rootkit installation attempt. Note that this mode may require an understanding of the guest OS to ensure that valid, sensible code is returned.

Break mode: NICKLE will take no action and route the instruction fetch to the *shadow memory*. In the case where the attacker only modifies the original kernel code, this mode will lead to the execution of the original code – a desirable situation. However, in the case where *new* code is injected into the kernel, this mode will lead to an instruction fetch from presumably null content (containing 0s) in the shadow memory. As such, break mode prevents malicious kernel code execution but may or may not be graceful depending on how the OS handles invalid code execution faults.

3 NICKLE Implementation

To validate the portability of the NICKLE design, we have implemented NICKLE in three VMMs: QEMU+KQEMU [8], VirtualBox [9], and VMware

Workstation[6]. Since the open-source QEMU+KQEMU is the VMM platform where we first implemented NICKLE, we use it as the representative VMM to describe our implementation details. For most of this section, we choose RedHat 8.0 as the default guest OS. We will also discuss the limitations of our current prototype in supporting Windows guest OSes.

3.1 Memory Shadowing and Guest Memory Access Indirection

To implement memory shadowing, we have considered two options: (1) NICKLE could interfere as instructions are executed; or (2) NICKLE could interfere when instructions are dynamically translated. Note that dynamic instruction translation is a key technique behind existing software-based VMMs, which transparently translates guest machine code into native code that will run in the physical host. We favor the second option for performance reasons: By being part of the translator, NICKLE can take advantage of the fact that translated code blocks are cached. In QEMU+KQEMU, for example, guest kernel instructions are grouped into "blocks" and are dynamically translated at runtime. After a block of code is translated, it is stored in a cache to make it available for future execution. In terms of NICKLE, this means that if we intercede during code translation we need not intercede as often as we would if we did so during code execution, resulting in a smaller impact on system performance.

The pseudo-code for memory shadowing and guest memory access indirection is shown in Algorithm 1. Given the guest physical address of an instruction to be executed by the VM, NICKLE first checks the current privilege level of the processor (CPL). If the processor is in supervisor mode, NICKLE knows that it is executing in kernel mode. Using the guest physical address, NICKLE compares the content of the standard and shadow memories to determine whether the kernel instruction to be executed is already in the shadow memory (namely has been authenticated). If so, the kernel instruction is allowed to be fetched, translated, and executed. If not, NICKLE will determine if the guest OS kernel is being bootstrapped or a kernel module is being loaded. If either is the case, the corresponding kernel text or kernel module code will be authenticated and, if successful, shadowed into the shadow memory. Otherwise, NICKLE detects an attempt to execute an unauthorized instruction in the kernel space and prevents it by executing our response to the attempt.

In Algorithm 1, the way to determine whether the guest OS kernel is being bootstrapped or a kernel module is being loaded requires OS-specific knowledge. Using the Linux 2.4 kernel as an example, when the kernel's *startup_32* function, located at physical address `0x00100000` or virtual address `0xc0100000` as shown in the *System.map* file, is to be executed, we know that this is the first

[6] We acknowledge the VMware Academic Program for providing the source code. Due to space and licensing constraints, however, the VMware port is not further discussed or evaluated in this work. Some additional discussion of the port is available in our technical report [16].

Algorithm 1. Algorithm for Memory Shadowing and Guest Memory Access Indirection

```
Input: (1) GuestPA: guest physical address of instruction to be executed; (2) ShadowMEM[]:
       shadow memory; (3) StandardMEM[]: standard memory
1 if !IsUserMode(vcpu) AND ShadowMEM[GuestPA] != StandardMEM[GuestPA] then
2     if (kernel is being bootstrapped) OR (module is being loaded) then
3         Authenticate and shadow code;
4     else
5         Unauthorized execution attempt - Execute response;
6     end
7 end
8 Fetch, translate, and cache code;
```

instruction executed to load the kernel and we can intercede appropriately. For kernel module loading, there is a specific system call to handle that. As such, the NICKLE module inside the VMM can intercept the system call and perform kernel module authentication and shadowing right before the module-specific *init_module* routine is executed.

In our implementation, the loading of LKMs requires special handling. More specifically, providing a hash of a kernel module's code space ends up being slightly complicated in practice. This is due to the fact that kernel modules are dynamically relocatable and hence some portions of the kernel module's code space may be modified by the module loading function. Accordingly, the cryptographic hash of a loaded kernel module will be different depending on where it is relocated to. To solve this problem, we perform an off-line, a priori profiling of the legitimate kernel module binaries. For each known good module we calculate the cryptographic hash by excluding the portions of the module that will be changed during relocation. In addition, we store a list of bytes affected by relocation so that the same procedure can be repeated by NICKLE during runtime hash evaluation of the same module.

We point out that although the implementation of NICKLE requires certain guest OS-specific information, it does *not* require modifications to the guest OS itself. Still, for a closed-source guest OS (e.g., Windows), lack of information about kernel bootstrapping and dynamic kernel code loading may lead to certain limitations. For example, not knowing the timing and "signature" of dynamic (legal) kernel code loading events in Windows, the current implementation of NICKLE relies on the administrator to designate a time instance when all authorized Windows kernel code has been loaded into the standard memory. Not knowing the exact locations of the kernel code, NICKLE traverses the shadow page table and copies those executable pages located in the kernel space from the standard memory to the shadow memory, hence creating a "gold standard" to compare future kernel code execution against. From this time on, NICKLE can transparently protect the Windows OS kernel from executing any unauthorized kernel code. Moreover, this limited implementation can be made complete when the relevant information becomes available through vendor disclosure or reverse engineering.

3.2 Flexible Response

In response to an attempt to execute an unauthorized instruction in the kernel space, NICKLE provides two response modes. Our initial implementation of NICKLE simply re-routes the instruction fetch to the shadow memory for a string of zeros (break mode). As to be shown in our experiments, this produces some interesting outcomes: a Linux guest OS would react to this by triggering a kernel fault and terminating the offending process. Windows, on the other hand, reacts to the NICKLE response by immediately halting with a blue screen – a less graceful outcome.

In search of a more flexible response mode, we find that by rewriting the offending instructions at runtime (rewrite mode), NICKLE can respond in a less disruptive way. We also observe that most kernel rootkits analyzed behave the following way: They first insert a new chunk of malicious code into the kernel space; then they somehow ensure their code is `call`'d as a function. With this observation, we let NICKLE dynamically replace the code with `return -1;`, which in assembly is: `mov $0xffffffff, %eax; ret`. The main kernel text or the kernel module loading process will interpret this as an error and gracefully handle it: Our experiments with Windows 2K/XP, Linux 2.4, and Linux 2.6 guest OSes all confirm that NICKLE's rewrite mode is able to handle the malicious kernel code execution attempt by triggering the OS to terminate the offending process without causing a fault in the OS.

3.3 Porting Experience

We have experienced no major difficulty in porting NICKLE to other VMMs. The NICKLE implementations in both VMMs are lightweight: The SLOC (source lines of code) added to implement NICKLE in QEMU+KQEMU, VirtualBox, and VMware Workstation are 853, 762, and 1181 respectively. As mentioned earlier, we first implemented NICKLE in QEMU+KQEMU. It then took less than one week for one person to get NICKLE functional in VirtualBox 1.5.0 OSE, details of which can be found in our technical report [16].

4 NICKLE Evaluation

4.1 Effectiveness Against Kernel Rootkits

We have evaluated the effectiveness of NICKLE with 23 real-world kernel rootkits. They consist of nine Linux 2.4 rootkits, seven Linux 2.6 rootkits, and seven Windows rootkits[7] that can infect Windows 2000 and/or XP. The selected rootkits cover the main attack platforms and attack vectors thus providing a good representation of the state-of-the-art kernel rootkit technology. Table 1 shows

[7] There is a Windows rootkit named hxdef or Hacker Defender, which is usually classified as a user-level rootkit. However, since hxdef contains a device driver which will be loaded into the kernel, we consider it a kernel rootkit in this paper.

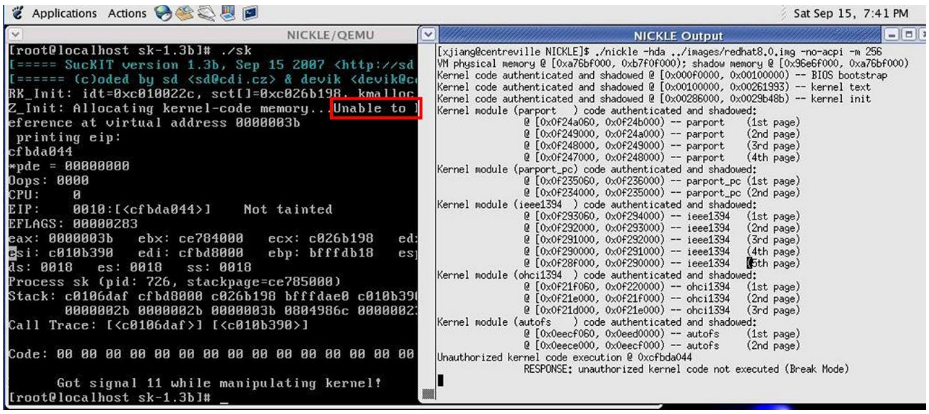
Table 1. Effectiveness of NICKLE in detecting and preventing 23 real-world kernel rootkits (*DKOM*[†] is a common rootkit technique which directly manipulates kernel objects; *"partial"*[‡] means the in-kernel component of the Hacker Defender rootkit fails; *BSOD*[§] stands for "Blue Screen Of Death")

Guest OS	Rootkit	Attack Vector	Outcome of NICKLE Response			
			Rewrite Mode		Break Mode	
			Prevented?	Outcome	Prevented?	Outcome
Linux 2.4	adore 0.42, 0.53	LKM	✓	`insmod` fails	✓	Seg. fault
	adore-ng 0.56	LKM	✓	`insmod` fails	✓	Seg. fault
	knark	LKM	✓	`insmod` fails	✓	Seg. fault
	rkit 1.01	LKM	✓	`insmod` fails	✓	Seg. fault
	kbdv3	LKM	✓	`insmod` fails	✓	Seg. fault
	allroot	LKM	✓	`insmod` fails	✓	Seg. fault
	rial	LKM	✓	`insmod` fails	✓	Seg. fault
	Phantasmagoria	LKM	✓	`insmod` fails	✓	Seg. fault
	SucKIT 1.3b	`/dev/kmem`	✓	Installation fails silently	✓	Seg. fault
Linux 2.6	adore-ng 0.56	LKM	✓	`insmod` fails	✓	Seg. fault
	eNYeLKM v1.2	LKM	✓	`insmod` fails	✓	Seg. fault
	sk2rc2	`/dev/kmem`	✓	Installation fails	✓	Seg. fault
	superkit	`/dev/kmem`	✓	Installation fails	✓	Seg. fault
	mood-nt 2.3	`/dev/kmem`	✓	Installation fails	✓	Seg. fault
	override	LKM	✓	`insmod` fails	✓	Seg. fault
	Phalanx b6	`/dev/mem`	✓	Installation crashes	✓	Seg. fault
Windows 2K/XP	FU	DKOM[†]	✓	Driver loading fails	✓	BSOD[§]
	FUTo	DKOM	✓	Driver loading fails	✓	BSOD
	he4hook 215b6	Driver	✓	Driver loading fails	✓	BSOD
	hxdef 1.0.0 revisited	Driver	partial[‡]	Driver loading fails	✓	BSOD
	hkdoor11	Driver	✓	Driver loading fails	✓	BSOD
	yyt_hac	Driver	✓	Driver loading fails	✓	BSOD
	NT Rootkit	Driver	✓	Driver loading fails	✓	BSOD

our experimental results: NICKLE is able to detect and prevent the execution of malicious kernel code in *all* experiments using both rewrite and break response modes. Finally, we note that NICKLE in all three VMMs is able to achieve the same results. In the following, we present details of two representative experiments. Some additional experiments are presented in [16].

SucKIT Rootkit Experiment. The SucKIT rootkit [18] for Linux 2.4 infects the Linux kernel by directly modifying the kernel through the `/dev/kmem` interface. During installation SucKIT first allocates memory within the kernel, injects its code into the allocated memory, and then causes the code to run as a function. Figure 2 shows NICKLE preventing the SucKIT installation. The window on the left shows the VM running RedHat 8.0 (with 2.4.18 kernel), while the window on the right shows the NICKLE output. Inside the VM, one can see that the SucKIT installation program fails and returns an error message "*Unable to handle kernel NULL pointer dereference*". This occurs because NICKLE (operating in break mode) foils the execution of injected kernel code by fetching a string of zeros from the shadow memory, which causes the kernel to terminate the rootkit installation program. Interestingly, when NICKLE operates in rewrite mode, it rewrites the malicious code and forces it to return −1. However, it seems that SucKIT does not bother to check the return value and so the rootkit installation just fails silently and the kernel-level functionality does not work.

In the right-side window in Figure 2, NICKLE reports the authentication and shadowing of sequences of kernel instructions starting from the initial BIOS



```
[root@localhost sk-1.3b]# ./sk
[===== SucKIT version 1.3b, Sep 15 2007 <http://sd
[====== (c)oded by sd <sd@cdi.cz> & devik <devik@c
RK_Init: idt=0xc010022c, sct[]=0xc026b198, kmalloc
Z_Init: Allocating kernel-code memory...Unable to 
eference at virtual address 0000003b
 printing eip:
cfbda044
*pde = 00000000
Oops: 0000
CPU:    0
EIP:    0010:[<cfbda044>]    Not tainted
EFLAGS: 00000283
eax: 0000003b   ebx: ce784000   ecx: c026b198   ed
esi: c010b390   edi: cfbd8000   ebp: bfffdb18   es
ds: 0018   es: 0018   ss: 0018
Process sk (pid: 726, stackpage=ce785000)
Stack: c0106daf cfbd8000 c026b198 bfffdae0 c010b39
       0000002b 0000002b 0000003b 0804986c 000002
Call Trace: [<c0106daf>] [<c010b390>]

Code: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

      Got signal 11 while manipulating kernel!
[root@localhost sk-1.3b]# _
```

```
[xjiang@centreville NICKLE]$ ./nickle -hda ../images/redhat8.0.img -no-acpi -m 256
VM physical memory @ [0xa76bf000, 0xb7f0f000); shadow memory @ [0x96e6f000, 0xa76bf000)
Kernel code authenticated and shadowed @ [0x000f0000, 0x00100000) -- BIOS bootstrap
Kernel code authenticated and shadowed @ [0x00100000, 0x00261993) -- kernel text
Kernel code authenticated and shadowed @ [0x00286000, 0x0029b48b) -- kernel init
Kernel module (parport   ) code authenticated and shadowed:
                @ [0x0f24a060, 0x0f24b000) -- parport    (1st page)
                @ [0x0f249000, 0x0f24a000) -- parport    (2nd page)
                @ [0x0f248000, 0x0f249000) -- parport    (3rd page)
                @ [0x0f247000, 0x0f248000) -- parport    (4th page)
Kernel module (parport_pc) code authenticated and shadowed:
                @ [0x0f235060, 0x0f236000) -- parport_pc (1st page)
                @ [0x0f234000, 0x0f235000) -- parport_pc (2nd page)
Kernel module (ieee1394  ) code authenticated and shadowed:
                @ [0x0f293060, 0x0f294000) -- ieee1394   (1st page)
                @ [0x0f292000, 0x0f293000) -- ieee1394   (2nd page)
                @ [0x0f291000, 0x0f292000) -- ieee1394   (3rd page)
                @ [0x0f290000, 0x0f291000) -- ieee1394   (4th page)
                @ [0x0f28f000, 0x0f290000) -- ieee1394   (5th page)
Kernel module (ohci1394  ) code authenticated and shadowed:
                @ [0x0f21f060, 0x0f220000) -- ohci1394   (1st page)
                @ [0x0f21e000, 0x0f21f000) -- ohci1394   (2nd page)
                @ [0x0f21d000, 0x0f21e000) -- ohci1394   (3rd page)
Kernel module (autofs    ) code authenticated and shadowed:
                @ [0x0eecf060, 0x0eed0000) -- autofs     (1st page)
                @ [0x0eece000, 0x0eecf000) -- autofs     (2nd page)
Unauthorized kernel code execution @ 0xcfbda044
                RESPONSE: unauthorized kernel code not executed (Break Mode)
```

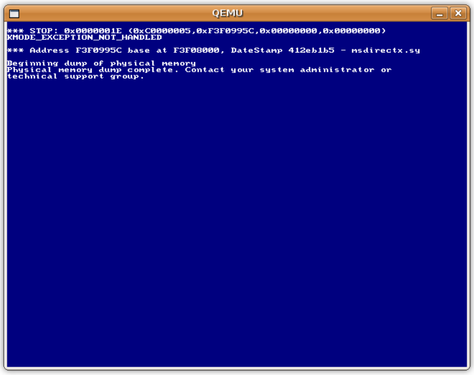
Fig. 2. NICKLE/QEMU+KQEMU foils the SucKIT rootkit (guest OS: RedHat 8.0)

bootstrap code to the kernel text as well as its initialization code and finally to various legitimate kernel modules. In this experiment, there are five legitimate kernel modules, *parport.o*, *parport_pc.o*, *ieee1394.o*, *ohci1394*, and *autofs.o*, all authenticated and shadowed. The code portion of the kernel module begins with an offset of `0x60` bytes in the first page. The first `0x60` bytes are for the kernel module header, which stores pointers to information such as the module's name, size, and other entries linking to the global linked list of loaded kernel modules. This is another example of *mixed kernel pages* with code and data in Linux (Section 2.2).

FU Rootkit Experiment. The FU rootkit [19] is a Windows rootkit that loads a kernel driver and proceeds to manipulate kernel data objects. The manipulation will allow the attacker to hide certain running processes or device drivers loaded in the kernel. When running FU on NICKLE, the driver is unable to load successfully as the driver-specific initialization code is considered unauthorized kernel code. Figure 3 compares NICKLE's two response modes against FU's attempt to load its driver. Under break mode, the OS simply breaks with a blue screen. Under rewrite mode, the FU installation program fails ("Failed to initialize driver.") but the OS does not crash.

4.2 Impact on Performance

To evaluate NICKLE's impact on system performance we have performed benchmark-based measurements on both VMMs – with and without NICKLE. The physical host in our experiments has an Intel 2.40GHz processor and 3GB of RAM running Ubuntu Linux 7.10. QEMU version 0.9.0 with KQEMU 1.3.0pre11 or VirtualBox 1.5.0 OSE is used where appropriate. The VM's guest OS is Redhat 8.0 with a custom compile of a vanilla Linux 2.4.18 kernel and is started inuniprocessor mode with the default amount of memory (256MB for

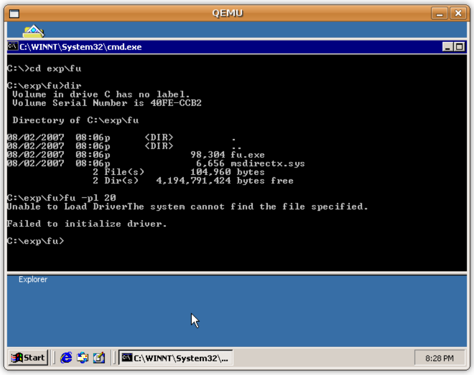


```
*** STOP: 0x0000001E (0xC0000005,0xF3F0995C,0x00000000,0x00000000)
KMODE_EXCEPTION_NOT_HANDLED

*** Address F3F0995C base at F3F08000, DateStamp 412eb1b5 - msdirectx.sy

Beginning dump of physical memory
Physical memory dump complete. Contact your system administrator or
technical support group.
```

(a) Under break mode



```
C:\>cd exp\fu

C:\exp\fu>dir
 Volume in drive C has no label.
 Volume Serial Number is 40FE-CCB2

 Directory of C:\exp\fu

08/02/2007  08:06p      <DIR>          .
08/02/2007  08:06p      <DIR>          ..
08/02/2007  08:06p              98,304 fu.exe
08/02/2007  08:06p               6,656 msdirectx.sys
               2 File(s)        104,960 bytes
               2 Dir(s)   4,194,791,424 bytes free

C:\exp\fu>fu -pl 20
Unable to Load DriverThe system cannot find the file specified.

Failed to initialize driver.

C:\exp\fu>
```

(b) Under rewrite mode

Fig. 3. Comparison of NICKLE/QEMU+KQEMU's response modes against the FU rootkit (guest OS: Windows 2K)

Table 2. Software configuration for performance evaluation

Item	**Version**	**Configuration**	**Item**	**Version**	**Configuration**
Redhat	8.0	Using Linux 2.4.18	Apache	2.0.59	Using the default high-performance configuration file
Kernel	2.4.18	Standard kernel compilation	ApacheBench	2.0.40-dev	`-c3 -t 60 <url/file>`
			Unixbench	4.1.0	`-10 index`

Table 3. Application benchmark results

	QEMU+KQEMU			**VirtualBox**		
Benchmark	**w/o NICKLE**	**w/NICKLE**	**Overhead**	**w/o NICKLE**	**w/ NICKLE**	**Overhead**
Kernel Compiling	231.490s	233.529s	0.87%	156.482s	168.377s	7.06%
`insmod`	0.088s	0.095s	7.34%	0.035s	0.050s	30.00%
Apache	351.714 req/s	349.417 req/s	0.65%	463.140 req/s	375.024 req/s	19.03%

VirtualBox and 128MB for QEMU+KQEMU). Table 2 shows the software configuration for the measurement. For the Apache benchmark, a separate machine connected to the host via a dedicated gigabit switch is used to launch ApacheBench. When applicable, benchmarks are run 10 times and the results are averaged.

Three application-level benchmarks (Table 3) and one micro-benchmark (Table 4) are used to evaluate the system. The first application benchmark is a kernel compilation test: A copy of the Linux 2.4.18 kernel is uncompressed, configured, and compiled. The total time for these operations is recorded and a lower number is better. Second, the `insmod` benchmark measures the amount of time taken to insert a module (in this case, the *ieee1394* module) into the kernel and again lower is better. Third, the ApacheBench program is used to measure the VM's throughput when serving requests for a 16KB file. In this case, higher is better. Finally, the UnixBench micro-benchmark is executed to evaluate the more fine-grained performance impact of NICKLE. The numbers

Table 4. UnixBench results (for the first two data columns, higher is better)

	QEMU+KQEMU			VirtualBox		
Benchmark	**w/o NICKLE**	**w/NICKLE**	**Overhead**	**w/o NICKLE**	**w/ NICKLE**	**Overhead**
Dhrystone	659.3	660.0	-0.11%	1843.1	1768.6	4.04%
Whetstone	256.0	256.0	0.00%	605.8	543.0	10.37%
Execl	126.0	127.3	-1.03%	205.4	178.2	13.24%
File copy 256B	45.5	46	-1.10%	2511.8	2415.7	3.83%
File copy 1kB	67.6	68.2	-0.89%	4837.5	4646.9	3.94%
File copy 4kB	128.4	127.4	0.78%	7249.9	7134.3	1.59%
Pipe throughput	41.7	40.7	2.40%	4646.9	4590.9	1.21%
Process creation	124.7	118.2	5.21%	92.1	85.3	7.38%
Shell scripts (8)	198.3	196.7	0.81%	259.2	239.8	7.48%
System call	20.9	20.1	3.83%	2193.3	2179.9	0.61%
Overall	106.1	105.0	**1.01%**	1172.6	1108.7	**5.45%**

reported in Table 4 are an index where higher is better. It should be noted that the benchmarks are meant primarily to compare a NICKLE-enhanced VMM with the corresponding unmodified VMM. These numbers are not meant to compare different VMMs (such as QEMU+KQEMU vs. VirtualBox).

QEMU+KQEMU. The QEMU+KQEMU implementation of NICKLE exhibits very low overhead in most tests. In fact, a few of the benchmark tests show a slight performance gain for the NICKLE implementation, but we consider these results to signify that there is no noticeable slowdown due to NICKLE for that test. From Table 3 it can be seen that both the kernel compilation and Apache tests come in below 1% overheard. The `insmod` test has a modest overhead, 7.3%, primarily due to the fact that NICKLE must calculate and verify the hash of the module prior to copying it into the shadow memory. Given how infrequently kernel module insertion occurs in a running system, this overhead is not a concern. The UnixBench tests in Table 4 further testify to the efficiency of the NICKLE implementation in QEMU+KQEMU, with the worst-case overhead of any test being 5.21% and the overall overhead being 1.01%. The low overhead of NICKLE is due to the fact that NICKLE's modifications to the QEMU control flow only take effect while executing kernel code (user-level code is executed by the unmodified KQEMU accelerator).

VirtualBox. The VirtualBox implementation has a more noticeable overhead than the QEMU+KQEMU implementation, but still runs below 10% for the majority of the tests. The kernel compilation test, for example, exhibits about 7% overheard; while the UnixBench suite shows a little less than 6% overall. The Apache test is the worst performer, showing a 19.03% slowdown. This can be attributed to the heavy number of user/kernel mode switches that occur while serving web requests. It is during the mode switches that the VirtualBox implementation does its work to ensure only verified code will be executed directly [16], hence incurring overhead. The `insmod` test shows a large performance degradation, coming in at 30.0%. This is due to the fact that module insertion on the VirtualBox implementation entails the VMM leaving native code execution as well as verifying the module. However, this is not a concern as module insertion is an uncommon event at runtime. Table 4 shows that the worst performing UnixBench test (Execl) results in an overhead of 13.24%. This result is most likely due to a larger number of user/kernel mode switches that occur during that test.

In summary, our benchmark experiments show that NICKLE incurs minimal to moderate impact on system performance, relative to that of the respective original VMMs.

5 Discussion

In this section, we discuss several issues related to NICKLE. First, the goal of NICKLE is to prevent unauthorized code from executing in the kernel space, but not to protect the integrity of kernel-level control flows. This means that it is possible for an attacker to launch a "return-into-libc" style attack within the kernel by leveraging only the existing authenticated kernel code. Recent work by Shacham [20] builds a powerful attacker who can execute virtually arbitrary code using only a carefully crafted stack that causes jumps and calls into existing code. Fortunately, this approach cannot produce *persistent* code to be called on demand from other portions of the kernel. And Petroni et al. [3] found that 96% of the rootkits they surveyed require persistent code changes. From another perspective, an attacker may also be able to directly or indirectly influence the kernel-level control flow by manipulating certain non-control data [21]. However, without its own kernel code, this type of attack tends to have limited functionality. For example, all four stealth rootkit attacks described in [22] need to execute their own code in the kernel space and hence will be defeated by NICKLE. Meanwhile, solutions exist for protecting control flow integrity [3, 23, 24] and data flow integrity [25], which can be leveraged and extended to complement NICKLE.

Second, the current NICKLE implementation does not support self-modifying kernel code. This limitation can be removed by intercepting the self-modifying behavior (e.g., based on the translation cache invalidation resulting from the self-modification) and re-authenticating and shadowing the kernel code after the modification.

Third, NICKLE currently does not support kernel page swapping. Linux does not swap out kernel pages, but Windows does have this capability. To support kernel page swapping in NICKLE, it would require implementing the introspection of swap-out and swap-in events and ensuring that the page being swapped in has the same hash as when it was swapped out. Otherwise an attacker could modify swapped out code pages without NICKLE noticing. This limitation has not yet created any problem in our experiments, where we did not encounter any kernel level page swapping.

Fourth, targeting kernel-level rootkits, NICKLE is ineffective against user-level rootkits. However, NICKLE significantly elevates the trustworthiness of the guest OS, on top of which anti-malware systems can be deployed to defend against user-level rootkits more effectively.

Fifth, the deployment of NICKLE increases the memory footprint for the protected VM. In the worst case, memory shadowing will double the physical memory usage. As our future work, we can explore the use of demand-paging to effectively reduce the extra memory requirement to the actual amount of memory needed. Overall, it is reasonable and practical to trade memory space for elevated OS kernel security.

Finally, we point out that NICKLE assumes a trusted VMM to achieve the "NICKLE" property. This assumption is needed because it essentially establishes the root-of-trust of the entire system and secures the lowest-level system access. We also acknowledge that a VM environment can potentially be fingerprinted and detected [26, 27] by attackers so that their malware can exhibit different behavior [28]. We can improve the fidelity of the VM environment (e.g., [29, 30]) to thwart some of the VM detection methods. Meanwhile, as virtualization continues to gain popularity, the concern over VM detection may become less significant as attackers' incentive and motivation to target VMs increases.

6 Related Work

Rootkit Prevention Through Kernel Integrity Enforcement. The first area of related work includes recent efforts in enforcing kernel integrity to thwart kernel rootkit installation or execution. Livewire [6], based on a software-based VMM, aims at protecting the guest OS kernel code and critical data structures from being modified. However, an attacker may choose to load malicious rootkit code into the kernel space without manipulating the original kernel code.

SecVisor [7] is a closely related work that leverages new hardware extensions to enforce life-time kernel integrity and provide a guarantee similar to "NICKLE". However, there are two main differences between SecVisor and NICKLE: First, the deployment of SecVisor requires modification to OS kernel source code as well as the latest hardware support for MMU and IOMMU virtualization. In comparison, NICKLE is a guest-transparent solution that supports guest OSes "as is" on top of legacy hardware platforms. In particular, NICKLE does not rely on the protection of any guest OS data structures (e.g., the GDT – global descriptor table). Second, SecVisor is developed to enforce the W$\oplus$X principle for the protected VM kernel code. This principle intrinsically conflicts with mixed kernel pages, which exist in current OSes (e.g., Linux and Windows). NICKLE works in the presence of mixed kernel pages. OverShadow [31] adopts a similar technique of memory shadowing at the VMM level with the goal of protecting application memory pages from modification by even the OS itself. In comparison, NICKLE has a different goal and aims at protecting the OS from kernel rootkits.

To ensure kernel code integrity, techniques such as driver signing [32] as well as various forms of driver verification [5, 33] have also been proposed. These techniques are helpful in verifying the identity or integrity of the loaded driver. However, a kernel-level vulnerability could potentially be exploited to bypass these techniques. In comparison, NICKLE operates at the lower VMM level and is capable of blocking zero-day kernel-level exploitations.

Symptom-Driven Kernel Rootkit Detection. The second area of related work is the modeling and specification of symptoms of a rootkit-infected OS kernel which can be used to detect kernel rootkits. Petroni et al. [4] and Zhang et al. [34] propose the use of external hardware to grab the runtime OS memory image and detect possible rootkit presence by spotting certain kernel code integrity violations (e.g., rootkit-inflicted kernel code manipulation). More recent works further identify possible violations of semantic integrity of dynamic kernel data [2] or state based control-flow integrity of kernel code [3]. Generalized control-flow integrity [23] may have strong potential to be used as a prevention technique, but as yet has not been applied to kernel integrity. Other solutions such as Strider GhostBuster [35] and VMwatcher [1] target the self-hiding nature of rootkits and infer rootkit presence by detecting discrepancies between the views of the same system from different perspectives. All the above approaches are, by design, for the *detection* of a kernel rootkit *after* it has infected a system. Instead, NICKLE is for the *prevention* of kernel rootkit execution in the first place.

Attestation-Based Rootkit Detection. The third area of related work is the use of attestation techniques to verify the software running on a target platform. Terra [13] and other code attestation schemes [36, 37, 38] are proposed to verify software that is being located into the memory for execution. These schemes are highly effective in providing the *load-time* attestation guarantee. Unfortunately, they are not able to provide *run-time* kernel integrity.

7 Conclusion

We have presented the design, implementation, and evaluation of NICKLE, a VMM-based approach that transparently detects and prevents the launching of kernel rootkit attacks against guest VMs. NICKLE achieves the "NICKLE" guarantee, which foils the common need of existing kernel rootkits to execute their own unauthorized code in the kernel space. NICKLE is enabled by the scheme of memory shadowing, which achieves guest transparency through the guest memory access indirection technique. NICKLE's portability has been demonstrated by its implementation in three VMM platforms. Our experiments show that NICKLE is effective in preventing 23 representative real-world kernel rootkits that target a variety of commodity OSes. Our measurement results show that NICKLE adds only modest overhead to the VMM platform.

Acknowledgements. The authors would like to thank the anonymous reviewers for their insightful comments that helped improve the presentation of this paper. This work was supported in part by NSF Grants CNS-0716376, CNS-0716444 and CNS-0546173.

References

[1] Jiang, X., Wang, X., Xu, D.: Stealthy Malware Detection through VMM-Based "Out-of-the-Box" Semantic View Reconstruction. In: Proceedings of the ACM Conference on Computer and Communications Security (CCS 2007) (October 2007)
[2] Petroni Jr., 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)
[3] Petroni Jr., N.L., Hicks, M.: Automated Detection of Persistent Kernel Control-Flow Attacks. In: Proceedings of the ACM Conference on Computer and Communications Security (CCS 2007) (October 2007)
[4] Petroni, N., Fraser, T., Molina, J., Arbaugh, W.: Copilot: A Coprocessor-based Kernel Runtime Integrity Monitor. In: Proceedings of the 13th USENIX Security Symposium, pp. 179–194 (2004)
[5] Wilhelm, J., Chiueh, T.-c.: A Forced Sampled Execution Approach to Kernel Rootkit Identification. In: Kruegel, C., Lippmann, R., Clark, A. (eds.) RAID 2007. LNCS, vol. 4637, pp. 219–235. Springer, Heidelberg (2007)
[6] Garfinkel, T., Rosenblum, M.: A Virtual Machine Introspection Based Architecture for Intrusion Detection. In: Proc. Network and Distributed Systems Security Symposium (NDSS 2003) (February 2003)
[7] Seshadri, A., Luk, M., Qu, N., Perrig, A.: SecVisor: A Tiny Hypervisor to Guarantee Lifetime Kernel Code Integrity for Commodity OSes. In: Proceedings of the ACM Symposium on Operating Systems Principles (SOSP 2007) (October 2007)
[8] Bellard, F.: QEMU: A Fast and Portable Dynamic Translator. In: Proceedings of the USENIX Annual Technical Conference, FREENIX Track, pp. 41–46 (2005)
[9] Innotek: Virtualbox (Last accessed, September 2007), http://www.virtualbox.org/
[10] Intel: Vanderpool Technology (2005), http://www.intel.com/technology/computing/vptech
[11] AMD: AMD64 Architecture Programmer's Manual Volume 2: System Programming, 3.12 edition (September 2006)
[12] Dunlap, G., King, S., Cinar, S., Basrai, M., Chen, P.: ReVirt: Enabling Intrusion Analysis through Virtual Machine Logging and Replay. In: Proc. USENIX Symposium on Operating Systems Design and Implementation (OSDI 2002) (2002)
[13] Garfinkel, T., Pfaff, B., Chow, J., Rosenblum, M., Boneh, D.: Terra: A Virtual Machine-Based Platform for Trusted Computing. In: Proc. of ACM Symposium on Operating System Principles (SOSP 2003) (October 2003)
[14] Jiang, X., Wang, X.: "Out-of-the-Box" Monitoring of VM-Based High-Interaction Honeypots. In: Kruegel, C., Lippmann, R., Clark, A. (eds.) RAID 2007. LNCS, vol. 4637, pp. 198–218. Springer, Heidelberg (2007)
[15] Joshi, A., King, S., Dunlap, G., Chen, P.: Detecting Past and Present Intrusions through Vulnerability-specific Predicates. In: Proc. ACM Symposium on Operating Systems Principles (SOSP 2005), pp. 91–104 (2005)
[16] Riley, R., Jiang, X., Xu, D.: Guest-Transparent Prevention of Kernel Rootkits with VMM-based Memory Shadowing. Technical report CERIAS TR 2001-146, Purdue University
[17] Arbaugh, W.A., Farber, D.J., Smith, J.M.: A Secure and Reliable Bootstrap Architecture. In: Proceedings of IEEE Symposium on Security and Privacy, May 1997, pp. 65–71 (1997)
[18] sd, devik: Linux on-the-fly Kernel Patching without LKM. Phrack 11(58) Article 7
[19] fuzen_op: Fu rootkit (Last accessed, September 2007), http://www.rootkit.com/project.php?id=12

[20] Shacham, H.: The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86). In: Proceedings of the ACM Conference on Computer and Communications Security (CCS 2007) (October 2007)
[21] Chen, S., Xu, J., Sezer, E.C., Gauriar, P., Iyer, R.: Non-Control-Data Attacks Are Realistic Threats. In: Proceedings of the 14th USENIX Security Symposium (August 2005)
[22] Baliga, A., Kamat, P., Iftode, L.: Lurking in the Shadows: Identifying Systemic Threats to Kernel Data. In: Proc. of IEEE Symposium on Security and Privacy (Oakland 2007) (May 2007)
[23] Abadi, M., Budiu, M., Erlingsson, U., Ligatti, J.: Control Flow Integrity: Principles, Implementations, and Applications. In: Proc. ACM Conference on Computer and Communications Security (CCS 2005) (November 2005)
[24] Grizzard, J.B.: Towards Self-Healing Systems: Re-establishing Trust in Compromised Systems. Ph.D. Thesis, Georgia Institute of Technology (May 2006)
[25] Castro, M., Costa, M., Harris, T.: Securing Software by Enforcing Data-Flow Integrity. In: Proc. of USENIX Symposium on Operating Systems Design and Implementation (OSDI 2006) (2006)
[26] Klein, T.: Scooby Doo - VMware Fingerprint Suite (2003), http://www.trapkit.de/research/vmm/scoopydoo/index.html
[27] Rutkowska, J.: Red Pill: Detect VMM Using (Almost) One CPU Instruction (November 2004), http://invisiblethings.org/papers/redpill.html
[28] F-Secure Corporation: Agobot, http://www.f-secure.com/v-descs/agobot.shtml
[29] Kortchinsky, K.: Honeypots: Counter Measures to VMware Fingerprinting (January 2004), http://seclists.org/lists/honeypots/2004/Jan-Mar/0015.html
[30] Liston, T., Skoudis, E.: On the Cutting Edge: Thwarting Virtual Machine Detection (2006), http://handlers.sans.org/tliston/ThwartingVMDetection_Liston_Skoudis.pdf
[31] 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: Proc. of the 13th Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2008) (March 2008)
[32] Microsoft Corporation: Driver Signing for Windows, http://www.microsoft.com/resources/documentation/windows/xp/all/proddocs/en-us/code_signing.mspx?mfr=true
[33] Kruegel, C., Robertson, W., Vigna, G.: Detecting Kernel-Level Rootkits Through Binary Analysis. In: Yew, P.-C., Xue, J. (eds.) ACSAC 2004. LNCS, vol. 3189, pp. 91–100. Springer, Heidelberg (2004)
[34] Zhang, X., van Doorn, L., Jaeger, T., Perez, R., Sailer, R.: Secure Coprocessor-based Intrusion Detection. In: Proceedings of the 10th ACM SIGOPS European Workshop, pp. 239–242 (2002)
[35] Wang, Y.M., Beck, D., Vo, B., Roussev, R., Verbowski, C.: Detecting Stealth Software with Strider GhostBuster. In: Proc. IEEE International Conference on Dependable Systems and Networks (DSN 2005), pp. 368–377 (2005)
[36] Kennell, R., Jamieson, L.H.: Establishing the Genuinity of Remote Computer Systems. In: Proc. of the 12th USENIX Security Symposium (August 2003)
[37] Sailer, R., Jaeger, T., Zhang, X., van Doorn, L.: Attestation-based Policy Enforcement for Remote Access. In: Proc. of ACM Conference on Computer and Communications Security (CCS 2004) (October 2004)
[38] Sailer, R., Zhang, X., Jaeger, T., van Doorn, L.: Design and Implementation of a TCG-based Integrity Measurement Architecture. In: Proc. of the 13th USENIX Security Symposium (August 2004)

Countering Persistent Kernel Rootkits through Systematic Hook Discovery

Zhi Wang[1], Xuxian Jiang[1], Weidong Cui[2], and Xinyuan Wang[3]

[1] North Carolina State University
[2] Microsoft Research
[3] George Mason University

Abstract. Kernel rootkits, as one of the most elusive types of malware, pose significant challenges for investigation and defense. Among the most notable are *persistent kernel rootkits*, a special type of kernel rootkits that implant persistent kernel hooks to tamper with the kernel execution to hide their presence. To defend against them, an effective approach is to first identify those kernel hooks and then protect them from being manipulated by these rootkits. In this paper, we focus on the first step by proposing a systematic approach to identify those kernel hooks. Our approach is based on two key observations: First, rootkits by design will attempt to hide its presence from *all* running rootkit-detection software including various system utility programs (e.g., *ps* and *ls*). Second, to manipulate OS kernel control-flows, persistent kernel rootkits by their nature will implant kernel hooks on the corresponding kernel-side execution paths invoked by the security programs. In other words, for any persistent kernel rootkit, either it is detectable by a security program or it has to tamper with one of the kernel hooks on the corresponding kernel-side execution path(s) of the security program. As a result, given an authentic security program, we *only* need to monitor and analyze its kernel-side execution paths to identify the related set of kernel hooks that could be potentially hijacked for evasion. We have built a proof-of-concept system called HookMap and evaluated it with a number of Linux utility programs such as *ls*, *ps*, and *netstat* in RedHat Fedora Core 5. Our system found that there exist 35 kernel hooks in the kernel-side execution path of *ls* that can be potentially hijacked for manipulation (e.g., for hiding files). Similarly, there are 85 kernel hooks for *ps* and 51 kernel hooks for *netstat*, which can be respectively hooked for hiding processes and network activities. A manual analysis of eight real-world rootkits shows that our identified kernel hooks cover all those used in them.

1 Introduction

Rootkits have been increasingly adopted by general malware or intruders to hide their presence on or prolong their control of compromised machines. In particular, kernel rootkits, with the unique capability of directly subverting the victim operating system (OS) kernel, have been frequently leveraged to expand the basic OS functionalities with additional (illicit) ones, such as providing unauthorized system backdoor access, gathering personal information (e.g., user keystrokes), escalating the privilege of a malicious process, as well as neutralizing defense mechanisms on the target system.

In this paper, we focus on a special type of kernel rootkits called *persistent kernel rootkits*. Instead of referring to those rootkits that are stored as persistent disk files and will survive machine reboots, the notion of persistent kernel rootkits here (inherited from [14]) represents those rootkits that will make persistent modifications to run-time OS kernel control-flow, so that normal kernel execution will be somehow hijacked to provide illicit rootkit functionality[1]. For example, many existing rootkits [1,2] will modify the system call table to hijack the kernel-level control flow. This type of rootkits is of special interest to us for a number of reasons. First, a recent survey [14] of both Windows and Linux kernel rootkits shows that 96% of them are persistent kernel rootkits and they will make persistent control-flow modifications. Second, by running inside the OS kernel, these rootkits have the highest privilege on the system, making them very hard to be detected or removed. In fact, a recent report [3] shows that, once a system is infected by these rootkits, the best way to recover from them is to re-install the OS image. Third, by directly making control-flow modifications, persistent kernel rootkits provide a convenient way to add a rich set of malicious rootkit functionalities.

On the defensive side, one essential step to effectively defending against persistent kernel rootkits is to identify those hooking points (or kernel hooks) that are used by rootkits to regain kernel execution control and then inflict all sorts of manipulations to cloak their presence. The identification of these kernel hooks is useful for not only understanding the hooking mechanism [23] used by rootkits, but also providing better protection of kernel integrity [10,14,20]. For example, existing anti-rootkit tools such as [8,16,17] all can be benefited because they require the prior knowledge of those kernel hooks to detect the rootkit presence.

To this end, a number of approaches [14,23] have been proposed. For example, SBCFI [14] analyzes the Linux kernel source code and builds an approximation of kernel control-flow graph that will be followed at run-time by a legitimate kernel. Unfortunately, due to the lack of dynamic run-time information, it is only able to achieve an approximation of kernel control-flow graph. From another perspective, HookFinder [23] is developed to automatically analyze a given malware sample and identify those hooks that are being used by the provided malware. More specifically, HookFinder considers any changes made by the malware as tainted and recognizes a specific change as a hooking point if it eventually redirects the execution control to the tainted attack code. Though effective in identifying specific hooks used by the malware, it cannot discover other hooks that can be equally hijacked but are not being used by the malware.

In this paper, we present a systematic approach that, given a rootkit-detection program, discovers those related kernel hooks that could be potentially used by persistent kernel rootkits to evade from it. Our approach is motivated by the following observation: To hide its presence, a persistent kernel rootkit by design will hide from the given security program and the hiding is achieved by implanting kernel hooks in a number of strategic locations within the kernel-side execution paths of the security program. In other words, for any persistent kernel rootkit, either it is detectable by the security program or it has to tamper with one of the kernel hooks. Therefore, for the purpose of

[1] For other types of kernel rootkits that may attack kernel data, they are *not* the focus of this paper and we plan to explore them as future work.

detecting persistent kernel rootkits, it is sufficient to just identify all kernel hooks in the kernel-side execution paths of a given rootkit-detection program.

To identify hooks in the kernel-side execution of a program, we face three main challenges: (1) accurately identifying the right kernel-side execution path for monitoring; (2) obtaining the relevant run-time context information (e.g., the ongoing system call and specific kernel functions) with respect to the identified execution path; (3) uncovering the kernel hooks in the execution path and extracting associated semantic definition. To effectively address the first two challenges, we developed a context-aware kernel execution monitor and the details will be described in Section 3.1. For the third one, we have built a kernel hook identifier (Section 3.2) that will first locate the run-time virtual address of an uncovered kernel hook and then perform OS-aware semantics resolution to reveal a meaningful definition of the related kernel object or variable.

We have developed a prototype called HookMap on top of a software-based QEMU virtual machine implementation [6]. It is appropriate for two main reasons: First, software-based virtualization allows to conveniently support commodity OSes as guest virtual machines (VMs). And more importantly, given a selected execution path, the virtualization layer can be extended to provide the unique capability in instrumenting and recording its execution without affecting its functionality. Second, since we are dealing with a legitimate OS kernel in a clean system, not with a rootkit sample that may detect the VM environment and alter its behavior accordingly, the use of virtualization software will not affect the results in identifying kernel hooks.

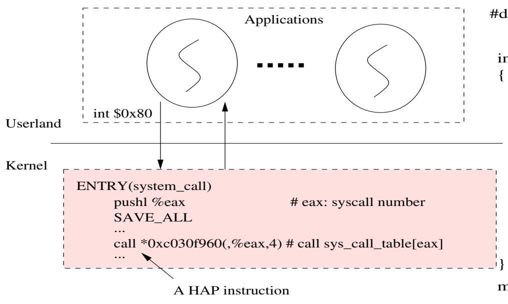
To evaluate the effectiveness of our approach, we ran a default installation of RedHat Fedora Core 5 (with Linux kernel 2.6.15) in our system. Instead of using any commercial rootkit-detection software, we chose to test with three utility programs, *ls*, *ps* and *netstat* since they are often attacked by rootkits to hide files, processes or network connections. By monitoring their kernel-side executions, our system was able to accurately identify their execution contexts, discover all encountered kernel hooks, and then resolve their semantic definitions. In particular, our system identified 35, 85, and 51 kernel hooks, for *ls*, *ps* and *netstat*, respectively. To empirically evaluate the completeness of identified kernel hooks, we performed a manual analysis of eight real-world kernel rootkits and found that the kernel hooks employed by these rootkits are only a small subset of our identified hooks.

The rest of the paper is structured as follows: Section 2 introduces the background on rootkit hooking mechanisms. Section 3 gives an overview of our approach, followed by the description of HookMap implementation in Section 4. Section 5 presents the experimental results and Section 6 discusses some limitations of the proposed approach. Finally, Section 7 surveys related work and Section 8 concludes the paper.

2 Background

In this section, we introduce the hooking mechanisms that are being used by persistent kernel rootkits and define a number of terms that will be used throughout the paper.

There exist two main types of kernel hooks: *code hooks* and *data hooks*. To implant a code hook, a kernel rootkit typically modifies the kernel text so that the execution of the affected text will be directly hijacked. However, since the kernel text section is



```
ENTRY(system_call)
      pushl %eax                  # eax: syscall number
      SAVE_ALL
      ...
      call *0xc030f960(,%eax,4) # call sys_call_table[eax]
      ...
```

(a) The system call dispatcher on Linux

```
#define REPLACE(x) o_##x = sys_call_table[__NR_##x];\
               sys_call_table[__NR_##x] = n_##x

int adore_init(void)
{
          ...
          REPLACE(write);
          REPLACE(getdents);
          REPLACE(kill);
          REPLACE(fork);
          REPLACE(clone);
          REPLACE(close);
          REPLACE(open);
          REPLACE(stat);
          REPLACE(lstat);
          ...
}
module_init(adore_init);
```

(b) The Linux adore rootkit

Fig. 1. A HAP instruction example inside the Linux system call dispatcher – the associated kernel data hooks have been attacked by various rootkits, including the Linux adore rootkit [1]

usually static and can be marked as read-only (or not writable), the way to implant the code hook can be easily detected. Because of that, rootkit authors are now more inclined to implant data hooks at a number of strategic memory locations in the kernel space. Data hooks are usually a part of kernel data that are interpreted as the destination addresses in control-flow transition instructions such as *call* and *jmp*. A typical example of kernel data hook is the system call table that contains the addresses to a number of specific system call service routines (e.g., *sys_open*). In addition, many data hooks may contain dynamic content as they are mainly used to hold the run-time addresses of kernel functions and can be later updated because of the loading or unloading of kernel modules. For ease of presentation, we refer to the control-flow transition instructions (i.e., *call* or conditional or un-conditional jumps) whose destination addresses are not hard-coded constants as *hook attach points* (HAPs).

In Figure 1, we show an HAP example with associated kernel data hooks, i.e., the system call table, which is commonly attacked by kernel rootkits. In particular, Figure 1(a) shows the normal system call dispatcher on Linux while Figure 1(b) contains the code snippet of a Linux rootkit – adore [1]. From the control-flow transfer instruction – *call *0xc030f960(,%eax,4)*[2] in Figure 1(a), we can tell the existence of a hook attach point inside the system call dispatcher. In addition, Figure 1(b) reveals that the adore rootkit will replace a number of system call table entries (as data hooks) so that it can intervene and manipulate the execution of those replaced system calls. For instance, the code statement *REPLACE(write)* rewrites the system call table entry *sys_call_table[4]* to intercept the *sys_write* routine before its execution. The corresponding run-time memory location $0xc030f970$ and the associated semantic definition of *sys_call_table[4]* will be identified as a data hook. More specifically, the memory location $0xc030f970$ is calculated as $0xc030f960 + \%eax \times 4$ where $0xc030f960$ is the base address of system call table and $\%eax = 4$ is the actual number for the specific *sys_write* system call. We defer an in-depth analysis of this particular rootkit in Section 5.2.

[2] This instruction is in the standard AT&T assembly syntax, meaning that it will transfer its execution to another memory location pointed to by $0xc030f960 + \%eax \times 4$.

Meanwhile, as mentioned earlier, there are a number of rootkits that will replace specific instructions (as code hooks) in the system call handler. For instance, the SucKit [19] rootkit will prepare its own version of the system call table and then change the dispatcher so that it will invoke system call routines populated in its own system call table. Using Figure 1(a) as an example, the rootkit will modify the control-flow transfer instruction or more specifically the base address of the system call table $0xc030f960$ to point to a rootkit-controlled system call table. Considering that (1) implanting a code hook will inflict kernel code modifications, which can be easily detected, and (2) every kernel instruction could be potentially overwritten for code hook purposes, we in this paper focus on the identification of kernel data hooks. Without ambiguity, we use the term kernel hooks to represent kernel data hooks throughout the paper.

Finally, we point out that kernel hooks are elusive to identify because they can be widely scattered across the kernel space and rootkit authors keep surprising us in using new kernel hooks for rootkit purposes [7,18]. In fact, recent research results [23] show that some stealth rootkits use previously unknown kernel hooks to evade all existing security programs for rootkit detection. In this paper, our goal is to systematically discover all kernel hooks that can be used by persistent kernel rootkits to tamper with and thus hide from a given security program.

3 System Design

The intuition behind our approach is straightforward but effective: a rootkit by nature is programmed to hide itself especially from various security programs including those widely-used system utility programs such as *ps*, *ls*, and *netstat*. As such for an infected OS kernel, the provided kernel service (e.g., handling a particular system call) to any request from these security software is likely manipulated. The manipulation typically comes from the installation of kernel hooks at strategic locations somewhere *within* the corresponding kernel-side execution path of these security software. Based on this insight, if we can develop a system to comprehensively monitor the kernel-side execution of the same set of security programs within a clean system, we can use it to exhaustively uncover all kernel hooks related to the execution path being monitored. Figure 2 shows an architectural overview of our system with two main components: *context-aware execution monitor* and *kernel hook identifier*. In the following, we will describe these two components in detail.

3.1 Context-Aware Execution Monitor

As mentioned earlier, our system is built on top of an open-source virtual machine implementation, which brings the convenient support of commodity OSes as guest VMs. In addition, for a running VM, the *context-aware execution monitor* is further designed to monitor the internal process events including various system calls made by running processes. As suggested by the aforementioned insight, we need to only capture those kernel events related to security software that is running inside the VM. Note that the main purpose of monitoring these events is to understand the right execution context inside the kernel (e.g., “which process is making the system call?”). With that, we can

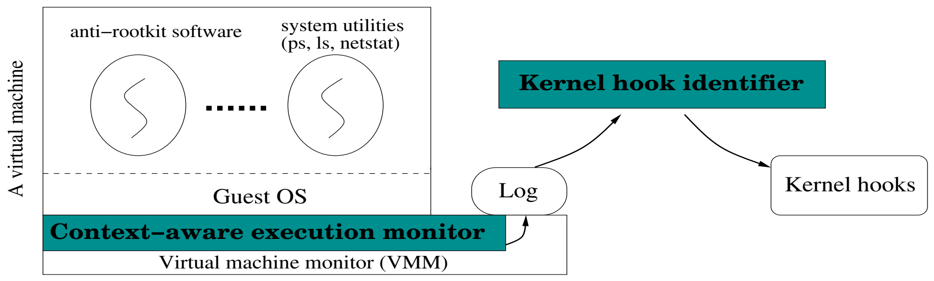

anti–rootkit software
system utilities (ps, ls, netstat)
A virtual machine
Guest OS
Context–aware execution monitor
Virtual machine monitor (VMM)
Log
Kernel hook identifier
Kernel hooks

Fig. 2. A systematic approach to discovering kernel hooks

then accurately instrument and record all executed kernel instructions that are relevant to the chosen security software.

However, a challenging part is that modern OS kernels greatly complicate the capture and interpretation of execution contexts with the introduction of "out of order" execution (mainly for improving system concurrency and performance reasons). The "out of order" execution means that the kernel-side execution of any process can be asynchronously interrupted to handle an incoming interrupt request or temporarily context-switched out for the execution of another unrelated process. Notice that the "out of order" execution is considered essential in modern OSes for the support of multi-tasking and asynchronous interrupt handling.

Fortunately, running a commodity OS as a guest VM provides a convenient way to capture those external events[3] that trigger the "out of order" executions in a guest kernel. For example, if an incoming network packet leads to the generation of an interrupt, the interrupt event needs to be emulated by the underlying virtual machine monitor and thus can be intercepted and recorded by our system. The tricky part is to determine when the corresponding interrupt handler ends. For that purpose, we instrument the execution of *iret* instruction to trace when the interrupt handler returns. However, additional complexities are introduced for the built-in support of *nested interrupts* in the modern OS design where an interrupt request (IRQ) of a higher priority is allowed to preempt IRQs of a lower priority. For that, we need to maintain a shadow interrupt stack to track the nested level of an interrupt.

In addition to those external events, the "out of order" execution can also be introduced by some internal events. For example, a running process may voluntarily yield the CPU execution to another process. For that, instead of locating and intercepting all these internal events, we need to take another approach by directly intercepting context switch events occurred inside the monitored VM. The interception of context switch events requires some knowledge of the OS internals. We will describe it in more details in Section 4.

With the above capabilities, we can choose and run a particular security program (or any rootkit-detection tool) inside the monitor. The monitor will record into a local trace

[3] Note that the external events here may also include potential debug exceptions caused from hardware-based debugger registers. However, in this work, we do not count those related hooks within the debug interrupt handler.

file a stream of system calls made by the chosen program and for each system call, a sequence of kernel instructions executed within the system call execution path.

3.2 Kernel Hook Identifier

The context-aware execution monitor will collect a list of kernel instructions that are sequentially executed when handling a system call request from a chosen security program. Given the collected instructions, the kernel hook identifier component is developed to identify those HAPs where kernel hooks are involved. The identification of potential HAPs is relatively straightforward because they are the control-flow transfer instructions, namely those *call* or *jmp* instructions.

Some astute readers may wonder "wouldn't static analysis work for the very same need?" By statically analyzing kernel code, it is indeed capable of identifying those HAPs. Unfortunately, it cannot lead to the identification of the corresponding kernel hooks. There are two main reasons: (1) A HAP may use registers or memory locations to resolve the run-time locations of the related kernel hooks. In other words, the corresponding kernel hook location cannot be determined through static analysis. (An example is already shown in Figure 1(a).) (2) Moreover, there exists another complexity that is introduced by the loadable kernel module (LKM) support in commodity OS kernels. In particular, when a LKM is loaded into the kernel, not only its loading location may be different from previous runs, but also the module text content will be updated accordingly during the time when the module is being loaded. This is mainly due to the existence of certain dependencies of the new loaded module on other loaded modules or the main static kernel text. And we cannot resolve these dependencies until at run-time.

Our analysis shows that for some discovered HAPs, their run-time execution trace can readily reveal the locations of associated kernel hooks. As an example, in the system call dispatcher shown in Figure 1(a), the HAP instruction – *call *0xc030f960(,%eax,4)*, after the execution, will jump to a function which is pointed to from the memory location: $0xc030f960 + \%eax \times 4$, where the value of *%eax* register can be known at run-time. In other words, the result of the calculation at run-time will be counted as a kernel hook in the related execution path. In addition, there also exist other HAPs (e.g., *call *%edx*) that may directly call registers and reveal nothing about kernel hooks but the destination addresses the execution will transfer to. For that, we need to start from the identified HAP and examine in a backwards manner those related instructions to identify the source, which eventually affects the calculated destination value and will then be considered a kernel hook. (The detailed discussion will be presented in Section 4.2.) In our analysis, we also encounter some control-flow transfer instructions whose destination addresses are hardcoded or statically linked inside machine code. In this case, both static analysis and dynamic analysis can be used to identify the corresponding hooks. Note that according to the nature of this type of hooks (Section 2), we consider them as code hooks in this paper.

Finally, after identifying those kernel hooks, we also aim to resolve the memory addresses to the corresponding semantic definitions. For that, we leverage the symbol information available in the raw kernel text file as well as loaded LKMs. More specifically, for main kernel text, we obtain the corresponding symbol information (e.g., object names and related memory locations) from the related *System.map* file. For kernel modules, we derive the corresponding symbol information from the object files (e.g., by running the *nm* command)[4]. If we use Figure 1(a) as an example, in an execution path related to the *sys_open* routine, the hook's memory address is calculated as *0xc030f974*. From the symbol information associated with the main kernel text, that memory address is occupied by the system call table (with the symbol name *sys_call_table*) whose base address is *0xc030f960*. As a result, the corresponding kernel hook is resolved as *sys_call_table[5]*[5] where 5 is actually the system call number for the *sys_open* routine.

4 Implementation

We have built a prototype system called HookMap based on an open-source QEMU 0.9.0 [6] virtual machine monitor (VMM) implementation. As mentioned earlier, we choose it due to the following considerations: (1) First, since we are dealing with normal OS kernels, the VM environment will not affect the results in the identified kernel hooks; (2) Second, it contains the implementation of a key virtualization technique called dynamic binary translation [6,4], which can be leveraged and extended to select, record, and disassemble kernel instruction sequences of interest; (3) Third, upon the observation of VM-internal process events, we need to embed our own interpretation logic to extract related execution context information. The open-source nature of the VM implementation provides great convenience and flexibility in making our implementation possible. Also, due to the need of obtaining run-time symbols for semantic resolution, our current system only supports Linux. Nevertheless, we point out that the principle described here should also be applicable for other software-based VM implementations (e.g., VMware Workstation [4]) and other commodity OSes (e.g., Windows).

4.1 Context-Aware Execution Logging

One main task in our implementation is that, given an executing kernel instruction, we need to accurately understand the current execution context so that we can determine whether the instruction should be monitored and recorded. Note that the execution context here is defined as the system call context the current (kernel) instruction belongs to. To achieve that, we have the need of keeping track of the lifetime of a system call event. Fortunately, the lifetime of a system call event is well defined as the kernel accepts only two standard methods in requesting for a system call service: *int $0x80* and *sysenter*. Since we are running the whole system on top of a binary-translation-capable VMM, we can conveniently intercept these two instructions and then interpret the associated system call arguments accordingly. For this specific task, we leverage an "out-of-the-box" VM monitoring framework called VMscope [11] as it already allows

[4] We point out that the *nm* command output will be further updated with the run-time loading address of the corresponding module. For that, we will instrument the module-loading instructions in the kernel to determine the address at run-time.

[5] The calculation is based on the following: $(0xc030f974 - 0xc030f960)/4 = 5$, where 4 represents the number of bytes occupied by a function pointer.

to real-time capture system calls completely outside the VM. What remains to do is to correlate a system call event and the related system call return event to form its lifetime. Interested readers are referred to [11] for more details.

Meanwhile, we also face another challenge caused by the "out-of-order" execution (Section 3). To address that, we monitor relevant external events (e.g., interrupts) as well as internal events (e.g., context switches) to detect run-time changes of the execution context. The main goal here is to avoid the introduction of "noises" – unnecessary kernel executions – into the execution path for monitoring and analysis. Fortunately, with a software-based VM implementation, we are able to intercept all these external events as they need to be eventually emulated by the underlying VMM. However, an interesting part is to handle the nested interrupts scenario where a shadow interrupt stack should be maintained at the VMM layer to keep track of the nested level of the ongoing interrupt. For the internal events, our prototype sets a breakpoint on a kernel function that actually performs context-switching. On Linux, the related function is called *__switch_to* and its location is exported by kernel and can be found in the *System.map* file[6].

With the above capabilities, our system essentially organizes the kernel instruction execution into a stream of system calls and each system call contains a sequence of kernel instructions executed within this specific context. Furthermore, to facilitate later identification and analysis of kernel hooks, for each kernel instruction in one particular context, we further dump the memory locations as well as registers, if any, involved in this instruction. The additional information is needed for later kernel hook identification, which we describe next.

4.2 Kernel Hook Identification

Based on the collected sequence of kernel instructions, the kernel hook identifier locates and analyzes those control-flow transfer *call* or *jmp* instructions (as HAP instructions) to uncover relevant kernel hooks. As a concrete example, we show in Table 1 a list of identified HAPs, associated system call contexts, as well as those kernel hooks that are obtained by monitoring kernel-side execution of the *ls* command. Note that a (small) subset of those identified kernel hooks have already been used by rootkits for file-hiding purposes (more in Section 5).

As mentioned earlier, for an HAP instruction that will read a memory location and jump to the function pointed by a memory location, we can simply record the memory location as a kernel hook. However, if an HAP instruction directly calls a register (e.g., *call *%edx*), we need to develop an effective scheme to trace back to the source – a kernel hook that determines the value of the register.

We point out that this particular problem is similar to the classic problem addressed by dynamic program slicing [5,24]: Given an execution history and a variable as the input, the goal of dynamic program slicing is to extract a slice that contains all the instructions in the execution history that affected the value of that variable. As such, for the register involved in an identified HAP instruction, we apply the classic dynamic

[6] A different version of *ls* can result in the execution of *sys_getdents64* instead of *sys_getdents*, which leads to one variation in the identified kernel hooks – *sys_call_table[220]* instead of *sys_call_table[141]*. A similar scenario also happens when identifying another set of kernel hooks by monitoring the *ps* command (to be shown in Table 2).

Table 1. File-hiding kernel hooks obtained by monitoring the *ls -alR /* command in RedHat Fedora Core 5

execution path	#	Hook Attach Points (HAPs)		Kernel Hooks
		address	instruction	address
sys_write	1	0xc0102b38	call *0xc030f960(,%eax,4)	sys_call_table[4]
	2	0xc014e5a3	call *0xec(%ecx)	selinux_ops[59]
	3	0xc014e5c9	call *%edi	tty_fops[4]
	4	0xc01c63c6	jmp *0xc02bfb40(,%eax,4)	dummy_con[33]
	5	0xc01fa9d2	call *0xc(%esp)	tty_ldisc_N_TTY.write_chan
	6	0xc01fd4f5	call *0xc8(%ecx)	con_ops[3]
	7	0xc01fd51e	call *0xd0(%edx)	con_ops[5]
	8	0xc01fd5fa	call *%edx	con_ops[4]
	9	0xc01fd605	call *0xc4(%ebx)	con_ops[2]
	10	0xc0204caa	call *0x1c(%ecx)	vga_con[7]
sys_open	1	0xc0102b38	call *0xc030f960(,%eax,4)	sys_call_table[5]
	2	0xc014f024	call *0xf0(%edx)	selinux_ops[60]
	3	0xc0159677	call *%esi	ext3_dir_inode_operations[13] (ext3.ko)
	4	0xc015969d	call *0xbc(%ebx)	selinux_ops[47]
	5	0xc019ea96	call *0xbc(%ebx)	capability_ops[47]
sys_close	1	0xc0102b38	call *0xc030f960(,%eax,4)	sys_call_table[6]
	2	0xc014f190	call *%ecx	ext3_dir_operations[14] (ext3.ko)
	3	0xc014f19a	call *0xf4(%edx)	selinux_ops[61]
sys_ioctl	1	0xc0102b38	call *0xc030f960(,%eax,4)	sys_call_table[54]
	2	0xc015dbcf	call *%esi	tty_fops[8]
	3	0xc015de16	call *0xf8(%ebx)	selinux_ops[62]
	4	0xc01fc5a1	call *%ebx	con_ops[7]
	5	0xc01fc5c9	call *%ebx	tty_ldisc_N_TTY.n_tty_ioctl
sys_mmap2	1	0xc0102b38	call *0xc030f960(,%eax,4)	sys_call_table[192]
	2	0xc0143e0e	call *0xfc(%ebx)	selinux_ops[63]
	3	0xc0143ebc	call *0x2c(%edx)	selinux_ops[11]
	4	0xc0144460	call *%esi	mm→get_unmapped_area
	5	0xc019dc50	call *0x18(%ecx)	capability_ops[6]
	6	0xc019f5d5	call *0xfc(%ebx)	capability_ops[63]
sys_fstat64	1	0xc0102b38	call *0xc030f960(,%eax,4)	sys_call_table[197]
	2	0xc0155f33	call *0xc4(%ecx)	selinux_ops[49]
sys_getdents[6]	1	0xc0102b38	call *0xc030f960(,%eax,4)	sys_call_table[114]
	2	0xc015de80	call *0xec(%ecx)	selinux_ops[59]
	3	0xc015decc	call *0x18(%ebx)	ext3_dir_operations[6] (ext3.ko)
	4	0xc016b711	call *%edx	ext3_dir_inode_operations[3] (ext3.ko)
sys_getdents64	1	0xc0102b38	call *0xc030f960(,%eax,4)	sys_call_table[220]
	2	0xc015de80	call *0xec(%ecx)	selinux_ops[59]
	3	0xc015decc	call *0x18(%ebx)	ext3_dir_operations[6] (ext3.ko)
	4	0xc016b711	call *%edx	ext3_dir_inode_operations[3] (ext3.ko)
sys_fcntl64	1	0xc0102b38	call *0xc030f960(,%eax,4)	sys_call_table[221]
	2	0xc015d7a7	call *0x108(%ebx)	selinux_ops[66]

program slicing algorithm [5] to find out a memory location that is associated with a kernel object (including a global static variable) and whose content determines the register value. To do that, we follow the algorithm by first computing two sets for

```
#line machine code        instruction                DEF        USE
====  =============     ====================     ========     =====
i-1 : ...
i+0 : 89 c3               mov    %eax,%ebx          %ebx         %eax
i+1 : 83 ec 04            sub    $0x4,%esp          %esp         %esp
i+2 : 8b 80 c4 00 00 00   mov    0xc4(%eax),%eax    %eax         mem[%eax+0xc4], %eax
i+3 : f6 c2 03            test   $0x3,%dl           eflags       %dl
i+4 : 89 04 24            mov    %eax,(%esp)        mem[esp]     %eax
i+5 : 74 0e               je     c016b713                        eflags
i+6 : 8b 40 24            mov    0x24(%eax),%eax    %eax         mem[%eax+0x24], %eax
i+7 : 8b 50 0c            mov    0xc(%eax),%edx     %edx         mem[%eax+0xc],  %eax
i+8 : 85 d2               test   %edx,%edx          eflags       %edx
i+9 : 74 04               je     c016b713                        eflags
i+10: 89 d8               mov    %ebx,%eax          %eax         %ebx
i+11: ff d2               call   *%edx              %eip         %edx
i+12: ...
```

Fig. 3. Discovering a kernel hook based on dynamic program slicing

each related instruction: one is $DEF[i]$ that contains the variable(s) defined by this instruction, and another is $USE[i]$ that includes all variables used by this instruction. Each set can contain an element of either a memory location or a machine register. After that, we then examine backwards to find out the memory location that is occupied by a kernel object and whose content determines the register value. In the following, we will walk-through the scheme with an example. (For the classic algorithm, interested readers are referred to [5] for more details.)

Figure 3 shows some sequential kernel instructions[7] of a kernel function *__mark_inode_dirty* that are executed in the *sys_getdent64* context of the *ls* command. In particular, the sequence contains an HAP instruction – *call *%edx* – at the memory location $0xc016b711$ (line $i + 11$ in Figure 3). Note that since we monitor at run-time, we can precisely tell which memory locations/registers are defined and/or used. As a result, we directly derive the corresponding destination address (contained in the *%edx* register), which is $0xc885bca0$ – the entry point of a function *ext3_dirty_inode* within a LKM named *ext3.ko*. Obviously, it is the destination address the HAP instruction will transfer to, *not* the relevant kernel hook. Next, our prototype further expands the associated semantics of every executed instruction i to compute the two sets $DEF[i]$ and $USE[i]$ and the results are shown in Figure 3. With the two sets defined for each instruction, we can then apply the dynamic slicing algorithm. Specifically, from the HAP instruction (line $i+11$), the USE set contains the *%edx* register, which is defined by the instruction at line $i + 7$. This particular instruction is associated with a USE set having two members: *%eax* and *mem[%eax+0xc]*. It turns out the *%eax* points to the kernel object *ext3_dir_inode_operations* and $0xc$ is an offset from the kernel object. After identifying the responsible kernel object, the slicing algorithm then outputs *ext3_dir_inode_operations[3]* as the corresponding kernel hook and terminates. In Table 1, this is the fourth kernel hook identified in the *sys_getdent64* context. Note that this particular kernel object is a jump table containing a number of function pointers. The offset $0xc$ indicates that it is the fourth member function in the object as each function pointer is four bytes in size. (The first four member functions in the kernel object are in the offsets of $0x0$, $0x4$, $0x8$, and $0xc$, respectively.)

[7] These instructions are in the AT&T assembly syntax, where source and destination operands, if any, are in the reverse order when compared with the Intel assembly syntax.

5 Evaluation

In this section, we present the evaluation results. In particular, we conduct two sets of experiments. The first set of experiments (Section 5.1) is to monitor the execution of various security programs and identify those kernel hooks that can be potentially hijacked for hiding purposes. The second set of experiments (Section 5.2) is to

Table 2. Process-hiding kernel hooks obtained by monitoring the *ps -ef* command in RedHat Fedora Core 5

execution path	# kernel hooks	Details
sys_read	17	sys_call_table[3], selinux_ops[5], selinux_ops[59], capability_ops[5], kern_table[336], timer_pmtmr[2], proc_info_file_operations[2], proc_file_operations[2], proc_sys_file_operations[2], proc_tty_drivers_operations[2], tty_drivers_op[0], tty_drivers_op[1], tty_drivers_op[2], tty_drivers_op[3], proc_inode.op.proc_read, simple_ones[1].read_proc, simple_ones[2].read_proc
sys_write	11	sys_call_table[4], selinux_ops[59], dummy_con[33], tty_fops[4], con_ops[2], con_ops[3], con_ops[4], con_ops[5], vga_con[6], vga_con[7], tty_ldisc_N_TTY.write_chan
sys_open	20	sys_call_table[5], selinux_ops[34], selinux_ops[46], selinux_ops[47], selinux_ops[60], selinux_ops[88], selinux_ops[112], capability_ops[46], capability_ops[47], pid_base_dentry_operations[0], proc_sops[0], proc_sops[2], proc_root_inode_operations[1], proc_dir_inode_operations[1], proc_self_inode_operations[10], proc_sys_file_operations[12] , proc_tgid_base_inode_operations[1], proc_tty_drivers_operations[12], ext3_dir_inode_operations[13] (ext3.ko), ext3_file_operations[12] (ext3.ko)
sys_close	10	sys_call_table[6], selinux_ops[35], selinux_ops[50], selinux_ops[61], pid_dentry_operations[3], proc_dentry_operations[3], proc_tty_drivers_operations[14], proc_sops[1], proc_sops[6], proc_sops[7]
sys_time	2	sys_call_table[13], timer_pmtmr[2]
sys_lseek	2	sys_call_table[19], proc_file_operations[1]
sys_ioctl	5	sys_call_table[54], tty_fops[8], selinux_ops[62], con_ops[7], tty_ldisc_N_TTY.n_tty_ioctl
sys_mprotect	3	sys_call_table[125], selinux_ops[64], capability_ops[64]
sys_getdents[8]	3	sys_call_table[141], selinux_ops[59], proc_root_operations[6]
sys_getdents64	3	sys_call_table[220], selinux_ops[59], proc_root_operations[6]
sys_mmap2	8	sys_call_table[192], selinux_ops[63], selinux_ops[11], capability_ops[6], capability_ops[63], ext3_dir_inode_operations[3] (ext3.ko), ext3_file_operations[11], mm→get_unmapped_area
sys_stat64	16	sys_call_table[195], selinux_ops[34], selinux_ops[46], selinux_ops[47], selinux_ops[49], selinux_ops[88], selinux_ops[112], capability_ops[46], capability_ops[47], ext3_dir_inode_operations[13] (ext3.ko), pid_base_dentry_operations[0], pid_dentry_operations[3], proc_root_inode_operations[1], proc_self_inode_operations[10], proc_sops[0], proc_tgid_base_inode_operations[1]
sys_fstat64	2	sys_call_table[197], selinux_ops[49]
sys_geteuid32	1	sys_call_table[201]
sys_fcntl64	2	sys_call_table[221], selinux_ops[66]

Table 3. Network-hiding kernel hooks obtained by monitoring the *netstat -atp* command in RedHat Fedora Core 5

execution path	# kernel hooks	Details
sys_read	8	sys_call_table[3], selinux_ops[59], seq_ops.start, seq_ops.show, seq_ops.next, seq_ops.stop, proc_tty_drivers_operations[2]
sys_write	12	sys_call_table[4], selinux_ops[59], dummy_con[33], con_ops[2], con_ops[3], con_ops[4], con_ops[5], tty_fops[4], tty_ldisc_N_TTY.write_chan, vga_con[6], vga_con[7], vga_ops[8]
sys_open	19	sys_call_table[5], selinux_ops[34], selinux_ops[35], selinux_ops[47], selinux_ops[50], selinux_ops[60], selinux_ops[61], selinux_ops[112], capability_ops[47], ext3_dir_inode_operations[13] (ext3.ko), pid_dentry_operations[3], proc_root_inode_operations[1], proc_dir_inode_operations[1], proc_sops[0], proc_sops[1], proc_sops[2], proc_sops[6], proc_sops[7], tcp4_seq_fops[12]
sys_close	9	sys_call_table[6], selinux_ops[35], selinux_ops[50], selinux_ops[61], proc_dentry_operations[3], proc_tty_drivers_operations[14], proc_sops[1], proc_sops[6], proc_sops[7],
sys_munmap	2	sys_call_table[91], mm→unmap_area
sys_mmap2	6	sys_call_table[192], selinux_ops[11], selinux_ops[63], capability_ops[6], capability_ops[63], mm→get_unmapped_area
sys_fstat64	2	sys_call_table[197], selinux_ops[49]

empirically evaluate those identified hooks by analyzing a number of real-world rootkits and see whether the used kernel hooks are actually a part of the discovered ones[8].

5.1 Kernel Hooks

In our experiments, we focus on three types of resources that are mainly targeted by rootkits: files, processes, and network connections. To enumerate related kernel hooks, we correspondingly chose three different utility programs – *ls*, *ps*, and *netstat*. These three programs are from the default installation of Red Hat Linux Fedora Core 5 that runs as a guest VM (with $512MB$ memory) on top of our system. Our testing platform was a modest system, a Dell PowerEdge 2950 server with Xeon 3.16Ghz and 4GB memory running Scientific Linux 4.4. As mentioned earlier, the way to choose these programs is based on the intuition that to hide a file (, a process, or a network connection), a persistent kernel rootkit needs to compromise the kernel-side execution of the *ls* (, *ps*, or *netstat*) program.

In our evaluation, we focus on those portions of collected traces that are related to the normal functionality of the security program (e.g., the querying of system states of interest as well as the final result output) and exclude other unrelated ones. For example, if some traces are part of the loading routine that prepares the process memory layout,

[8] Different versions of *ps* invokes different system calls to list files under a directory. In our evaluation, the 3.2.7 version of ps uses the *sys_getdents* system call while the version 3.2.3 uses another system call – *sys_getdents64*. Both system calls work the same way except one has a kernel hook *sys_call_table[141]* while another has *sys_call_table[220]*.

we consider them not related to the normal functionality of the chosen program and thus simply ignore them. Further, we assume that the chosen security program as well as those dependent libraries are not compromised. Tables 1, 2, and 3 contain our results, including those specific execution contexts of related system calls. Encouragingly, for each encountered HAP instruction, we can always locate the corresponding kernel hook and our manual analysis on Linux kernel source code further confirms that each identified kernel hook is indeed from a meaningful kernel object or data structure.

More specifically, these three tables show that most identified kernel hooks are part of jump tables defined in various kernel objects. In particular, there are three main kernel objects containing a large collection of function pointers that can be hooked for hiding purposes: the system call table *sys_call_table*, the SELinux-related security operations table *selinux_ops*, as well as the capability-based operations table *capability_ops*. There are other kernel hooks that belong to a particular dynamic kernel object. One example is the function pointer *get_unmapped_area* (in the *sys_mmap2* execution path of Table 2) inside the *mm* kernel object that manages the process memory layout. Note that this particular kernel hook cannot be determined by static analysis.

More in-depth analysis also reveals that an HAP instruction executed in different execution contexts can be associated with different kernel hooks. One example is the HAP instruction located in the system call dispatcher (Figure 1(a)) where around 300 system call service routines are called by the same HAP instruction. A kernel hook can also be associated with multiple HAP instructions. This is possible because a function pointer (contained in a kernel hook) can be invoked at multiple locations in a function. One such example is *selinux_ops[47]*, a kernel hook that is invoked a number of times in the *sys_open* execution context of the *ps* command. In addition, we observed many one-to-one mappings between an HAP instruction and its associated kernel hook. Understanding the relationship between HAP instructions and kernel hooks is valuable for real-time accurate enforcement of kernel control-flow integrity [14].

5.2 Case Studies

To empirically evaluate those identified kernel rootkits, we manually analyzed the source code of eight real-world Linux rootkits (Table 4). For each rootkit, we first identified what kernel hooks are hijacked to implement a certain hiding feature and then checked whether they are a part of the results shown in Tables 1, 2, and 3. It is encouraging that for every identified kernel hook[9], there always exists an exact match in our results. In the following, we explain two rootkit experiments in detail:

The *Adore* Rootkit. This rootkit is distributed in the form of a loadable kernel module. If activated, the rootkit will implant 15 kernel hooks in the system call table by replacing them with its own implementations. Among these 15 hooks, only three of them are responsible for hiding purposes[10]. More specifically, two system call table en-

[9] Our evaluation focuses on those kernel data hooks. As mentioned earlier, for kernel code hooks, they can be scattered over every kernel instruction in the corresponding system call execution path.

[10] The other 12 hooks are mainly used to provide hidden backdoor accesses. One example is the sys_call_table[6] (sys_close), which is hooked to allow the attacker to escalate the privilege to root without going through the normal authorization process.

Table 4. Kernel hooks used by real-world rootkits (‡ means a code hook)

rootkit	**kernel hooks based on the hiding features**		
	file-hiding	process-hiding	network-hiding
adore	sys_call_table[141] sys_call_table[220]	sys_call_table[141] sys_call_table[220]	sys_call_table[4]
adore-ng	ext3_dir_operations[6]	proc_root_operations[6]	tcp4_seq_fops[12]
hideme.vfs	sys_getdents64‡	proc_root_operations[6]	N/A
override	sys_call_table[220]	sys_call_table[220]	sys_call_table[3]
Synapsys-0.4	sys_call_table[141]	sys_call_table[141]	sys_call_table[4]
Rial	sys_call_table[141]	sys_call_table[141]	sys_call_table[3], sys_call_table[5] sys_call_table[6]
knark	sys_call_table[141] sys_call_table[220]	sys_call_table[141] sys_call_table[220]	sys_call_table[3]
kis-0.9	sys_call_table[141]	sys_call_table[141]	tcp4_seq_fops[12]

tries – *sys_getdents* (sys_call_table[141]) and *sys_getdents64* (sys_call_table[220]) – are hijacked for hiding files and processes while another one – *sys_write* (sys_call_table[4]) – is replaced to hide network activities related to backdoor processes protected by the rootkit. A customized user-space program called *ava* is provided to send hiding instructions to the malicious LKM so that certain files or processes of attackers' choices can be hidden. All these three kernel hooks are uncovered by our system, as shown in Tables 1, 2, and 3, respectively.

The *Adore-ng* Rootkit. As the name indicates, this rootkit is a more advanced successor from the previous *adore* rootkit. Instead of directly manipulating the system call table, the *adore-ng* rootkit subverts the jump table of the virtual file system by replacing the directory listing handler routines with its own ones. Such replacement allows it to manipulate the information about the *root* file system as well as the */proc* pseudo-file system to achieve the file-hiding or process-hiding purposes. More specifically, the *readdir* function pointer (*ext3_dir_operations[6]*) in the root file system operations table is hooked for hiding attack files, while the similar function (*proc_root_operations[6]*) in the */proc* file system operations table is hijacked for hiding attack processes. The fact that the kernel hook *ext3_dir_operations[6]* is located in the loadable module space (*ext3.ko*) indicates that this rootkit is more stealthier and these types of kernel hooks are much more difficult to uncover than those kernel hooks at static memory locations (e.g., the system call table). Once again, our system successfully identified these stealth kernel hooks, confirming our observation in Section 1. Further, the comparisons between those hooks used by rootkits (Table 4) and the list of hooks from our system (Tables 1, 2, and 3) indicate that only a small subset of them have been used.

6 Discussion

Our system leverages the nature of persistent kernel rootkits to systematically discover those kernel hooks that can potentially be exploited for hiding purposes. However, as a rootkit may implant other kernel hooks for other non-hiding features as its payload, our current prototype is ineffective in identifying them. However, the prototype can be readily re-targeted to those non-hiding features and apply the same techniques to identify those kernel hooks. Also, our system by design only works for persistent kernel rootkits but could be potentially extended for other types of rootkits as well (e.g,. persistent user-level rootkits).

Our current prototype is developed to identify those kernel hooks related to the execution of a chosen security program, either an anti-rootkit software or a system utility program. However, with different programs as the input, it is likely that different running instances will result in different sets of kernel hooks. Fortunately, for the rootkit author, he faces the challenge in hiding itself from all security programs. As a result, our defense has a unique advantage in only analyzing a single instantiated execution path of a rootkit-detection program. In other words, a persistent kernel rootkit cannot evade its detection if the hijacked kernel hooks are not a part of the corresponding kernel-side execution path. There may exist some "in-the-wild" rootkits that take chances in only evading selected security software. However, in response, we can monitor only those kernel hooks related to an installed security software. As mentioned earlier, to hide from it, persistent kernel rootkits will hijack at least one of these kernel hooks.

Meanwhile, it may be argued that our results from monitoring a running instance of a security program could lead to false positives. However, the fact that these kernel hooks exist in the kernel-side execution path suggest that each one could be equally exploited for hooking purposes. From another perspective, we point out that the scale of our results is manageable since it contains tens, not hundreds, of kernel hooks.

Finally, we point out that our current prototype only considers those kernel objects or variables that may contain kernel hooks of interest to rootkits. However, there also exist other types of kernel data such as non-control data [9] (e.g., the *uid* field in the process control block data structure or the doubly-linked process list), which can be manipulated to contaminate kernel execution. Though they may not be used to implement a persistent kernel rootkit for control-flow modifications, how to extend the current system to effectively address them (e.g., by real-time enforcing kernel control flow integrity [10]) remains as an interesting topic for future work.

7 Related Work

Hook Identification. The first area of related work is the identification of kernel hooks exploitable by rootkits for hiding purposes. Particularly, HookFinder [23] analyzes a given rootkit example and reports a list of kernel hooks that are being used by the malware. However, by design, it does not lead to the identification of other kernel hooks that are not being used but could still be potentially exploited for the same hiding purposes. From another perspective, SBCFI [14] performs static analysis of Linux kernel source code and aims to build a kernel control-flow graph that will be followed by a legitimate kernel at run-time. However, the graph is not explicitly associated with those kernel hooks for rootkit hiding purposes. Furthermore, the lack of run-time information could greatly limit its accuracy. In comparison, our system complements them with the unique capability of exhaustively deriving those kernel hooks for a given security program, which could be potentially hijacked by a persistent rootkit to hide from it.

Hook-based Rootkit Detection. The second area of related work is the detection of rootkits based on the knowledge of those specific hooking points that may be used by rootkits. For example, existing anti-rootkit tools such as VICE [8], IceSword [16], System Virginity Verifier [17] examine known memory regions occupied by these specific hooking points to detect any illegitimate modification. Our system is designed with a unique focus in uncovering those specific kernel hooks. As a result, they can be naturally combined together to build an integrated rootkit-defense system.

Other Rootkit Defenses. There also exist a number of recent efforts [12,13,15,20,21] [22] that defend against rootkits by detecting certain anomalous symptoms likely caused by rootkit infection. For example, The Strider GhostBuster system [21] and VMwatcher [12] apply the notion of cross-view detection to expose any discrepancy caused by stealth rootkits. CoPilot [15] as well as the follow-up work [13] identify rootkits by detecting possible violations in kernel code integrity or semantic constraints among multiple kernel objects. SecVisor [20] aims to prevent unauthorized kernel code from execution in the kernel space. Limbo [22] characterizes a number of run-time features that can best distinguish between legitimate and malicious kernel drivers and then utilizes them to prevent a malicious one from being loaded into the kernel. Our system is complementary to these systems by pinpointing specific kernel hooks that are likely to be chosen by stealth rootkits for manipulation.

8 Conclusion

To effectively counter persistent kernel rootkits, we have presented a systematic approach to uncover those kernel hooks that can be potentially hijacked by them. Our approach is based on the insight that those rootkits by their nature will tamper with the execution of deployed rootkit-detection software. By instrumenting and recording possible control-flow transfer instructions in the kernel-side execution paths related to the deployed security software, we can reliably derive all related kernel hooks. Our experience in building a prototype system as well as the experimental results with real-world rootkits demonstrate the effectiveness of the proposed approach.

Acknowledgments

The authors would like to thank the anonymous reviewers for their insightful comments that helped to improve the presentation of this paper. This work was supported in part by NSF Grants CNS-0716376, CNS-0524286, and CCF-0728771.

References

1. The adore Rootkit, http://lwn.net/Articles/75990/
2. The Hideme Rootkit, http://www.sophos.com/security/analyses/viruses-and-spyware/trojhidemea.html

3. The Strange Decline of Computer Worms, http://www.theregister.co.uk/2005/03/17/f-secure_websec/print.html
4. VMware, http://www.vmware.com/
5. Agrawal, H., Horgan, J.R.: Dynamic Program Slicing. In: Proceedings of ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation (1990)
6. Bellard, F.: QEMU, a Fast and Portable Dynamic Translator. In: Proc. of USENIX Annual Technical Conference 2005 (FREENIX Track) (July 2005)
7. Butler, J.: R$\hat{2}$: The Exponential Growth of Rootkit Techniques, http://www.blackhat.com/presentations/bh-usa-06/BH-US-06-Butler.pdf
8. Butler, J.: VICE 2.0, http://www.infosecinstitute.com/blog/README_VICE.txt
9. Chen, S., Xu, J., Sezer, E.C., Gauriar, P., Iyer, R.: Non-Control-Data Attacks Are Realistic Threats. In: Proc. USENIX Security Symposium (August 2005)
10. Grizzard, J.B.: Towards Self-Healing Systems: Re-Establishing Trust in Compromised Systems. Ph.D. thesis, Georgia Institute of Technology (May 2006)
11. Jiang, X., Wang, X.: "Out-of-the-Box" Monitoring of VM-Based High-Interaction Honeypots. In: Kruegel, C., Lippmann, R., Clark, A. (eds.) RAID 2007. LNCS, vol. 4637, pp. 198–218. Springer, Heidelberg (2007)
12. Jiang, X., Wang, X., Xu, D.: "Out-of-the-Box" Semantic View Reconstruction. In: Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS 2007) (October 2007)
13. Petroni, N., Fraser, T., Walters, A., Arbaugh, W.: An Architecture for Specification-Based Detection of Semantic Integrity Violations in Kernel Dynamic Data. In: Proc. of the 15th USENIX Security Symposium (August 2006)
14. Petroni, N., Hicks, M.: Automated Detection of Persistent Kernel Control-Flow Attacks. In: Proc. of ACM CCS 2007 (October 2007)
15. Petroni, N.L., Fraser, T., Molina, J., Arbaugh, W.A.: Copilot - a Coprocessor-based Kernel Runtime Integrity Monitor. In: Proc. of the 13th USENIX Security Symposium (August 2004)
16. PJF. IceSword, http://www.antirootkit.com/software/IceSword.htm, http://pjf.blogcn.com/
17. Rutkowska, J.: System Virginity Verifier, http://invisiblethings.org/papers/hitb05_virginity_verifier.ppt
18. Rutkowska, J.: Rootkits vs. Stealth by Design Malware, http://invisiblethings.org/papers/rutkowska_bheurope2006.ppt
19. sd.: Linux on-the-fly kernel patching without LKM. Phrack 11(58), article 7 of 15 (2001)
20. Seshadri, A., Luk, M., Qu, N., Perrig, A.: SecVisor: A Tiny Hypervisor to Guarantee Lifetime Kernel Code Integrity for Commodity OSes. In: Proc. of the ACM SOSP 2007 (October 2007)
21. Wang, Y., Beck, D., Vo, B., Roussev, R., Verbowski, C.: Detecting Stealth Software with Strider GhostBuster. In: Proc. of the 2005 International Conference on Dependable Systems and Networks (June 2005)
22. Wilhelm, J., Chiueh, T.-c.: A Forced Sampled Execution Approach to Kernel Rootkit Identification. In: Kruegel, C., Lippmann, R., Clark, A. (eds.) RAID 2007. LNCS, vol. 4637, pp. 219–235. Springer, Heidelberg (2007)
23. Yin, H., Liang, Z., Song, D.: HookFinder: Identifying and Understanding Malware Hooking Behaviors. In: Proc. of ISOC NDSS 2008 (February 2008)
24. Zhang, X., Gupta, R., Zhang, Y.: Precise Dynamic Slicing Algorithms. In: Proc. of the IEEE/ACM International Conference on Software Engineering (May 2003)

Tamper-Resistant, Application-Aware Blocking of Malicious Network Connections

Abhinav Srivastava and Jonathon Giffin

School of Computer Science, Georgia Institute of Technology
{abhinav,giffin}@cc.gatech.edu

Abstract. Application-level firewalls block traffic based on the process that is sending or receiving the network flow. They help detect bots, worms, and backdoors that send or receive malicious packets without the knowledge of users. Recent attacks show that these firewalls can be disabled by knowledgeable attackers. To counter this threat, we develop VMwall, a fine-grained tamper-resistant process-oriented firewall. VMwall's design blends the process knowledge of application-level firewalls with the isolation of traditional stand-alone firewalls. VMwall uses the Xen hypervisor to provide protection from malware, and it correlates TCP or UDP traffic with process information using virtual machine introspection. Experiments show that VMwall successfully blocks numerous real attacks—bots, worms, and backdoors—against a Linux system while allowing all legitimate network flows. VMwall is performant, imposing only a 0–1 millisecond delay on TCP connection establishment, less than a millisecond delay on UDP connections, and a 1–7% slowdown on network-bound applications. Our attack analysis argues that with the use of appropriate external protection of guest kernels, VMwall's introspection remains robust and helps identify malicious traffic.

Keywords: Firewall, virtual machine introspection, attack prevention.

1 Introduction

Application-level firewalls are an important component of a computer system's layered defenses. They filter inbound and outbound network packets based on an access policy that includes lists of processes allowed to make network connections. This fine-grained filtering is possible because application-level firewalls have a complete view of the system on which they execute. In contrast, network- or host-level firewalls provide coarse-grained filtering using ports and IP addresses. Application-level firewalls help detect and block malicious processes, such as bots, worms, backdoors, adware, and spyware, that try to send or receive network flows in violation of the fine-grained policies. To be successful, these firewalls must be fast, mediate all network traffic, and accurately identify executing processes.

The conventional design of application-level firewalls has a deficiency that may prevent filtering of malicious traffic. The architectures pass packet information from a kernel-level network tap up to a user-level firewall process that executes alongside malicious software. The firewall is both performant and able to identify the processes attached to a network flow, but it is exposed to direct attack by any malicious software aware of the firewall. Baliga et al. [1] demonstrated the ease of such attacks by manipulating the netfilter framework inside the Linux kernel to remove the hooks to packet filtering functions. Similarly, attackers can disable the Windows Firewall by halting particular services normally running on the system. Once the firewall fails, then all network traffic will be unmediated and the malware can send and receive data at will.

An alternative design isolates firewalls from vulnerable systems to gain protection from direct attack. Virtual machines allow construction of firewall appliances that execute outside of operating systems under attack. Such firewalls dispense with application-level knowledge and filter inbound and outbound packets using coarse-grained rules over IP addresses and port numbers. Attacks can easily evade these firewalls by using allowed ports directly or via tunneling.

This paper leverages the benefits of both application-level firewalls and virtual machine isolation to develop tamper-resistant application-oriented firewalls. Such a firewall needs good visibility of the system so that it can correlate network flows with processes, but it also needs strong isolation from any user-level or kernel-level malware that may be present. We architect an application-level firewall resistant to direct attack from malicious software on the system. Our design isolates the application-level firewall in a trusted virtual machine (VM) and relies on the hypervisor to limit the attack surface between any untrusted VM running malware and the trusted VM. Our firewall, executing in the trusted VM, becomes an application-level firewall by using virtual machine introspection (VMI) [10] to identify the process in another VM that is connected to a suspicious network flow.

Our prototype implementation, VMwall, uses the Xen [2] hypervisor to remain isolated from malicious software. VMwall executes entirely within Xen's trusted virtual machine dom0; it operates with both paravirtualized and fully virtualized domains. A dom0 kernel component intercepts network connections to and from untrusted virtual machines. A user-space process performs introspection to correlate each flow to a sending or receiving process, and it then uses a predefined security policy to decide whether the connection should be allowed or blocked. Policies are straightforward whitelists of known software in the untrusted VM allowed to communicate over the network. To correlate network flows with processes, VMwall's user-space component maps the untrusted operating system's kernel memory into its own address space and uses programmed knowledge of kernel data structures to extract the identity of the process attached to the flow.

VMwall is effective at identifying and blocking malicious network connections without imposing significant performance degradation upon network traffic. Using a Linux system and a collection of known attacks that either send or receive network traffic, we show that VMwall identifies all malicious connections immediately when the first packet is sent or received. In particular, VMwall blocked 100% of the malicious connections when tested against bots, worms, and backdoors, and it correctly allowed all legitimate network traffic. In our design, VMwall only performs introspection for the first packet of a new connection, so network performance remains high. Our tool adds only about 0–1 milliseconds of overhead to the first packet of a session. This is a latency cost to network connection creation that will not impact the subsequent data transfer of legitimate connections.

VMwall looks into the state of the untrusted operating system's memory to find the process bound to a network connection. The system monitors network flows, and it is not an intrustion detection system designed to detect an attack against the OS. Hence, an attacker may try to evade VMwall either by hijacking a process or by subverting the inspected kernel data structures. In Sect. 6.4, we study this problem, provide an in-depth security analysis of VMwall, and suggest appropriate measures to thwart these attacks.

We believe that our tamper-resistant application-oriented firewall represents an appropriate use of virtualization technology for improved system security. We feel that our paper provides the following contributions:

– Correlation between network flows and processes from outside the virtual machine (Sect. 4).
– VMwall, an implementation of a tamper-resistant application-oriented firewall (Sect. 5).
– Evidence that application-aware firewalls outside the untrusted virtual machine can block malicious network connections successfully while maintaining network performance (Sect. 6).

2 Related Work

Prior research has contributed to the development of conventional host-based firewalls. Mogul et al. [21] developed a kernel-resident packet filter for UNIX that gave user processes flexibility in selecting legitimate packets. Venema [29] designed a utility to monitor and control incoming network traffic. These traditional firewalls performed filtering based on restrictions inherent in network topology and assumed that all parties inside the network were trusted. As part of the security architecture of the computer system, they resided in kernel-space and user-space, and hence were vulnerable to direct attack by malicious software.

Administration of firewalls can be cumbersome, and distributed firewalls have been proposed to ease the burden [3,15]. In distributed firewalls, an administrator manages security policies centrally but pushes enforcement of these policies out to the individual hosts. Although we have not implemented support for distributed management, we expect VMwall to easily fit into this scheme. VMwall policies dictate which processes can legitimately make use of network resources. In a managed environment where administrators are knowledgeable of the software running on the machines in the local network, preparing and distributing VMwall policies from a central location may be an appealing solution.

The recent support for virtual machines by commodity hardware has driven development of new security services deployed with the assistance of VMs [27, 9, 30]. Garfinkel et al. [11] showed the feasibility of implementing distributed network-level firewalls using virtual machines. In another work [10], they proposed an intrusion detection system design using virtual machine introspection of an untrusted VM. VMwall applies virtual machine introspection to a different problem, using it to correlate network flows with the local processes bound to those flows.

Other research used virtual machines for malware detection. Borders et al. [4] designed a system, Siren, that detected malware running within a virtual machine. Yin et al. [33] proposed a system to detect and analyze privacy-breaching malware using taint analysis. Jiang et al. [17] presented an out-of-the-box VMM-based malware detection system. Their proposed technique constructed the internal semantic views of a VM from an external vantage point. In another work [16], they proposed a monitoring tool that observes a virtual machine based honeypot's internal state from outside the honeypot. As a pleasant side-effect of malicious network flow detection and process correlation, VMwall can often identify processes in the untrusted system that comprise portions of an attack.

Previous research has developed protection strategies for different types of hardware-level resources in the virtualized environment. Xu et al. [32] proposed a VMM-based usage control model to protect the integrity of kernel memory. Ta-Min et al. [28] proposed a hypervisor based system that allowed applications to partition their system call interface into trusted and untrusted components. VMwall, in contrast, protects network resources from attack by malware that runs inside the untrusted virtual machine by blocking the illegitimate network connections attempts.

These previous hypervisor-based security applications generally take either a network-centric or host-centric view. Our work tries to correlate activity at both levels. VMwall monitors network connections but additionally peers into the state of the running, untrusted operating system to make its judgments about each connection's validity. Moreover, VMwall easily scales to collections of virtual machines on a single physical host. A single instance of VMwall can act as an application-level firewall for an entire network of VMs.

3 Overview

We begin with preliminaries. Section 3.1 explains our threat model, which assumes that attackers have the ability to execute the real-world attacks infecting widespread computer systems today. Section 3.2 provides a brief overview of Xen-based virtual machine architectures and methods allowing inspection of a running VM's state.

3.1 Threat Model

We assume that attackers have abilities commonly displayed by real-world attacks against commodity computer systems. Attackers can gain *superuser priv-*

ilege from remote. Attackers are external and have no physical access to the attacked computers, but they may install malicious software on a victim system by exploiting a software vulnerability in an application or operating system or by enticing unsuspecting users to install the malware themselves. The software exploit or the user often executes with full system privileges, so the malware may perform administrative actions such as kernel module or driver installation. Hence, malicious code may execute at both user and kernel levels. For ease of explanation, we initially describe VMwall's architecture in Sect. 4 under the assumption that kernel data structure integrity is maintained. This assumption is not valid in our threat model, and Sect. 6.4 revisits this point to describe technical solutions ensuring that the assumption holds.

The installed malware may periodically make or receive network connections. Many examples exist. *Bots* make network connections to a command and control channel to advertise their presence and receive instruction, and they send bulk network traffic such as denial-of-service packets and email spam. *Spyware* programs collect information, such as keystrokes and mouse clicks, and then transmit the confidential data across a network to the attacker. *Worms* may generate network connections to scan the network in search of additional victims suitable for infection. *Backdoors* open holes in machines by listening for incoming connections from the attacker. One common feature of these different classes of attacks is their interest in the network.

In a typical system, malware can directly affect an application-level firewall's execution. The architecture of these malware instances frequently combines a user-level application performing network activity with a kernel-level module that hides the application from the view of host-level security software. The malicious application, likely running with full system privileges, may halt the execution of the firewall. Similarly, the malicious kernel component may alter the hooks used by an in-kernel module supporting the user-level firewall so that the firewall is simply never invoked as data passes to and from the network. Conventional application-level firewalls fail under these direct attacks. Our goal is to develop a system that withstands direct attack from malware at the application layer or the kernel layer.

Our system has requirements for correct execution. As with all requirements, an attacker who is able to violate any requirement is likely able to escape detection. Our two requirements of note center on basic expectations for the in-memory data structures used by the kernel that may be infected by an attack.

First, we expect to be able to find the head of linked data structures, often by extracting a kernel symbol value at boot time. An attacker could conceivably cause our firewall to inspect the incorrect kernel information by replicating the data structure elsewhere in kernel memory and by altering all code references to the original structure to instead refer to the new structure. Our firewall would then analyze stale data. It is not immediately clear that such an attack is plausible; moreover, our tool could periodically verify that code references to the data match the symbol value extracted at boot.

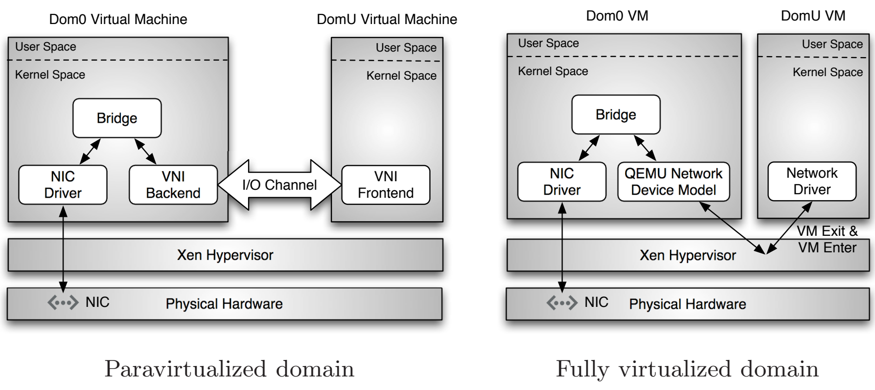


Fig. 1. Xen networking architecture

Second, we expect that attacks do not alter the ordering or length of fields in aggregate data structures. Our firewall is preprogrammed with type information about kernel structures, and an attack that alters the structure types would cause our system to read incorrect information from kernel memory. Successfully executing this attack without kernel recompilation appears to be complex, as all kernel code that accesses structure fields would need to be altered to use the attacker's structure layout. As a result, we believe that relying upon known structure definitions is not a limiting factor to our design.

3.2 Virtual Machine Introspection

Our design makes use of virtual machine technology to provide isolation between malicious code and our security software. We use Xen [2], an open source hypervisor that runs directly on the physical hardware of a computer. The virtual machines running atop Xen are of two types: unprivileged domains, called domU or guest domains, and a single fully-privileged domain, called dom0. We run normal, possibly vulnerable software in domU and deploy our application-level firewall in the isolated dom0.

Xen virtualizes the network input and output of the system. Dom0 is the device driver domain that runs the native network interface card driver software. Unprivileged virtual machines cannot directly access the physical network card, so Xen provides them with a virtualized network interface (VNI). The driver domain receives all the incoming and outgoing packets for all domU VMs executing on the physical system. Dom0 provides an Ethernet bridge connecting the physical network card to all virtual network devices provided by Xen to the domU VMs. (Xen offers other networking modes, such as network address translation, that are not used in our work and will not be considered further.) Dom0 uses its virtual bridge to multiplex and demultiplex packets between the physical network interface and each unprivileged virtual machine's VNI. Figure 1 shows the Xen networking architecture when the virtual machines' network interfaces are connected through a virtual Ethernet bridge. The guest VMs send and receive packets via either an I/O channel to dom0 or emulated virtual devices.

The strong isolation provided by a hypervisor between dom0 and the guest domains complicates the ability to correlate network flows with software executing in a guest domain. Yet, dom0 has complete access to the entire state of the guest operating systems running in untrusted virtual machines. *Virtual machine introspection* (VMI) [10] is a technique by which dom0 can determine execution properties of guest VMs by monitoring their runtime state, generally through direct memory inspection. VMI allows security software to remain protected from direct attack by malicious software executing in a guest VM while still able to observe critical system state.

Xen offers low-level APIs to allow dom0 to map arbitrary memory pages of domU as shared memory. XenAccess [31] is a dom0 userspace introspection library developed for Xen that builds onto the low-level functionality provided by Xen. VMwall uses XenAccess APIs to map raw memory pages of domU's kernel inside dom0. It then builds higher-level memory abstractions, such as aggregate structures and linked data types, from the contents of raw memory pages by using the known coding semantics of the guest operating system's kernel. Our application-level firewall inspects these meaningful, higher-level abstractions to determine how applications executing in the guest VM use network resources.

4 Tamper Resistant Architecture of VMwall

VMwall is our application-level firewall designed to resist the direct attacks possible in our threat model. The architecture of VMwall is driven by the following three goals:

– **Tamper Resistance:** VMwall should continue to function reliably and verify all network connections even if an attacker gains entry into the monitored system. In particular, the design should not rely on components installed in the monitored host as processes or kernel modules, as these have been points of direct attack in previous application-level firewalls.
– **Independence:** VMwall should work without any cooperation from the monitored system. In fact, the system may not be aware of the presence of the firewall.
– **Lightweight Verification:** Our intent is to use VMwall for online verification of network connections to real systems. The design should allow for efficient monitoring of network traffic and correlation to applications sending and receiving that traffic.

Our firewall design satisfies these goals by leveraging virtual machine isolation and virtual machine introspection. Its entire software runs within the privileged dom0 VM, and it hooks into Xen's virtual network interface to collect and filter all guest domains' network packets. Since the hypervisor provides strong isolation among the virtual machines, this design achieves the first goal of tamper-resistance.

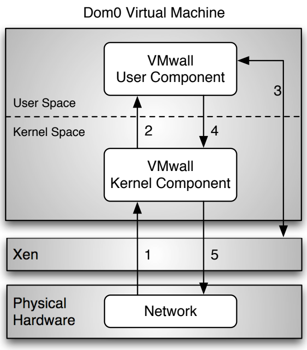


Fig. 2. VMwall's high-level architecture. (1) Packets inbound to and outbound from a guest domain are processed by dom0. (2) The VMwall kernel component intercepts the packets and passes them to a user-space component for analysis. (3) The user-space component uses virtual machine introspection to identify software in a guest domain processing the data. (4) The user-space component instructs the kernel component to either allow or block the connection. (5) Packets from allowed connections will be placed on the network.

In order to provide application-level firewalling, VMwall must identify the process that is sending or receiving packets inside domU. VMwall correlates packet and process information by directly inspecting the domU virtual machine's memory via virtual machine introspection. It looks into the kernel's memory and traverses the data structures to map process and network information. This achieves our second design goal of independence, as there are no components of VMwall inside domU. Our introspection procedure rapidly analyzes the kernel's data structures, satisfying the third goal of lightweight verification.

The high-level design of VMwall has two components: a kernel module and user agent, both in dom0 (Fig. 2). The VMwall kernel component enforces a per-packet policy given by the user agent and either allows or drops each packet. The user agent determines policy by performing introspection to extract information about processes executing in guest VMs and evaluating the legitimacy of those processes. Sections 4.1 and 4.2 present detailed information about the two components.

4.1 Kernel Component

VMwall's kernel component is a module loaded inside the dom0 Linux kernel. It intercepts all network packets to or from untrusted virtual machines and uses security policies to decide whether each packet should be allowed or dropped. Interception occurs by hooking into Xen's network bridge between the physical interface card and virtual network interface. When the kernel component intercepts a packet, it checks a rule table to see if a firewall rule already exists for the packet, as determined by the local endpoint IP address and port. If so, it takes the allow or block action specified in the rule. If there is no rule, then it

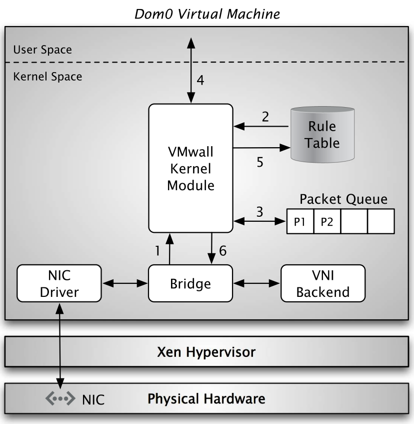


Fig. 3. VMwall's kernel module architecture. (1) Packets inbound to and outbound from a guest domain are intercepted and passed to the kernel module. (2) The module receives each packet and looks into its rule table to find the rule for the packet. (3) The kernel module queues the packet if there is no rule present. (4) VMwall sends an introspection request to the user agent and, after the agent completes, receives the dynamically generated rule for the packet. (5) The kernel module adds the rule into its rule table to process future packets from the same connection. (6) The kernel module decides based on the action of the rule either to accept the packet by reinjecting it into the network or to drop it from the queue.

invokes the VMwall user agent to analyze the packet and create a rule. The user agent performs introspection, generates a rule for the packet, and sends this rule back to the kernel module. The kernel module adds this new rule to its policy table and processes the packet. Further packets from the same connection are processed using the rule present in the kernel component without invoking the user agent and without performing introspection.

As kernel code, the kernel component cannot block and must take action on a packet before the user agent completes introspection. VMwall solves this problem for packets of unknown legitimacy by queuing the packets while waiting for the user agent's reply. When the user agent sends a reply, the module adds a rule for the connection. If the rule's action is to block the connection, then it drops all the packets that are queued. Otherwise, it re-injects all the packets into the network.

Figure 3 presents the kernel module's complete architecture. It illustrates the steps involved in processing the packet inside the kernel. It shows the queue architecture, where packets are stored inside the kernel during introspection.

4.2 User Agent

The VMwall user agent uses virtual machine introspection to correlate network packets and processes. It receives introspection requests from the kernel component containing network information such as source port, source IP address,

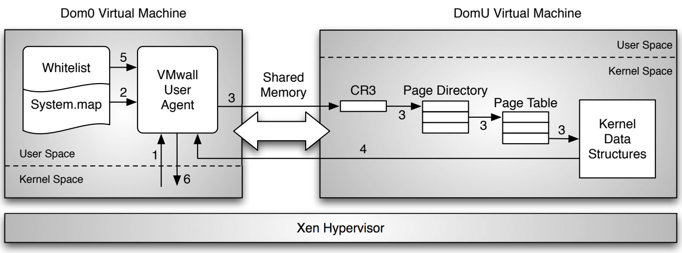


Fig. 4. VMwall's user agent architecture. (1) The VMwall user agent receives the introspection request. (2) The user agent reads the System.map file to extract the kernel virtual addresses corresponding to known kernel symbols. (3) The user agent uses Xen to map the domU kernel memory pages containing process and network data structures. (4) VMwall traverses the data structures to correlate network and process activity. (5) The agent searches for the recovered process name in the whitelist. (6) The user agent sends a filtering rule for the connection to the VMwall kernel module.

destination port, destination IP address, and protocol. It first uses the packet's source (or destination) IP address to identify the VM that is sending (or receiving) the packet. When it finds the VM, it then tries to find the process that is bound to the source (or destination) port.

VMwall's user agent maps a network port to the domU process that is bound to the port, shown in Fig. 4. As needed, it maps domU kernel data structures into dom0 memory. Process and network information is likely not available in a single data structure but instead is scattered over many data structures. VMwall works in steps by first identifying the domU kernel data structures that store IP address and port information. Then, VMwall identifies the process handling this network connection by iterating over the list of running processes and checking each process to see if it is bound to the port. When it finds the process bound to the port, it extracts the process' identifier, its name, and the full path to its executable. If the user agent does not find any process bound to the port, it considers this to be an anomaly and will block the network connection.

VMwall uses information about the process to create a firewall rule enforceable by the kernel component. The user agent maintains a whitelist of processes that are allowed to make network connections. When the user agent extracts the name of a process corresponding to the network packet, it searches the whitelist for the same name. VMwall allows the connection if it finds a match and blocks the connection otherwise. It then generates a rule for this connection that it passes to the VMwall kernel component. This rule contains the network connection information and either an allow or block action. The kernel component then uses this rule to filter subsequent packets in this attempted connection.

5 Implementation

We have implemented a prototype of VMwall using the Xen hypervisor and a Linux guest operating system. VMwall supports both paravirtualized and fully-virtualized (HVM) Linux guest operating systems. Its implementation consists of two parts corresponding to the two pieces described in the previous section: the kernel module and the user agent. The following sections describe specific details affecting implementation of the two architectural components.

5.1 Extending Ebtables

Our kernel module uses a modified ebtables packet filter to intercept all packets sent to or from a guest domain. Ebtables [7] is an open source utility that filters packets at an Ethernet bridge. VMwall supplements the existing coarse-grained firewall provided by ebtables. Whenever ebtables accepts packets based on its coarse-grained rules, we hook the operation and invoke the VMwall kernel module for our additional application-level checks. We modified ebtables to implement this hook, which passes a reference to the packet to VMwall.

Ebtables does not provide the ability to queue packets. Were it present, queuing would enable filters present inside the kernel to store packets for future processing and reinjection back into the network. To allow the VMwall kernel module to queue packets currently under inspection by the user agent, we altered ebtables to incorporate packet queuing and packet reinjection features.

5.2 Accessing DomU Kernel Memory

VMwall uses the XenAccess introspection library [31] to accesses domU kernel memory from dom0. It maps domU memory pages containing kernel data structures into the virtual memory space of the user agent executing in the trusted VM. XenAccess provides APIs that map domU kernel memory pages identified either by explicit kernel virtual addresses or by exported kernel symbols. In Linux, the exported symbols are stored in the file named `System.map`. VMwall uses certain domU data structures that are exported by the kernel and hence mapped with the help of kernel symbols. Other data structures reachable by pointers from the known structures are mapped using kernel virtual addresses. The domU virtual machine presented in Fig. 4 shows the internal mechanism involved to map the memory page that contains the desired kernel data structure.

5.3 Parsing Kernel Data Structures

To identify processes using the network, VMwall must be able to parse high-level kernel data structures from the raw memory pages provided by XenAccess. Extracting kernel data structures from the mapped memory pages is a non-trivial task. For example, Linux maintains a doubly-linked list that stores the kernel's private data for all running processes. The head pointer of this list is stored in the exported kernel symbol `init_task`. If we want to extract the list of processes running inside domU, we can map the memory page of domU that contains the `init_task` symbol. However, VMwall must traverse the complete linked list and hence requires the offset to the `next` member in the process structure. We extract this information offline directly from the kernel source code and use these values

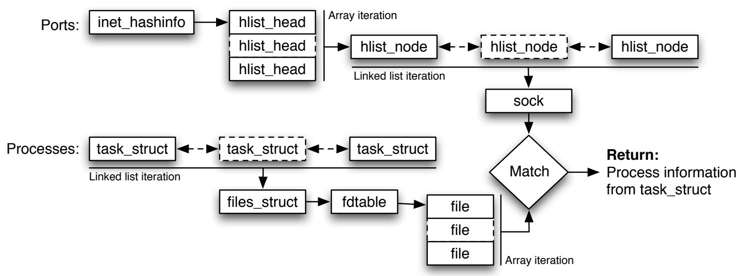


Fig. 5. DomU Linux kernel data structures traversed by the VMwall user agent during correlation of the process and TCP packet information

in the user agent. This source code inspection is not the optimal way to identify offsets because the offset values often change with the kernel versions. However, there are other automatic ways to extract this information from the kernel binary if it was compiled with a debug option [18].

This provides VMwall with sufficient information to traverse kernel data structures. VMwall uses known field offsets to extract the virtual addresses of pointer field members from the mapped memory pages. It then maps domU memory pages by specifying the extracted virtual addresses. This process is performed recursively until VMwall traverses the data structures necessary to extract the process name corresponding to the source or destination port of a network communication. Figure 5 shows the list of the kernel data structures traversed by the user agent to correlate a TCP packet and process information. First, it tries to obtain a reference to the socket bound to the port number specified in the packet. After acquiring this reference, it iterates over the list of processes to find the process owning the socket.

5.4 Policy Design and Rules

VMwall identifies legitimate connections via a whitelist-based policy listing processes allowed to send or receive data. Each process that wants to communicate over the network must be specified in the whitelist *a priori*. This whitelist resides inside dom0 and can only be updated by administrators in a manner similar to traditional application-level firewalls. The whitelist based design of VMwall introduces some usability issues because all applications that should be allowed to make network connections must be specified in the list. This limitation is not specific to VMwall and is inherent to the whitelist based products and solutions [6,12].

VMwall's kernel module maintains its own rule table containing rules that are dynamically generated by the user agent after performing introspection. A rule contains source and destination port and IP address information, an action, and a timeout value used by the kernel module to expire and purge old rules for UDP connections. In the case of TCP connections, the kernel module purges TCP rules automatically whenever it processes a packet with the TCP `fin` or `rst` flag set. In an abnormal termination of a TCP connection, VMwall uses the timeout mechanism to purge the rules.

6 Evaluation

The basic requirement of an application-level firewall is to block connections to or from malicious software and allow connections to or from benign applications. We evaluated the ability of VMwall to filter out packets made by several different classes of attacks while allowing packets from known processes to pass unimpeded. We tested VMwall against Linux-based backdoors, worms, and bots that attempt to use the network for malicious activity. Section 6.1 tests VMwall against attacks that receive inbound connections from attackers or connect out to remote systems. Section 6.2 tests legitimate software in the presence of VMwall. We measure VMwall's performance impact in Sect. 6.3, and lastly analyze its robustness to a knowledgeable attacker in Sect. 6.4.

6.1 Illegitimate Connections

We first tested attacks that receive inbound connections from remote attackers. These attacks are rootkits that install backdoor programs. The backdoors run as user processes, listen for connections on a port known to the attacker, and receive and execute requests sent by the attacker. We used the following backdoors:

– **Blackhole** runs a TCP server on port 12345 [22].
– **Gummo** runs a TCP server at port 31337 [22].
– **Bdoor** runs a backdoor daemon on port 8080 [22].
– **Ovas0n** runs a TCP server on port 29369 [22].
– **Cheetah** runs a TCP server at the attacker's specified port number [22].

Once installed on a vulnerable system, attacks such as worms and bots may attempt to make outbound connections without prompting from a remote attacker. We tested VMwall with the following pieces of malware that generate outbound traffic:

– **Apache-ssl** is a variant of the Slapper worm that self-propagates by opening TCP connections for port scanning [23].
– **Apache-linux** is a worm that exploits vulnerable Apache servers and spawns a shell on port 30464 [23].
– **BackDoor-Rev.b** is a tool that is be used by a worm to make network connections to arbitrary Internet addresses and ports [20].
– **Q8** is an IRC-based bot that opens TCP connections to contact an IRC server to receive commands from the botmaster [14].
– **Kaiten** is a bot that opens TCP connections to contact an IRC server [24].
– **Corompter Dunno** is an IRC-based bot providing basic functionalities such as port scanning [13].

Table 1. Results of executing legitimate software in the presence of VMwall. "Allowed" indicates that the network connections to or from the processes were passed as though a firewall was not present.

Name	*Connection Type*	*Result*
rcp	Outbound	Allowed
rsh	Outbound	Allowed
yum	Outbound	Allowed
rlogin	Outbound	Allowed
ssh	Outbound	Allowed
scp	Outbound	Allowed
wget	Outbound	Allowed
tcp client	Outbound	Allowed
thttpd	Inbound	Allowed
tcp server	Inbound	Allowed
sshd	Inbound	Allowed

VMwall successfully blocked all illegitimate connections attempted by malware instances. In all cases, both sending and receiving, VMwall intercepted the first SYN packet of each connection and passed it to the userspace component. Since these malicious processes were not in the whitelist, the VMwall user space component informed the VMwall kernel component to block these malicious connections. As we used VMwall in packet queuing mode, no malicious packets were ever passed through VMwall.

6.2 Legitimate Connections

We also evaluated VMwall's ability to allow legitimate connections made by processes running inside domU. We selected a few network applications and added their name to VMwall's whitelist. We then ran these applications inside domU. Table 1 shows the list of processes that we tested, the type of connections used by the processes, and the effect of VMwall upon those connections. To be correct, all connections should be allowed.

VMwall allowed all connections made by these applications. The `yum` application, a package manager for Fedora Core Linux, had runtime behavior of interest. In our test, we updated domU with the `yum update` command. During the package update, `yum` created many child processes with the same name `yum` and these child processes made network connections. VMwall successfully validated all the connections via introspection and allowed their network connections.

6.3 Performance Evaluation

A firewall verifying all packets traversing a network may impact the performance of applications relying on timely delivery of those packets. We investigated the performance impact of VMwall as perceived by network applications running inside the untrusted virtual machine. We performed experiments both with and

Table 2. Introspection time (μs) taken by VMwall to perform correlation of network flow with the process executing inside domU

Configuration	*TCP Introspection Time*	*UDP Introspection Time*
Inbound Connection to domU	251	438
Outbound Connection from domU	1080	445

without VMwall running inside dom0. All experiments were conducted on a machine with an Intel Core 2 Duo T7500 processor at 2.20 GHz with 2 GB RAM. Both dom0 and domU virtual machines ran 32 bit Fedora Core 5 Linux. DomU had 512 MB of physical memory, and dom0 had the remaining 1.5 GB. The versions of Xen and XenAccess were 3.0.4 and 0.3, respectively. We performed our experiments using both TCP and UDP connections. All reported results show the median time taken from five measurements. We measured microbenchmarks with the Linux `gettimeofday` system call and longer executions with the `time` command-line utility.

VMwall's performance depends on the introspection time taken by the user component. Since network packets are queued inside the kernel during introspection, the introspection time is critical for the performance of the complete system. We measured the introspection time both for incoming and outgoing connections to and from domU. Table 2 shows the results of experiments measuring introspection time.

It is evident that the introspection time for incoming TCP connections is very small. Strangely, the introspection time for outgoing TCP connections is notably higher. The reason for this difference lies in the way that the Linux kernel stores information for TCP connections. It maintains TCP connection information for listening and established connections in two different tables. TCP sockets in a listening state reside in a table of size 32, whereas the established sockets are stored in a table of size 65536. Since the newly established TCP sockets can be placed at any index inside the table, the introspection routine that iterates on this table from dom0 must search half of the table on average.

We also measured the introspection time for UDP data streams. Table 2 shows the result for UDP inbound and outbound packets. In this case, the introspection time for inbound and outbound data varies little. The Linux kernel keeps the information for UDP streams in a single table of size 128, which is why the introspection time is similar in both cases.

To measure VMwall's performance overhead on network applications that run inside domU, we performed experiments with two different metrics for both inbound and outbound connections. In the first experiment, we measured VMwall's impact on network I/O by transferring a 175 MB video file over the virtual network via `wget`. Our second experiment measured the time necessary to establish a TCP connection or transfer UDP data round-trip as perceived by software in domU.

We first transferred the video file from dom0 to domU and back again with VMwall running inside dom0. Table 3 shows the result of our experiments. The

Table 3. Time (seconds) to transfer a 175 MB file between dom0 and domU, with and without VMwall

Direction	*Without VMwall*	*With VMwall*	*Overhead*
File Transfer from Dom0 to DomU	1.105	1.179	7%
File Transfer from DomU to Dom0	1.133	1.140	1%

Table 4. Single TCP connection setup time (μs) measured both with and without VMwall inside dom0

Direction	*Without VMwall*	*With VMwall*	*Overhead*
Connection from Dom0 to DomU	197	465	268
Connection from DomU to Dom0	143	1266	1123

median overhead imposed by VMwall is less than 7% when transferring from dom0 to domU, and less than 1% when executing the reverse transfer.

Our second metric evaluated the impact of VMwall upon connection or data stream setup time as perceived by applications executing in domU. For processes using TCP, we measured both the inbound and outbound TCP connection setup time. For software using UDP, we measured the time to transfer a small block of data to a process in the other domain and to have the block echoed back.

We created a simple TCP client-server program to measure TCP connection times. The client program measured the time required to connect to the server, shown in Table 4. Inbound connections completed quickly, exhibiting median overhead of only 268 μs. Outbound connections setup from domU to dom0 had a greater median overhead of 1123 μs, due directly to the fact that the introspection time for outbound connections is also high. Though VMwall's connection setup overhead may look high as a percentage, the actual overhead remains slight. Moreover, the introspection cost occurring at connection setup is a one-time cost that gets amortized across the duration of the connection.

We lastly measured the time required to transmit a small block of data and receive an echo reply to evaluate UDP stream setup cost. We wrote a simple UDP echo client and server and measured the round-trip time required for the echo reply. Note that only the first UDP packet required introspection; the echo reply was rapidly handled by a rule in the VMwall kernel module created when processing the first packet. We again have both inbound and outbound measurements, shown in Table 5. The cost of VMwall is small, incurring slowdowns of 381 μs and 577 μs, respectively.

VMwall currently partially optimizes its performance, and additional improvements are clearly possible. VMwall performs introspection once per connection so that further packets from the same connection are allowed or blocked based on the in-kernel rule table. VMwall's performance could be improved in future work by introducing a caching mechanism to the introspection operation. The VMwall introspection routine traverses the guest OS data structures to perform correlation. In order to traverse a data structure, the memory page that contains the data structure needs to be mapped, which is a costly operation. One possi-

Table 5. Single UDP echo-reply stream setup time (μs) with and without VMwall. In an inbound-initiated echo, dom0 sent data to domU and domU echoed the data back to dom0. An outbound-initiated echo is the reverse.

Direction	*Without VMwall*	*With VMwall*	*Overhead*
Inbound Initiated	434	815	381
Outbound Initiated	271	848	577

ble improvement would be to support caching mechanisms inside VMwall's user agent to cache frequently used memory pages to avoid costly memory mapping operations each time.

6.4 Security Analysis

VMwall relies on particular data structures maintained by the domU kernel. An attacker who fully controls domU could violate the integrity of these data structures in an attempt to bypass VMwall's introspection. To counter such attacks, we rely on previous work in kernel integrity protection. Petroni et al. [26] proposed a framework for detecting attacks against dynamic kernel data structures such as `task_struct`. Their monitoring system executed outside the monitored kernel and detected any semantic integrity violation against the kernel's dynamic data. The system protected the integrity of the data structures with an external monitor that enforced high-level integrity policies. In another work, Loscocco et al. [19] introduced a system that used virtualization technology to monitor a Linux kernel's operational integrity. These types of techniques ensure that the kernel data structures read by VMwall remain valid.

Attackers can also try to cloak their malware by appearing to be whitelisted software. An attacker can guess processes that are in VMwall's whitelist by observing the incoming and outgoing traffic from the host and determining themselves what processes legally communicate over the network. They can then rename their malicious binary to the name of a process in the whitelist. VMwall counters this problem by extracting the full path to the process on the guest machine. Attackers could then replace the complete program binary with a trojaned version to evade the full path verification. VMwall itself has no defenses against this attack, but previous research has already addressed this problem with disk monitoring utilities that protect critical files [8,25].

An attacker could hijack a process by exploiting a vulnerability, and they could then change its in-memory image. To address this problem, VMwall user-space process can perform checksumming of the in-memory image of the process through introspection and compare it with previously stored hash value. However, this process is time consuming and may affect the connection setup time for an application.

An attacker could also hijack a connection after it has been established and verified by VMwall as legitimate. They could take control of the process bound to the port via a software exploit, or they could use a malicious kernel module to alter packet data before sending it to the virtual network interface. VMwall can counter certain instances of connection hijacking by timing out entries in its kernel rule table periodically. Subtle hijacking may require deep packet inspection within VMwall.

VMwall's kernel module internally maintains a small buffer to keep a copy of a packet while performing introspection. An attacker may try to launch a denial of service (DoS) attack, such as a `SYN` flood [5], against VMwall by saturating its internal buffer. VMwall remains robust to such attempted attacks because its buffer is independent of connection status. As soon as VMwall resolves the process name bound to a connection, it removes the packet from the buffer and does not wait for a TCP handshake to complete.

7 Conclusions and Future Work

We set out to design an application-oriented firewall resistant to the direct attacks that bring down these security utilities today. Our system, VMwall, remains protected from attack by leveraging virtual machine isolation. Although it is a distinct virtual machine, it can recover process-level information of the vulnerable system by using virtual machine introspection to correlate network flows with processes bound to those flows. We have shown the efficacy of VMwall by blocking backdoor, bot, and worm traffic emanating from the monitored system. Our malicious connection detection operates with reasonable overheads upon system performance.

Our current implementation operates for guest Linux kernels. VMwall could be made to work with Microsoft Windows operating systems if it can be programmed with knowledge of the data structures used by the Windows kernel. Since VMwall depends on the guest operating system's data structures to perform network and process correlation, it currently cannot be used for Windows-based guest systems. Recently, XenAccess started providing the ability to map Windows kernel memory into dom0 in the same way as done for Linux. If we have a means to identify and map Windows kernel data structures, then network and process correlation becomes possible.

Acknowledgment of Support and Disclaimer. This material is based upon work supported by the Defense Advanced Research Projects Agency and the United States Air Force under contract number FA8750-06-C-0182. 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 Defense Advanced Research Projects Agency and the United States Air Force.

We thank the anonymous reviewers for their comments that improved the quality of the paper. We thank Steve Dawson of SRI International for his assistance with this project. Portions of this work were performed while Abhinav Srivastava was at SRI International.

References

[1] Baliga, A., Kamat, P., Iftode, L.: Lurking in the shadows: Identifying systemic threats to kernel data. In: IEEE Symposium on Security and Privacy, Oakland, CA (May 2007)

[2] 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: 19^{th} ACM Symposium on Operating Systems Principles (SOSP), Bolton Landing, NY (October 2003)

[3] Bellovin, S.: Distributed firewalls. login (November 1999)

[4] Borders, K., Zhao, X., Prakash, A.: Siren: Catching evasive malware. In: IEEE Symposium on Security and Privacy, Oakland, CA (May 2005)

[5] CERT. TCP SYN Flooding and IP Spoofing Attacks. CERT Advisory CS-1996-21 (Last accessed April 4 , 2008), http://www.cert.org/advisories/CA-1996-21.html

[6] Check Point. ZoneAlarm (Last accessed April 4, 2008), http://www.zonealarm.com/store/content/home.jsp

[7] Community Developers. Ebtables (Last accessed November 1, 2007), http://ebtables.sourceforge.net/

[8] Community Developers. Tripwire (Last accessed November 1, 2007), http://sourceforge.net/projects/tripwire/

[9] Garfinkel, T., Pfaff, B., Chow, J., Rosenblum, M., Boneh, D.: Terra: A virtual machine-based platform for trusted computing. In: ACM Symposium on Operating Systems Principles (SOSP), October 2003, Bolton Landing, NY (2003)

[10] Garfinkel, T., Rosenblum, M.: A virtual machine introspection based architecture for intrusion detection. In: Network and Distributed System Security Symposium (NDSS), San Diego, CA, Feburary (2003)

[11] Garfinkel, T., Rosenblum, M., Boneh, D.: Flexible OS support and applications for trusted computing. In: 9th Hot Topics in Operating Systems (HOTOS), Lihue, HI (May 2003)

[12] Oskoboiny, G.: Whitelist-based spam filtering (Last accessed April 4, 2008), http://impressive.net/people/gerald/2000/12/spam-filtering.html

[13] Grok. Coromputer Dunno (Last accessed April 4, 2008), http://lists.grok.org.uk/pipermail/full-disclosure/attachments/20070911/87396911/attachment-0001.txt

[14] Honeynet Project. Q8 (Last accessed April 4, 2008), http://www.honeynet.org/papers/bots/

[15] Ioannidis, S., Keromytis, A., Bellovin, S., Smith, J.: Implementing a distributed firewall. In: ACM Conference on Computer and Communications Security (CCS), Athens, Greece (November 2000)

[16] Jiang, X., Wang, X.: Out-of-the-box monitoring of VM-based high-interaction honeypots. In: Kruegel, C., Lippmann, R., Clark, A. (eds.) RAID 2007. LNCS, vol. 4637, pp. 198–218. Springer, Heidelberg (2007)

[17] Jiang, X., Wang, X., Xu, D.: Stealthy malware detection through VMM-based 'out-of-the-box' semantic view. In: 14^{th} ACM Conference on Computer and Communications Security (CCS), Alexandria, VA (November 2007)

[18] LKCD Project. LKCD - Linux Kernel Crash Dump (Last accessed April 4, 2008), http://lkcd.sourceforge.net/

[19] Loscocco, P.A., Wilson, P.W., Pendergrass, J.A., McDonell, C.D.: Linux kernel integrity measurement using contextual inspection. In: 2^{nd} ACM Workshop on Scalable Trusted Computing (STC), Alexandria, VA (November 2007)

[20] McAfee. BackDoor-Rev.b. (Last accessed April 4, 2008), http://vil.nai.com/vil/Content/v_136510.htm
[21] Mogul, J., Rashid, R., Accetta, M.: The packet filter: An efficient mechanism for user-level network code. In: ACM Symposium on Operating Systems Principles (SOSP), Austin, TX (November 1987)
[22] Packet Storm (Last accessed April 4, 2008), http://packetstormsecurity.org/UNIX/penetration/rootkits/bdoor.c,blackhole.c,cheetah.c,server.c,ovas0n.c
[23] Packet Storm (Last accessed April 4, 2008), http://packetstormsecurity.org/0209-exploits/apache-ssl-bug.c,apache-linux.txt
[24] Packet Storm. Kaiten (Last accessed April 4, 2008), http://packetstormsecurity.org/irc/kaiten.c
[25] Payne, B.D., Carbone, M., Lee, W.: Secure and flexible monitoring of virtual machines. In: 23rd Annual Computer Security Applications Conference (ACSAC), Miami, FL (December 2007)
[26] Petroni Jr., N.L., Fraser, T., Walters, A., Arbaugh, W.A.: An architecture for specification-based detection of semantic integrity violations in kernel dynamic data. In: 15th USENIX Security Symposium, Vancouver, BC, Canada (August 2006)
[27] Petroni Jr., N.L., Hicks, M.: Automated detection of persistent kernel control-flow attacks. In: 14th ACM Conference on Computer and Communications Security (CCS), Alexandria, VA (November 2007)
[28] Ta-Min, R., Litty, L., Lie, D.: Splitting interfaces: Making trust between applications and operating systems configurable. In: Symposium on Operating System Design and Implementation (OSDI), Seattle, WA (October 2006)
[29] Venema, W.: TCP wrapper: Network monitoring, access control and booby traps. In: USENIX UNIX Security Symposium, Baltimore, MD (September 1992)
[30] Whitaker, A., Cox, R.S., Shaw, M., Gribble, S.D.: Constructing services with interposable virtual hardware. In: 1st Symposium on Networked Systems Design and Implementation (NSDI), San Francisco, CA (March 2004)
[31] XenAccess Project. XenAccess Library (Last accessed April 4, 2008), http://xenaccess.sourceforge.net/
[32] Xu, M., Jiang, X., Sandhu, R., Zhang, X.: Towards a VMM-based usage control framework for OS kernel integrity protection. In: 12th ACM Symposium on Access Control Models and Technologies (SACMAT), Sophia Antipolis, France (June 2007)
[33] Yin, H., Song, D., Egele, M., Kruegel, C., Kirda, E.: Panorama: Capturing system-wide information flow for malware detection and analysis. In: ACM Conference on Computer and Communications Security (CCS), Arlington, VA (October 2007)

A First Step towards Live Botmaster Traceback

Daniel Ramsbrock[1], Xinyuan Wang[1], and Xuxian Jiang[2]

[1] Department of Computer Science
George Mason University Fairfax, VA 22030, USA
{dramsbro,xwangc}@gmu.edu
[2] Department of Computer Science
North Carolina State University, Raleigh,
NC 27606, USA
jiang@cs.ncsu.edu

Abstract. Despite the increasing botnet threat, research in the area of botmaster traceback is limited. The four main obstacles are 1) the low-traffic nature of the bot-to-botmaster link; 2) chains of "stepping stones;" 3) the use of encryption along these chains; and 4) mixing with traffic from other bots. Most existing traceback approaches can address one or two of these issues, but no single approach can overcome all of them. We present a novel flow watermarking technique to address all four obstacles simultaneously. Our approach allows us to uniquely identify and trace any IRC-based botnet flow even if 1) it is encrypted (e.g., via SSL/TLS); 2) it passes multiple intermediate stepping stones (e.g., IRC server, SOCKs); and 3) it is mixed with other botnet traffic. Our watermarking scheme relies on adding padding characters to outgoing botnet C&C messages at the application layer. This produces specific differences in lengths between randomly chosen pairs of messages in a network flow. As a result, our watermarking technique can be used to trace any interactive botnet C&C traffic and it only requires a few dozen packets to be effective. To the best of our knowledge, this is the first approach that has the potential to allow real-time botmaster traceback across the Internet.

We have empirically validated the effectiveness of our botnet flow watermarking approach with live experiments on PlanetLab nodes and public IRC servers on different continents. We achieved virtually a 100% detection rate of watermarked (encrypted and unencrypted) IRC traffic with a false positive rate on the order of 10^{-5}. Due to the message queuing and throttling functionality of IRC servers, mixing chaff with the watermarked flow does not significantly impact the effectiveness of our watermarking approach.

1 Introduction

Botnets are currently one of the most serious threats to computers connected to the Internet. Recent media coverage has revealed many large-scale botnets worldwide. One botnet [22,23] has reportedly compromised and controlled over 400,000 computers – including computers at the Weapons Division of the U.S. Naval Air Warfare Center, U.S. Department of Defense Information Systems

Agency. Another recently discovered botnet is suspected to have controlled 1.5 million computers around the globe [9]. It has been estimated [20] that more than 5 percent of all computers connected to the Internet have been compromised and used as bots. Currently, botnets are responsible for most spam, adware, spyware, phishing, identity theft, online fraud and DDoS attacks on the Internet.

The botnet problem has recently received significant attention from the research community. Most existing work on botnet defense [1,2,3,6,11,14,15,18] has focused on the detection and removal of command and control (C&C) servers and individual bots. While such a capability is a useful start in mitigating the botnet problem, it does not address the root cause: the botmaster. For example, existing botnet defense mechanisms can detect and dismantle botnets, but they usually cannot determine the identity and location of the botmaster. As a result, the botmaster is free to create and operate another botnet by compromising other vulnerable hosts. Botmasters can currently operate with impunity due to a lack of reliable traceback mechanisms. However, if the botmaster's risk of being caught is increased, he would be hesitant to create and operate botnets. Therefore, even an imperfect botmaster traceback capability could effectively deter botmasters. Unfortunately, current botmasters have all the potential gains from operating botnets with minimum risk of being caught. Therefore, the botnet problem cannot be solved until we develop a reliable method for identifying and locating botmasters across the Internet. This paper presents a substantial first step towards achieving the goal of botmaster traceback.

Tracking and locating the botmaster of a discovered botnet is very challenging. First, the botmaster only needs to be online briefly to issue commands or check the bots' status. As a result, any botmaster traceback has to occur in real-time. Second, the botmaster usually does not directly connect to the botnet C&C server and he can easily launder his connection through various stepping stones. Third, the botmaster can protect his C&C traffic with strong encryption. For example, Agobot has built-in SSL/TLS support. Finally, the C&C traffic from the botmaster is typically low-volume. As a result, a successful botmaster traceback approach must be effective on low-volume, encrypted traffic across multiple stepping stones.

To the best of our knowledge, no existing traceback methods can effectively track a botmaster across the Internet in real-time. For example, methods [33, 32, 8, 31, 4, 29, 30] have been shown to be able to trace encrypted traffic across various stepping stones and proxies, but they need a large amount of traffic (at least hundreds of packets) to be effective. During a typical session, each bot exchanges only a few dozen packets with the botmaster. Due to this low traffic volume, the above techniques are not suitable for botmaster traceback.

In this paper, we address the botmaster traceback problem with a novel packet flow watermarking technique. Our goal is to develop a practical solution that can be used to trace low-volume botnet C&C traffic in real-time even if it is encrypted and laundered through multiple intermediate hosts (e.g., IRC servers, stepping stones, proxies). We assume that the tracer has control of a single rogue bot in the target botnet, and this bot can send messages in response to a the query from the botmaster. To trace the response traffic back to the botmaster, the rogue bot transparently injects a unique watermark into its response. If the injected watermark can survive the various transformations (e.g., encryption/decryption, proxying) of the botnet C&C traffic, we can trace the watermark and locate the botmaster via monitoring nodes across the Internet. To embed the watermark, we adjust the lengths of randomly selected pairs of packets such that the length difference between each packet pair will fall within a certain range. To track encrypted botnet traffic that mixes messages from multiple bots, we developed a hybrid length-timing watermarking method. Compared to previous approaches [31,29,30], our two proposed methods require far less traffic volume to encode high-entropy watermarks. We empirically validated the effectiveness of our watermarking algorithms using real-time experiments on live IRC traffic through PlanetLab nodes and public IRC servers across different continents. Both of our watermarking approaches achieved a virtually 100% watermark detection rate and a 10^{-5} false positive rate with only a few dozen packets. To the best of our knowledge, this is the first approach that has the potential to allow real-time botmaster traceback across the Internet.

The remainder of the paper is structured as follows: Section 2 introduces the botmaster traceback model. Section 3 presents the design and analysis of our flow watermarking schemes. Section 4 describes our experiments and their results, while section 5 discusses limitations and future work. Finally, Section 6 surveys related literature and Section 7 summarizes our findings.

2 Botmaster Traceback Model

According to [17,21,28], most botnets currently in the wild are IRC-based. Therefore, we will focus on tracing the botmaster in the context of IRC-based botnets. Nevertheless, our flow watermarking trace approach is applicable to any interactive botnet traffic.

2.1 Botnets and Stepping Stones

Bots have been covered extensively in the existing literature, for example [2,6,7,16,21] provide good overviews. The typical bot lifecycle starts with exploitation, followed by download and installation of the bot software. At this point, the bot contacts the central C&C server run by the botmaster, where he can execute commands and receive responses from his botnet.

Botmasters rarely connect directly to their C&C servers since this would reveal their true IP address and approximate location. Instead, they use a chain of stepping stone proxies that anonymously relay traffic. Popular proxy software used for this purpose is SSH, SOCKS, and IRC BNCs (such as psyBNC). Since the stepping stones are controlled by the attacker, they do not have an audit trail in place or other means of tracing the true source of traffic. However, there are two properties of stepping stones that can be exploited for tracing purposes: 1) the content of the message (the application-layer payload) is never modified and 2) messages are passed on immediately due to the interactive nature of IRC. Consequently, the relative lengths of messages and their timings are preserved, even if encryption is used. In the case of encryption, the message lengths are rounded up to the nearest multiple of the block size. This inherent length and timing preservation is the foundation of our botmaster traceback approach.

2.2 Tracking the Botmaster by Watermarking Botnet Traffic

Our botmaster traceback approach exploits the fact that the communication between the IRC-based bots and the botmaster is bidirectional and interactive. Whenever the botmaster issues commands to a bot, the response traffic will eventually return to the botmaster after being laundered and possibly transformed. Therefore, if we can watermark the response traffic from a bot to the botmaster, we can eventually trace and locate the botmaster. Since the response traffic we are tracking may be mixed with other IRC traffic, we need to be able to isolate the target traffic. With unencrypted traffic, this can be achieved by content inspection, but encrypted traffic presents a challenge which we address with our hybrid length-timing algorithm.

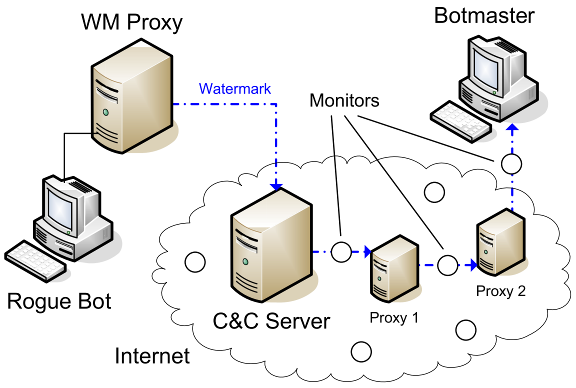


Fig. 1. Botmaster traceback by watermarking the botnet response traffic

Figure 1 shows the overall watermarking traceback model. We assume that we control a rogue bot, which could be a honeypot host that has been compromised and has joined a botnet. The rogue bot watermarks its outgoing PRIVMSG traffic in response to commands from the botmaster. As with any traceback approach, our watermark tracing scheme needs support from the network. Specifically, we assume there are cooperating monitor nodes across the Internet, which will inspect the passing traffic for the specified watermark and report back to us whenever they find it. Note that our approach does not require a global monitoring capability. If there are uncooperative or unmonitored areas, we would lose one or more links along the traceback path. However, we can pick up the trail again once the watermarked traffic re-enters a monitored area. In general, this appears to be the best possible approach in the absence of a global monitoring capability. We assume that the tracer can securely share the desired watermark with all monitor nodes prior to sending the watermarked traffic. This enables the monitors to report ‘sightings’ of the watermark in real-time and requires only a single watermarked flow to complete the trace.

3 Length-Based Watermarking Scheme

Our watermarking scheme was specifically designed for a low-traffic, text-based channel such as the one between a bot and its botmaster. This section describes the design and analysis of both the length-only (unencrypted traffic) and the length-timing hybrid algorithms (encrypted traffic). We describe the encoding and decoding formulas for both algorithms and address the issue of false positives and false negatives.

The terms ‘message’ and ‘packet’ are used interchangeably since a typical botnet C&C message is usually small (less than 512 bytes) and fits into a single packet).

3.1 Basic Length-Based Watermarking Scheme

Watermark Bit Encoding. Given a packet flow f of n packets $P_1, \ldots, P_n$, we want to encode an l-bit watermark $W = w_0, \ldots, w_{l-1}$ using $2l \leq n$ packets. We first use a pseudo-random number generator (PRNG) with seed s to randomly choose $2l$ distinct packets from $P_1, \ldots, P_n$, we then use them to randomly form l packet pairs: $\langle P_{r_i},\ P_{e_i} \rangle$ $(i = 0, \ldots, l-1)$ such that $r_i \leq e_i$. We call packet P_{r_i} a *reference packet* and packet P_{e_i} an *encoding packet*. We further use the PRNG to randomly assign watermark bit w_k $(0 \leq k \leq l-1)$ to packet pair $\langle P_{r_i},\ P_{e_i} \rangle$, and we use $\langle r_i, e_i, k \rangle$ to represent that packet pair $\langle P_{r_i},\ P_{e_i} \rangle$ is assigned to encode watermark bit w_k.

To encode the watermark bit w_k into packet pair $\langle P_{r_i},\ P_{e_i} \rangle$, we modify the length of the encoding packet P_{e_i} by adding padding characters to achieve a specific length difference to its corresponding reference packet P_{r_i}. The padding characters could be invisible (such as whitespace) or visible characters and they can be inserted in random locations within the message. This would make it difficult for the adversary to detect the existence of the padding. Let l_e and l_r be the packet lengths of the watermark encoding and reference packets respectively, $Z = l_e - l_r$ be the length difference, and $L > 0$ be the bucket size. We define the *watermark bit encoding function* as

$$e(l_r, l_e, L, w) = l_e + [(0.5 + w)L - (l_e - l_r)] \bmod 2L \tag{1}$$

which returns the increased length of watermark encoding packet given the length of the reference packet l_r, the length of the encoding packet l_e, the bucket size L, and the watermark bit to be encoded w.

Therefore,

$$(e(l_r, l_e, L, w) - l_r) \bmod 2L \tag{2}$$

$$\begin{aligned}
&= \{(l_e - l_r) + [(0.5 + w)L - (l_e - l_r)] \bmod 2L\} \bmod 2L \\
&= \{(0.5 + w)L\} \bmod 2L \\
&= (w + 0.5)L
\end{aligned}$$

This indicates that the packet length difference $Z = l_e - l_r$, after l_e is adjusted by the watermark bit encoding function $e(l_r, l_e, L, w)$, falls within the middle of either an even or odd numbered bucket depending on whether the watermark bit w is even or odd.

Watermark Bit Decoding. Assuming the decoder knows the watermarking parameters: PRNG, s, n, l, W and L, the watermark decoder can obtain the exact pseudo-random mapping $\langle r_i, e_i, k\rangle$ as that used by the watermark encoder. We use the following *watermark bit decoding function* to decode watermark bit w_k from the packet lengths of packets P_{r_i} and P_{e_i}

$$d(l_r, l_e, L) = \lfloor \frac{l_e - l_r}{L} \rfloor \bmod 2 \tag{3}$$

The equation below proves that any watermark bit w encoded by the encoding function defined in equation (1) will be correctly decoded by the decoding function defined in equation (3).

$$\begin{aligned}
&d(l_r, e(l_r, l_e, L, w), L) \qquad (4) \\
&= \lfloor \frac{e(l_r, l_e, L, w) - l_r}{L} \rfloor \bmod 2 \\
&= \lfloor \frac{(l_e - l_r) \bmod 2L + [(0.5 + w)L - (l_e - l_r)] \bmod 2L}{L} \rfloor \bmod 2 \\
&= \lfloor \frac{(0.5 + w)L}{L} \rfloor \bmod 2 \\
&= w
\end{aligned}$$

Assume the lengths of packets P_r and P_e (l_r and l_e) have been increased for $x_r \geq 0$ and $x_e \geq 0$ bytes respectively when they are transmitted over the network (e.g., due to padding of encryption), then $x_e - x_r$ is the distortion over the packet length difference $l_e - l_r$. Then the decoding with such distortion is

$$\begin{aligned}
&d(l_r + x_r, e(l_r, l_e, L, w) + x_e, L) \qquad (5) \\
&= \lfloor \frac{e(l_r, l_e, L, w) - l_r + (x_e - x_r)}{L} \rfloor \bmod 2 \\
&= w + \lfloor 0.5 + \frac{x_e - x_r}{L} \rfloor \bmod 2
\end{aligned}$$

Therefore, the decoding with distortion will be correct if and only if

$$(-0.5 + 2i)L \leq x_e - x_r < (0.5 + 2i)L \tag{6}$$

Specifically, when the magnitude of the distortion $|x_e - x_r| < 0.5L$, the decoding is guaranteed to be correct.

Watermark Decoding and Error Tolerance. Given a packet flow f and appropriate watermarking parameters (PRNG, s, n, l, W and L) used by the watermark encoder, the watermark decoder can obtain a l-bit decoded watermark W' using the watermark bit decoding function defined in equation (3). Due to potential distortion of the packet lengths in the packet flow f, the decoded W' could have a few bits different from the encoded watermark W. We introduce a Hamming distance threshold $h \geq 0$ to accommodate such partial corruption of the embedded watermark. Specifically, we will consider that packet flow f contains watermark W if the Hamming distance between W and W': $H(W, W')$ is no bigger than h.

Watermark Collision Probability (False Positive Rate). No matter what watermark W and Hamming distance threshold h we choose, there is always a non-zero possibility that the decoding W' of a random unwatermarked flow happens to have no more than h Hamming distance to the random watermark W we have chosen. In other words, watermark W is reported found in an unwatermarked flow; we refer to this case as a *watermark collision.*

Intuitively, the longer the watermark and the smaller the Hamming distance threshold, the smaller the probability of a watermark collision. Assume we have randomly chosen a l-bit watermark, and we are decoding l-bits from random unwatermarked flows. Any particular bit decoded from a random unwatermarked flow should have 0.5 probability to match the corresponding bit of the random watermark we have chosen. Therefore, the collision probability of l-bit watermark from random unwatermarked flows with Hamming distance threshold h is

$$\sum_{i=0}^{h} \binom{l}{i} (\frac{1}{2})^l \tag{7}$$

We have empirically validated the watermark collision probability distribution with the following experiment. We first use a PRNG and a random seed number s to generate 32 packet pairs $\langle r_i, e_i \rangle$ and pseudo-randomly assign each bit of a 32-bit watermark W to the 32 packet pairs, we then encode the 32 bit watermark W into a random packet flow f. Now we try to decode the watermarked flow f' with 1,000 wrong seed numbers. Given the pseudo-random nature of our selection of the packet pairs, decoding a watermarked flow with the wrong seed is equivalent of decoding an unwatermarked flow, which can be used to measure the watermark collision probability.

The left side of Figure 2 illustrates the number of matched bits from the decoding with each of the 1,000 wrong seed numbers. It shows that the numbers of matched bits are centered around the expected value of 16 bits, which is half of the watermark length. Based on these results and the experimental data in Section 4.2, we can choose a Hamming distance threshold of $h = 4$ (28 bits) as shown on the graph, yielding an expected false positive rate (FPR) of 9.64×10^{-6} according to equation (7). The right side of Figure 2 shows the distributions of the measured and the expected number of matched bits. It illustrates that the distribution of the measured number of matched bits is close to the expected binomial distribution with $p = 0.5$ and $n = 32$.

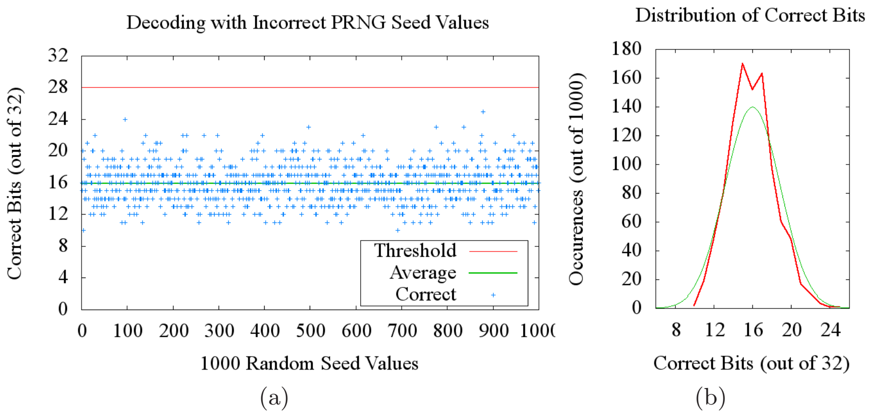


Fig. 2. 32-bit watermark collision probability and distribution

Watermark Loss (False Negative). Our length-only encoding scheme (without the hybrid timing approach) is highly sensitive to having the correct sequence of messages. If any messages are added or deleted in transit, the watermark will be lost in that flow. However, the chance of this happening is very remote since the encoding takes place at the application layer, on top of TCP. By its nature, TCP guarantees in-order delivery of all packets and their contents, so a non-intentional watermark loss is very unlikely.

In the case of active countermeasures, our scheme can tolerate distortion as long as $|x_e - x_r| < 0.5L$, as described by inequality (6). This property is the result of aiming for the center of each bucket when encoding. However, if an active adversary drops, adds, or reorders messages, the watermark will be lost unless additional redundancy is in place or the length-timing algorithm is used.

3.2 Hybrid Length-Timing Watermarking for Encrypted Traffic

By their nature, IRC-based botnets have many bots on one channel at once, many of them joining, parting, or sending data to the botmaster simultaneously. In this case, the watermarked messages from our rogue bot will be mixed with unwatermarked messages from other bots. We call these unwatermarked messages from others *chaff* messages. In order to reliably decode the embedded watermark, we need to filter out chaff messages as much as possible.

When the C&C traffic is unencrypted, it is easy for the watermark decoder to filter out chaff based on the sender nicks in the messages. However, if the traffic is encrypted (e.g., using SSL/TLS), we cannot rely on content inspection to identify chaff messages. To address this new challenge in filtering out chaff, we propose to use another dimension of information – the packet timing – to filter out chaff.

The basic idea is to send the watermark encoding packets at a specific time (e.g., t_i). Assuming the network jitter δ is limited, we can narrow the range of potential packets used for decoding to $[t_{e_i} - \frac{\delta}{2}, t_{e_i} + \frac{\delta}{2}]$. If $\delta > 0$ is small, then the chances that some chaff packet happens to fall within the range $[t_{e_i} - \frac{\delta}{2}, t_{e_i} + \frac{\delta}{2}]$ is small. This means we can decode the watermark correctly even if there are substantial encrypted chaff packets.

Watermark Encoding. The watermark bit encoding process is exactly the same as that of the basic length-based watermarking scheme. The difference is that now we send out each watermarked packet P_{e_i} at a precise time. Specifically, we use the watermark bit encoding function defined in equation (1) to adjust the length of the watermark encoding packet P_{e_i}. We use a pseudo-random number generator PRNG and seed s_t to generate the random time t_{e_i} at which P_{e_i} will be sent out.

An implicit requirement for the hybrid length-timing watermarking scheme is that we need to know when each watermark encoding packet P_{e_i} will be available. In our watermark tracing model, the tracer owns a rogue bot who can determine what to send out and when to send it. Since we have full control over the outgoing traffic, we can use the hybrid length-timing scheme to watermark the traffic in real-time.

Watermark Decoding. When we decode the encrypted botnet traffic, we do not know which packet is a watermark encoding packet P_{e_i}. However, given the PRNG and s_t we do know the approximate time t_{e_i} at which the watermark encoding packet P_{e_i} should arrive. We then use all packets in the time interval $[t_{e_i} - \frac{\delta}{2}, t_{e_i} + \frac{\delta}{2}]$ to decode. Specifically, we use the sum of the lengths of all the packets in the time interval $[t_{e_i} - \frac{\delta}{2}, t_{e_i} + \frac{\delta}{2}]$ as the length of the watermark encoding packet and apply that to the watermark bit decoding function (3).

Due to network delay jitter and/or active timing perturbation by the adversary, the exact arrival time of watermark encoding packet P_{e_i} may be different from t_{e_i}. Fortunately, the decoding can self-synchronize with the encoding by leveraging an intrinsic property of our hybrid length-timing watermarking scheme. Specifically, if the decoding of a watermarked flow uses the wrong offset or wrong seeds (s and s_t), then the decoded l-bit watermark W' will almost always have about $\frac{l}{2}$ bits matched with the true watermark W. This gives us an easy way to determine if we are using the correct offset, and we can try a range of possible offsets and pick the best decoding result.

4 Implementation and Experiment

To validate the practicality of our watermarking scheme, we implemented both the length-only algorithm (unencrypted traffic) and the length-timing hybrid algorithm (encrypted traffic). To let our watermarking proxy interact with a realistic but benign IRC bot, we obtained a sanitized version of Agobot from its source code, containing only benign IRC communication features. We ran the sanitized Agobot on a local machine to generate benign IRC traffic to test the effectiveness of our watermarking scheme across public IRC servers and PlanetLab nodes. At no time did we send malicious traffic to anyone in the course of our experiments.

4.1 Length-Only Algorithm (Unencrypted Traffic)

We implemented the length-only algorithm in a modified open-source IRC proxy server and ran a series of experiments using the sanitized Agobot and public Internet IRC servers. We were able to recover the watermark successfully from unencrypted traffic in all ten of our trials.

Modified IRC Bouncer. To achieve greater flexibility, we added our watermarking functionality to an existing IRC bouncer (BNC) package, psyBNC. Having the watermarking implemented on a proxy server allows us to use it on all bots conforming to the standard IRC protocol. It eliminates the need to have access to a bot's source code to add the watermarking functionality: outgoing traffic is modified by the BNC after the bot sends it.

In order for psyBNC to act as a transparent proxy, it needs to be configured identically to the bot. The information required consists of the C&C server's hostname, the port, and an IRC nick consistent with the bot's naming scheme. This information can be gathered by running the bot and monitoring the outgoing network traffic. In order to trick the bot into connecting to the BNC rather than to the real C&C host, we also need to update our local DNS cache so that a lookup of the C&C server's hostname resolves to the IP of our BNC.

Once it has been configured with this information, the BNC is completely transparent to the bot: when it starts up, the bot is automatically signed into the real C&C server by the BNC. The bot now joins the botnet channel as if it were directly connected and then waits for the botmaster's instructions. All PRIVMSG traffic from the bot to the C&C server (and by extension, to the botmaster) is watermarked by the transparent BNC in between.

Experiment and Results. To test our watermarking scheme, we devised an experiment that emulates the conditions of an Internet-wide botnet as closely as possible. To simulate the botmaster and stepping stones, we used PlanetLab nodes in California and Germany. We used a live, public IRC server in Arizona to act as a C&C host, creating a uniquely-named channel for our experiments. Our channel consisted of two IRC users: the Test Bot was running a copy of the sanitized Agobot and the Botmaster was acting as the botmaster (see Figure 3). As the diagram indicates, all traffic sent by the Test Bot passes through the psyBNC server (WM Proxy) where the watermark is injected. The distances involved in this setup are considerable: the watermarked traffic traverses literally half the globe (12 time zones) before reaching its ultimate destination in Germany, with a combined round-trip time of 292 milliseconds on average (at the time of our experiment).

The objective is to be able to decode the full watermark in the traffic captured at the Stepping Stone and Botmaster. Since only PRIVMSG traffic from the Test Bot is watermarked, all other traffic (chaff) must be filtered out before decoding. Most of this chaff consists of messages from other users on the channel, PING/PONG exchanges, and JOIN/PART notifications from the channel. There could be additional chaff on the same connection if the botmaster is logged into multiple channels on the same IRC server. However, filtering out the chaff is

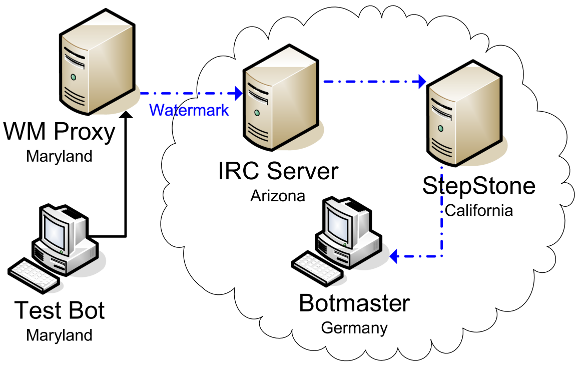


Fig. 3. Experiment setup for unencrypted traffic

trivial in the absence of encryption since all IRC messages contain the sender's nick. Therefore, we can easily isolate the watermarked packets based on the Test Bot's nick.

During our experiments, the psyBNC proxy was configured to inject a 32-bit watermark into a 64-packet stream. To generate traffic from the Test Bot, the Botmaster logged in and issued the `commands.list` command, causing the bot to send a list of all valid bot commands and their descriptions. We captured all traffic leaving the WM Proxy, arriving at the Stepping Stone, and arriving at the Botmaster. In ten trials with different (random) 32-bit watermarks, we were able to correct decode the full 32-bit watermark at all three monitoring locations: the WM Proxy in Maryland, the Stepping Stone in California, and Botmaster in Germany.

4.2 Hybrid Length-Timing Algorithm (Encrypted Traffic)

To test the hybrid length-timing algorithm, we implemented a simple IRC bot that sends length-watermarked messages out at specific intervals. We used a "chaff bot" on the channel to generate controlled amounts of chaff. We were able to recover the watermark with a high success rate, even when high amounts of chaff were present.

Hybrid Length-Timing Encoder. We implemented the hybrid encoding algorithm as a Perl program which reads in a previously length-only watermarked stream of messages and sends them out at specific times. To achieve highly precise timing, we used the `Time::HiRes` Perl package, which provides microsecond-resolution timers. At startup, the program uses the Mersenne Twister PRNG (via the `Math::Random::MT` package) to generate a list of departure times for all messages to be sent. Each message is sent at a randomly chosen time between 2 and 2.35 seconds after the previous message. The 2-second minimum spacing avoids IRC server packet throttling (more details are discussed in Section 4.2).

Hybrid Length-Timing Decoder. The hybrid decoding script was also written in Perl, relying on the PCAP library to provide a standardized network traffic

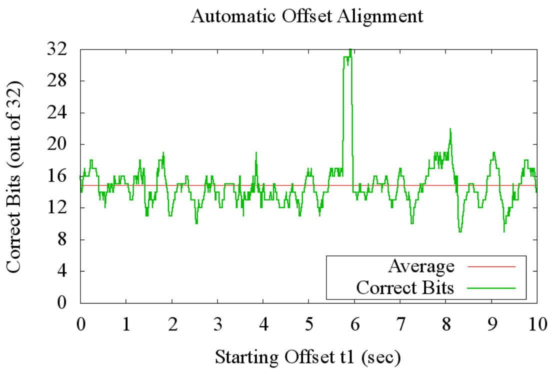


Fig. 4. Offset Self-Synchronization via Offset Sliding-Window

capture mechanism (via the `Net::Pcap` module). The program reads in a stream of packets (either from a live interface or from a PCAP file), then performs a sliding-window offset self-synchronization process to determine the time $t1$ of the first watermarked packet. To find the correct $t1$, the program steps through a range of possible values determined by the `offset`, `max`, and `step` parameters. It starts with $t1 =$`offset`, incrementing $t1$ by `step` until $t1 =$`(offset + max)`. It decodes the full watermark sequence for each $t1$, recording the number of bits matching the sought watermark W. It then chooses the $t1$ that produced the highest number of matching bits. If there are multiple $t1$ values resulting in the same number of matching bits, it uses the lowest value for $t1$. Figure 4 illustrates the synchronization process, showing that the correct $t1$ is near 6 seconds: 5.92 sec has 32 correct bits. For all incorrect $t1$ values, the decoding rate was significantly lower, averaging 14.84 correct bits.

Experiment and Results. The experiment setup in this case was similar to the unencrypted experiment described in Section 4.1. The three main differences were: 1) a single Source computer producing watermarked traffic on its own replaced the Test Bot and WM Proxy; 2) the connection between the Botmaster and the IRC server (via StepStone) was encrypted using SSL/TLS; and 3) we used a different IRC server because the one in Arizona does not support SSL/TLS connections. The IRC server in this case happens to be located in Germany, but not in the same place as the Botmaster. Please refer to Figure 5 for the full experiment setup. In this configuration, the distances involved are even greater, with the watermarked traffic traversing the equivalent of the entire globe (24 time zones). The combined round-trip time from Source to Botmaster was 482 milliseconds (on average) at the time of our experiment.

To handle encryption, the parameters for the length-only algorithm were adjusted to ensure that the bucket size matched or exceeded the encryption block size. Most SSL/TLS connections use a block size of 128 bits (16 bytes), though 192 and 256 bits are also common. To ensure that each added bucket also causes another encrypted block to be added to the message, the bucket size has to be greater than or equal to the block size. For our experiment, we used a bucket size of 16 bytes, which was sufficient for the 128-bit block size used in the SSL/TLS

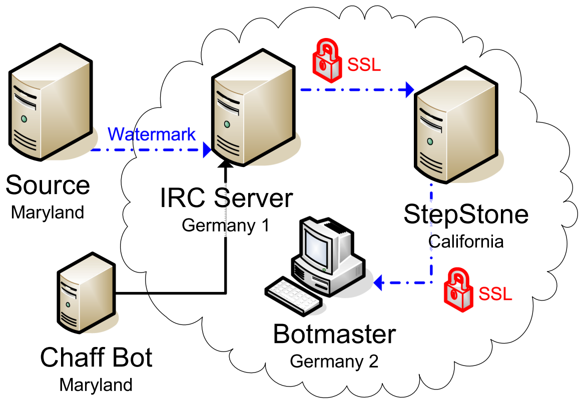


Fig. 5. Experiment setup for encrypted traffic

connection. For compatibility with the larger block sizes (192 and 256 bits), a bucket size of 32 bytes can be used.

For the experiments, the Source produced a stream of 64 packets, containing a randomly generated 32-bit watermark. The Chaff Bot produced a controlled amount of background traffic, spacing the packets at random intervals between 1 and 6 seconds (at least 1 second to avoid throttling). In addition to our Control run (no chaff), we ran five different chaff levels (Chaff 1 to 5). The number refers to the maximum time between packets (not including the minimum 1-second spacing). For example, for the Chaff 1 run, packets were sent at a random time between 1 and 2 seconds. Thus, one packet was sent on average every 1.5 seconds, resulting in a chaff rate of approximately $1/1.5 = 0.667$ packets/sec.

We captured network traffic in three places: 1) traffic from Source and Chaff Bot to IRC Server; 2) traffic arriving at StepStone from IRC Server; and 3) traffic arriving at Botmaster from StepStone. Traffic in all three locations includes both watermark and chaff packets. We decoded the traffic at each location, recording the number of matching bits. For decoding, we used a value of 200 milliseconds for the timing window size δ and a sliding offset range from 0 to 10 seconds. This δ value was large enough to account for possible jitter along the stepping stone chain but small enough to make it unlikely that a chaff packet appears within δ of an encoding packet. We also measured the actual chaff rate based on the departure times of each chaff packet, and these were very close to the expected rates based on an even distribution of random departure times. We repeated this process three times for each chaff level, resulting in a total of 18 runs. Our experiment results are summarized in Table 1, with each column representing the average values from three trials.

We had near-perfect decoding along the stepping-stone chain for all chaff rates of 0.5 packets/sec and below. Only when the chaff rate rose above 0.5 packets/sec did the chaff start having a slight impact, bringing the decoding rate down to an average of 31 bits. The overall average decoding rate at the StepStone and Botmaster was 31.69 bits, or 99.05 percent. The lowest recorded decoding rate

Table 1. Experiment results for encrypted traffic: Recovered watermark bits (out of 32) at each monitoring station along the watermark's path (averaged from three trials)

Monitoring Location	**Chaff 1**	**Chaff 2**	**Chaff 3**	**Chaff 4**	**Chaff 5**	**Control**
Chaff Rate (packets/sec)	0.6719	0.4976	0.4274	0.3236	0.2872	no chaff
Source - Maryland	29.67	30.33	29.67	30.33	30.33	32
StepStone - California	31	32	31.67	31.67	32	32
Botmaster - Germany	31	31.67	32	31.67	31.67	32

during our experiments was 28 bits, so we can use a Hamming distance threshold of $h = 4$ to obtain a 100 percent true positive rate (TPR) and a false positive rate (FPR) of 9.64×10^{-6}.

The most surprising result is that in all cases where chaff was present, the decoding rate was worse at the Source than downstream at the StepStone and Botmaster. After examining the network traces in detail, we realized that this behavior was due to the presence of traffic queuing and throttling on the IRC Server. To avoid flooding, IRC servers are configured to enforce minimum packet spacings, and most will throttle traffic at 0.5 to 1 packets/sec. To confirm this behavior, we sent packets to the IRC Server in Germany at random intervals of 100 to 300 milliseconds. For the first 5 seconds, packets were passed on immediately, but after that the throttling kicked in, limiting the server's outgoing rate to 1 packet/sec. After about 2 minutes, the server's packet queue became full with backlogged packets, and it disconnected our client. Figure 6 illustrates the effect of throttling on the packet arrival times, including the 5-second "grace period" at the beginning.

In the context of our hybrid encoding scheme, IRC message queuing is highly beneficial because it dramatically reduces the chances that chaff and encoding packets will appear close to each other. At the Source, packets appear at the exact intervals they are sent, which could be less than δ and therefore affect decoding. However, this interval will be increased due to queuing by the IRC server. By the time the packets reach the StepStone and Botmaster, they no longer affect decoding because they are more than δ apart. In our experiments, we observed

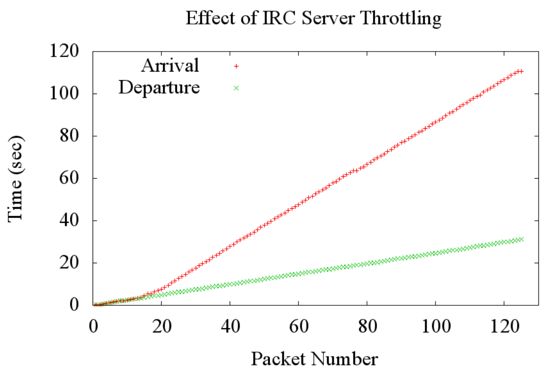


Fig. 6. IRC server throttling causes packets to be spaced apart further upon arrival

that the IRC server introduced a distance of at about 130 milliseconds between packets due to queuing. Since our δ value was 200 milliseconds, this made it unlikely that two packets would arrive in the same slot.

5 Discussion and Future Work

Our experiments show that our watermarking scheme is effective in tracing the botmaster of IRC-based botnets, which are still the predominant type in the wild [17,21,28]. Our watermark can be recovered with a high degree of accuracy even when the watermarked botnet C&C traffic is encrypted across multiple stepping stones and mixed with other flows.

In theory, our flow watermarking technique could be applied to trace any real-time and interactive botnet C&C traffic. Therefore, it could be used to track the botmaster of peer-to-peer (P2P) botnets which have started appearing recently [13]. However, HTTP-based botnets present a much higher level of traceback difficulty: the messages do not get passed from the bot to the botmaster in real-time. They are typically stored on the C&C server until the botmaster retrieves them in bulk, usually over an encrypted connection such as SSH. Due to this, any approach that relies on properties of individual packets (such as length and timing) will be unsuccessful.

When SSH is used as the final hop in a chain of stepping stones, it presents unique challenges. In this case, the botmaster uses SSH to log into a stepping stone, launches a commandline-based IRC client on that host, and uses this IRC client to connect to his botnet (possibly via more stepping stones). In this capacity, SSH is not acting as a proxy, passing on messages verbatim like psyBNC or SOCKS. Instead, it transfers the "graphical" screen updates of the running IRC client, which is not necessarily correlated to the incoming IRC messages. This situation is challenging for our approach because the application-layer content is transformed, altering the relative lengths of packets. We are working on this problem, but we have been unable to explore it in detail. Notice that if SSH is used in a tunnelling capacity (such as port forwarding or a SOCKS proxy) in the middle of a stepping stone chain, this limitation does not apply.

Once the botmaster become aware of the flow watermarking tracing approach, he may want to corrupt the embedded watermark from intermediate stepping stones. However, since the padding characters could be almost any character and they are inserted randomly in the botnet message, it would be difficult for any intermediate stepping stone to identify and remove the padding characters without knowing the original unwatermarked message. The botmaster may be able to detect and identify the padding if he knows exactly what he is expecting for. However, once he receives the watermarked message, the watermarked message has already left the complete trail toward the botmaster. The botmaster could have intermediate stepping stones to perturb the length of the passing botnet messages by adding random padding such as white space. Since the watermark is embedded in the length difference between randomly chosen packets, the negative impact of the padding by the adversary tends to cancel each other. We can further mitigate the negative impact by using redundant pairs of packets to encode the watermark. However, this would increase the number of packets needed. So this is essentially a tradeoff between the robustness and the efficiency.

As previously discussed in Section 2.2, our approach requires at least partial network coverage of distributed monitoring stations. This is a common requirement for network traceback approaches, especially since the coverage does not need to be global. The accuracy of the trace is directly proportional to the number and placement of monitoring nodes.

Our work is a significant step in the direction of live botmaster traceback, but as the title implies, it is indeed a first step. Our future work in this area includes the exploration of several topics, including optimal deployment of monitoring nodes, SSH traffic on the last hop, further data collection with longer stepping stone chains, and traceback experiments on in-the-wild botnets.

6 Related Work

The botnet research field is relatively new, but many papers have been published in the last few years as the botnet threat has accelerated. As one of the first in the botnet arena, the Honeynet Project [1] provided a starting point for future exploration of the problem. A comprehensive study at Johns Hopkins University [21] constructed a honeypot-based framework for acquiring and analyzing bot binaries. The framework can automatically generate rogue bots (drones) to actively infiltrate botnets, which is the first step in injecting a watermark and tracing the botmaster.

Most early botnet work focused on defining, understanding, and classifying botnets. Some examples are papers by Cooke et al. [6], Dagon et al. [7], Ianelli and Hackworth [17], Barford and Yegneswaran [2], and Holz's summary in *Security & Privacy* [16]. Since then, bot detection has become more of a focal point and many techniques have been proposed. Binkley and Singh [3] presented an anomaly-based detection algorithm for IRC-based botnets. Goebel and Holz [11] reported success with their Rishi tool, which evaluates IRC nicknames for likely botnet membership. Karasaridis et al. [18] described an ISP-level algorithm for detecting botnet traffic based on analysis of transport-layer summary statistics. Gu et al. [15] detailed their BotHunter approach, which is based on IDS dialog correlation techniques. They also published a related paper in 2008 [14] where they introduce BotSniffer, a tool for detecting C&C traffic in network traces.

Despite a large amount of literature regarding botnet detection and removal, relatively little work has been done on finding and eliminating the root cause: the botmaster himself. An earlier paper by Freiling et al. [10] describes a manual method of infiltrating a botnet and attempting to locate the botmaster, but the approach does not scale well due to lack of automation.

In the general traceback field, there are two main areas of interest: 1) network-layer (IP) traceback and 2) tracing approaches resilient to stepping stones. The advent of the first category dates back to the era of fast-spreading worms, when no stepping stones were used and IP-level traceback was sufficient. A leading paper in this area is Savage et al. [25], which introduced the probabilistic packet marking technique, embedding tracing information an IP header field. Two years later, Goodrich [12] expounded on this approach, introducing "randomize-and-link" with better scalability. A different technique for IP traceback is the log/hash-based one introduced by Snoeren et al. [26], and enhanced by Li et al. [19].

There are a number of works on how to trace attack traffic across stepping stones under various conditions. For example, [33,34,8,32,31,4,29,30] used inter-packet timing to correlate encrypted traffic across the stepping stones and/or low-latency anonymity systems. Most timing-based correlation schemes are passive, with the exception of the three active methods [31,29,30]. Our proposed method is based on the same active watermarking principle used in these three works. However, our method differs from them in that it uses the packet length, in addition to the packet timing, to encode the watermark. As a result, our method requires much fewer packets than methods [31,29,30] to be effective.

7 Conclusion

The key contribution of our work is that it addresses the four major obstacles in botmaster traceback: 1) stepping stones, 2) encryption, 3) flow mixing and 4) a low traffic volume between bot and botmaster. Our watermarking traceback approach is resilient to stepping stones and encryption, and it requires only a small number of packets in order to embed a high-entropy watermark into a network flow. The watermarked flow can be tracked even when it has been mixed with randomized chaff traffic. Due to these characteristics, our approach is uniquely suited for real-time tracing of the interactive, low-traffic botnet C&C communication between a bot and its botmaster. We believe that this is the first viable technique for performing live botmaster traceback on the Internet.

We validated our watermarking traceback algorithm both analytically and experimentally. In trials on public Internet IRC servers, we were able to achieve virtually a 100 percent TPR with an FPR of less than 10^{-5}. Our method can successfully trace a watermarked IRC flow from an IRC botnet member to the botmaster's true location, even if the watermarked flow 1) is encrypted with SSL/TLS; 2) passes through several stepping stones; and 3) travels tens of thousands of miles around the world.

Acknowledgments

The authors would like to thank the anonymous reviewers for their insightful comments that helped to improve the presentation of this paper. This work was partially supported by NSF Grants CNS-0524286, CCF-0728771 and CNS-0716376.

References

1. Bächer, P., Holz, T., Kötter, M., Wicherski, G.: Know Your Enemy: Tracking Botnets, March 13 (2005), http://www.honeynet.org/papers/bots/
2. Barford, P., Yegneswaran, V.: An Inside Look at Botnets. In: Proc. Special Workshop on Malware Detection, Advances in Info. Security, Springer, Heidelberg (2006)

3. Binkley, J., Singh, S.: An Algorithm for Anomaly-based Botnet Detection. In: Proc. 2nd Workshop on Steps to Reducing Unwanted Traffic on the Internet (SRUTI), San Jose, CA, July 7, 2006, pp. 43–48 (2006)
4. Blum, A., Song, D., Venkataraman, S.: Detection of Interactive Stepping Stones: Algorithms and Confidence Bounds. In: Jonsson, E., Valdes, A., Almgren, M. (eds.) RAID 2004. LNCS, vol. 3224, pp. 258–277. Springer, Heidelberg (2004)
5. Chi, Z., Zhao, Z.: Detecting and Blocking Malicious Traffic Caused by IRC Protocol Based Botnets. In: Proc. Network and Parallel Computing (NPC 2007). Dalian, China, pp. 485–489 (September 2007)
6. Cooke, E., Jahanian, F., McPherson, D.: The Zombie Roundup: Understanding, Detecting, and Disturbing Botnets. In: Proc. Steps to Reducing Unwanted Traffic on the Internet (SRUTI), Cambridge, MA, July 7, 2005, pp. 39–44 (2005)
7. Dagon, D., Gu, G., Zou, C., Grizzard, J., Dwivedi, S., Lee, W., Lipton, R.: A Taxonomy of Botnets (unpublished paper, 2005)
8. Donoho, D.L., Flesia, A.G., Shankar, U., Paxson, V., Coit, J., Staniford, S.: Multiscale Stepping Stone Detection: Detecting Pairs of Jittered Interactive Streams by Exploiting Maximum Tolerable Delay. In: Wespi, A., Vigna, G., Deri, L. (eds.) RAID 2002. LNCS, vol. 2516, pp. 17–35. Springer, Heidelberg (2002)
9. Evers, J.: 'Bot herders' may have controlled 1.5 million PCs. `http://news.com.com/2102-7350_3-5906896.html?tag=st.util.print`
10. Freiling, F., Holz, T., Wicherski, G.: Botnet Tracking: Exploring a Root-Cause Methodology to Prevent DoS Attacks. In: Proc. 10th European Symposium on Research in Computer Security (ESORICS), Milan, Italy (September 2005)
11. Goebel, J., Holz, T.: Rishi: Identify Bot Contaminated Hosts by IRC Nickname Evaluation. In: Proc. First Workshop on Hot Topics in Understanding Botnets (HotBots), Cambridge, MA, April 10 (2007)
12. Goodrich, M.T.: Efficient Packet Marking for Large-scale IP Traceback. In: Proc. 9th ACM Conference on Computer and Communications Security (CCS 2002), October 2002, pp. 117–126. ACM, New York (2002)
13. Grizzard, J., Sharma, V., Nunnery, C., Kang, B., Dagon, D.: Peer-to-Peer Botnets: Overview and Case Study. In: Proc. First Workshop on Hot Topics in Understanding Botnets (HotBots), Cambridge, MA (April 2007)
14. Gu, G., Zhang, J., Lee, W.: BotSniffer: Detecting Botnet Command and Control Channels in Network Traffic. In: Proc. 15th Network and Distributed System Security Symposium (NDSS), San Diego, CA (February 2008)
15. 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, Boston, MA (August 2007)
16. Holz, T.: A Short Visit to the Bot Zoo. Sec. and Privacy 3(3), 76–79 (2005)
17. Ianelli, N., Hackworth, A.: Botnets as a Vehicle for Online Crime. In: Proc. 18th Annual Forum of Incident Response and Security Teams (FIRST), Baltimore, MD, June 25-30 (2006)
18. Karasaridis, A., Rexroad, B., Hoein, D.: Wide-Scale Botnet Detection and Characterization. In: Proc. First Workshop on Hot Topics in Understanding Botnets (HotBots), Cambridge, MA, April 10 (2007)
19. Li, J., Sung, M., Xu, J., Li, L.: Large Scale IP Traceback in High-Speed Internet: Practical Techniques and Theoretical Foundation. In: Proc. 2004 IEEE Symposium on Security and Privacy. IEEE, Los Alamitos (2004)
20. Naraine, R.: Is the Botnet Battle Already Lost? `http://www.eweek.com/article2/0,1895,2029720,00.asp`

21. Rajab, M., Zarfoss, J., Monrose, F., Terzis, A.: A multifaceted approach to understanding the botnet phenomenon. In: Proc. 6th ACM SIGCOMM on Internet Measurement, October 25-27, 2006. Rio de Janeiro, Brazil (2006)
22. Roberts, P.F.: California Man Charged with Botnet Offenses, http://www.eweek.com/article2/0,1759,1881621,00.asp
23. Roberts, P.F.: Botnet Operator Pleads Guilty, http://www.eweek.com/article2/0,1759,1914833,00.asp
24. Roberts, P.F.: DOJ Indicts Hacker for Hospital Botnet Attack, http://www.eweek.com/article2/0,1759,1925456,00.asp
25. Savage, S., Wetherall, D., Karlin, A., Anderson, T.: Practical Network Support for IP Traceback. In: Proc. ACM SIGCOMM 2000, September 2000, pp. 295–306 (2000)
26. Snoeren, A., Patridge, C., Sanchez, L.A., Jones, C.E., Tchakountio, F., Kent, S.T., Strayer, W.T.: Hash-based IP Traceback. In: Proc. ACM SIGCOMM 2001, September 2001, pp. 3–14. ACM Press, New York (2001)
27. Symantec. Symantec Internet Security Threat Report – Trends for January 06 - June 06. Volume X (September 2006)
28. Micro, T.: Taxonomy of Botnet Threats. Trend Micro Enterprise Security Library (November 2006)
29. Wang, X., Chen, S., Jajodia, S.: Tracking Anonymous, Peer-to-Peer VoIP Calls on the Internet. In: Proc. 12th ACM Conference on Computer and Communications Security (CCS 2005) (October 2007)
30. Wang, X., Chen, S., Jajodia, S.: Network Flow Watermarking Attack on Low-Latency Anonymous Communication Systems. In: Proc. 2007 IEEE Symposium on Security and Privacy (S&P 2007) (May 2007)
31. Wang, X., Reeves, D.: Robust Correlation of Encrypted Attack Traffic Through Stepping Stones by Manipulation of Interpacket Delays. In: Proc. 10th ACM Conference on Computer and Communications Security (CCS 2003), October 2003, pp. 20–29. ACM, New York (2003)
32. Wang, X., Reeves, D., Wu, S.: Inter-packet Delay Based Correlation for Tracing Encrypted Connections Through Stepping Stones. In: Gollmann, D., Karjoth, G., Waidner, M. (eds.) ESORICS 2002. LNCS, vol. 2502, pp. 244–263. Springer, Heidelberg (2002)
33. Yoda, K., Etoh, H.: Finding a Connection Chain for Tracing Intruders. In: Cuppens, F., Deswarte, Y., Gollmann, D., Waidner, M. (eds.) ESORICS 2000. LNCS, vol. 1895, pp. 191–205. Springer, Heidelberg (2000)
34. Zhang, Y., Paxson, V.: Detecting Stepping Stones. In: Proc. 9th USENIX Security Symposium, pp. 171–184. USENIX (2000)

A Layered Architecture for Detecting Malicious Behaviors

Lorenzo Martignoni[1], Elizabeth Stinson[2], Matt Fredrikson[3], Somesh Jha[3], and John C. Mitchell[2]

[1] Università degli Studi di Milano
[2] Stanford University
[3] University of Wisconsin

Abstract. We address the *semantic gap* problem in behavioral monitoring by using hierarchical behavior graphs to infer high-level behaviors from myriad low-level events. Our experimental system traces the execution of a process, performing data-flow analysis to identify meaningful actions such as "proxying", "keystroke logging", "data leaking", and "downloading and executing a program" from complex combinations of rudimentary system calls. To preemptively address evasive malware behavior, our specifications are carefully crafted to detect alternative sequences of events that achieve the same high-level goal. We tested eleven benign programs, variants from seven malicious bot families, four trojans, and three mass-mailing worms and found that we were able to thoroughly identify high-level behaviors across this diverse code base. Moreover, we effectively distinguished malicious execution of high-level behaviors from benign by identifying remotely-initiated actions.

Keywords: Dynamic, Semantic Gap, Malware, Behavior, Data-Flow.

1 Introduction

In the first half of 2007, Symantec observed more than five million active, distinct bot-infected computers [1]. Botnets are used to perform nefarious tasks, such as: keystroke logging, spyware installation, denial-of-service (DoS) attacks, hosting phishing web sites or command-and-control servers, spamming, click fraud, and license key theft [2,3,4,5,6,7]. Malicious bots are generally installed as applications on an infected (Windows) host and have approximately the same range of control over the compromised host as its rightful owner. A botmaster can flexibly leverage this platform in real-time by issuing commands to his botnet. Several characteristics typical of botnets increase the difficulty of robust network-based detection; in particular, bots may: exhibit high IP diversity, have high-speed, always-on connections, and communicate over encrypted channels. Since a botmaster controls both the bots and the command-and-control infrastructure, these can be arbitrarily designed to evade network-based detection measures.

It is widely recognized that malware defenders operate at a fundamental disadvantage: malware producers can generate malware variants by simple measures such as packing transformations (encryption and/or compression) and may evade existing AV signatures by systematic means [8]. For the signature purveyors, moreover, analyzing a novel malware instance and creating a detection signature requires substantially greater effort than that required by evasion. The source of this asymmetry is the signature scanners' emphasis on malware's infinitely mutable syntax, rather than on the actions taken by malware. As a result, even the most effective signature-scanners fail to detect more than 30% of malware seen in the wild [9,10]. Therefore, it is essential to develop effective methods that identify the behaviors that make malware useful to their installers.

1.1 Our Approach

We propose, develop, and evaluate a behavior-based approach that targets the high-level actions that financially motivate malware distribution. For bots, these actions include "proxying", "keystroke logging", "data leaking", and "program download and execute." We build representations of these high-level actions hierarchically, taking care to identify only the essential components of each action. The lowest level event in our behavior specifications are system call invocations. Since any specific operating system kernel exports a finite set of operations, we can expect to be able to enumerate all possible ways to interface with that kernel in order to achieve a certain effect (e.g., send data over the network). Since there are a finite number of ways to achieve each high-level action, we can expect to create representations that encode all such ways. Consequently, we can hope to correct the asymmetry present in syntax-based approaches to malware detection.

In this paper we propose and evaluate a behavior-based malware detector that takes as input the behavior specifications introduced above and an event stream provided by our system-wide emulator (Qemu), which monitors process execution. A system-wide emulator provides a rich source of information but infers no higher-level effects or semantics from the observed events. This disconnect between a voluminous stream of low-level events and any understanding of the aggregate effect of those events [13] is referred to as the *semantic gap*. We address the semantic-gap by decomposing the problem of specifying high-level behaviors into layers, making our specifications composable, configurable, less error-prone, and easy to update. Our system compares a monitored process's event stream to behavior specifications and generates an event when there is a match. This generated event may then be used in the specification of a higher-layer behavior.

Fig. 1 provides a subset of the hierarchy of events used to specify our sample target high-level behavior: downloading and executing a program, which is used in malware distribution. Events are represented via rectangles, with directed edges between them indicating dependencies; e.g., the `tcp_client` event depends upon the `sync_tcp_client` and `async_tcp_client` events. At the lowest layer of the hierarchy, $L0$, we identify successful system call invocations. Each $L1$ event aggregates $L0$ events that have a common side effect, as is the case with the $L1$ `net_recv` event which is generated whenever any of the $L0$ events `recv`, `recvfrom`, or `read` occur. Consequently, we can represent "all ways to receive data over the network" using a single event. Events at layers $L2$ and higher identify correlated sequences of lower-layer events that have some

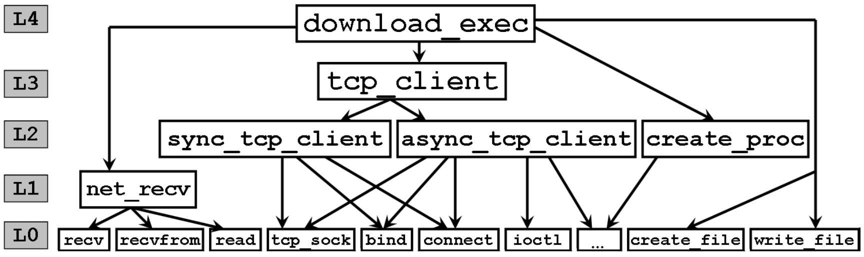


Fig. 1. A subset of the hierarchy of events used to specify `download_exec`

aggregate, composite effect; e.g., `sync_tcp_client` identifies when a synchronous TCP socket has been successfully created, bound, and connected upon.

Correlating low-level events generally entails specifying constraints on those events' arguments. In some cases, we need to specify that data used in one event is dependent upon data used in another event. Consequently, Qemu performs instruction-level data-flow analysis (tainting) and exports two related operations: `set_tainted` designates a memory region tainted with a particular label; and `tainted` determines whether a memory region contains data received from a particular source (as identified by its taint label). An important class of tainted data is that which is derived from local user input; this *clean data* is used to differentiate locally-initiated from remotely-initiated actions. Both `tainted` and `set_tainted` can be used in our behavior specifications; consequently, we can designate novel taint sources without changing our system implementation.

Commonly, malware variants are generated by: (I) applying packing transformations (compression and/or encryption) to a binary, (II) applying instruction-level obfuscation such as nop insertion as in [15], (III) applying source-level obfuscations as in [8], (IV) using a bot-development kit, which provides a point-and-click interface for specifying bot configuration details and builds the requested bot, or (V) directly modifying the source of an existing bot to add novel functionality and/or commands. Our behavioral graphs are insensitive to the type of changes entailed in (I) – (III) since the semantics of a malware's behavior are unchanged. The changes in (IV) also do not affect the bot's implementation of a particular command, only whether that command is available or not. Moreover, since we identify the fundamental system-call signatures for high-level behaviors, even changing the implementation as in (V) without changing the overall semantic effect would not suffice to evade detection.

The contributions of this paper include:

– A behavior-specification language (described in Section 2) that can be used to describe novel, semantically meaningful behaviors.
– A detector (described in Section 3) that identifies when a process performs a specified high-level action, regardless of the process's source-code implementation of the action.
– Our evaluation (described in Section 4) demonstrates that our detector can distinguish malicious execution of high-level behaviors from benign.

2 Representing High-Level Behaviors

In this section, we define our behavior graphs, each of which describes a correlated sequence of events that has some particular semantic effect (such as `connect` or `tcp_client`). The graph for a behavior B identifies only the fundamental component events required to achieve B and constrains these events as minimally as possible. Matching a behavior graph generates an event that can be used as a component within another graph; e.g., matching the `tcp_client` graph generates the `tcp_client` event, which can be used in specifying other behaviors, such as `tcp_proxy`. In this way, we compose graphs hierarchically, which enables us to recognize complex process behaviors.

2.1 Behavior Graphs

A *behavior graph* is a directed graph of a form that is adapted from and extends AND/OR graphs [26]. A behavior graph can be thought of as a template; multiple different sequences of events can be bound to this template subject to the edge constraints; binding and matching are described more precisely in sect. 2.1. Fig. 2 contains the behavior graph for our running example, `download_exec`.

Each behavior graph has a *start point*, drawn as a single point at the top of the graph, internal nodes, and an *output event*, which is represented via a shaded rectangle. Each internal node in the graph has a name, such as `create_file`, and formal parameters, such as `fh0`, `fname`, `fname_len`, as in fig. 2. Together, a node's name and formal parameters characterize a set of events, namely those events whose name is the same as the node's name. Whereas internal nodes represent input events needed in order to continue graph traversal, the special output event represents an action taken by our system; hence no additional input is required to traverse an edge from a penultimate node to the output event. For example, any sequence of events that matches the graph in fig. 2 up to the `create_proc` node will also reach the `download_exec` node and generate a `download_exec` event. When we match a graph and generate an output event e, the parameters for e are obtained from e's constituent events; e.g., the socket descriptor, `rem_ip`, and `rem_port` arguments for the `download_exec` output event in fig. 2 are obtained from its constituent `tcp_client` event.

AND-edge sets and OR-edge sets. A behavior graph may have AND-edges and OR-edges. OR-edges are drawn simply as directed edges, while AND-edges are drawn using a horizontal line to form an AND-edge set. In fig. 2, a sequence of events can reach the `net_recv` node by either of the two OR-edges leading into this node. In contrast, the AND-edges into `write_file` indicate that both `net_recv` *and* `create_file` are required to match this portion of the graph. If a node's in-edge set contains AND-edges and OR-edges, this expresses an OR of ANDs. We use AND-edge sets to identify events which can occur in any relative order but must all precede some other event.

Annihilator and Replicator Nodes. We correlate events by specifying predicates on their parameters; thus, it's important to know when a parameter has

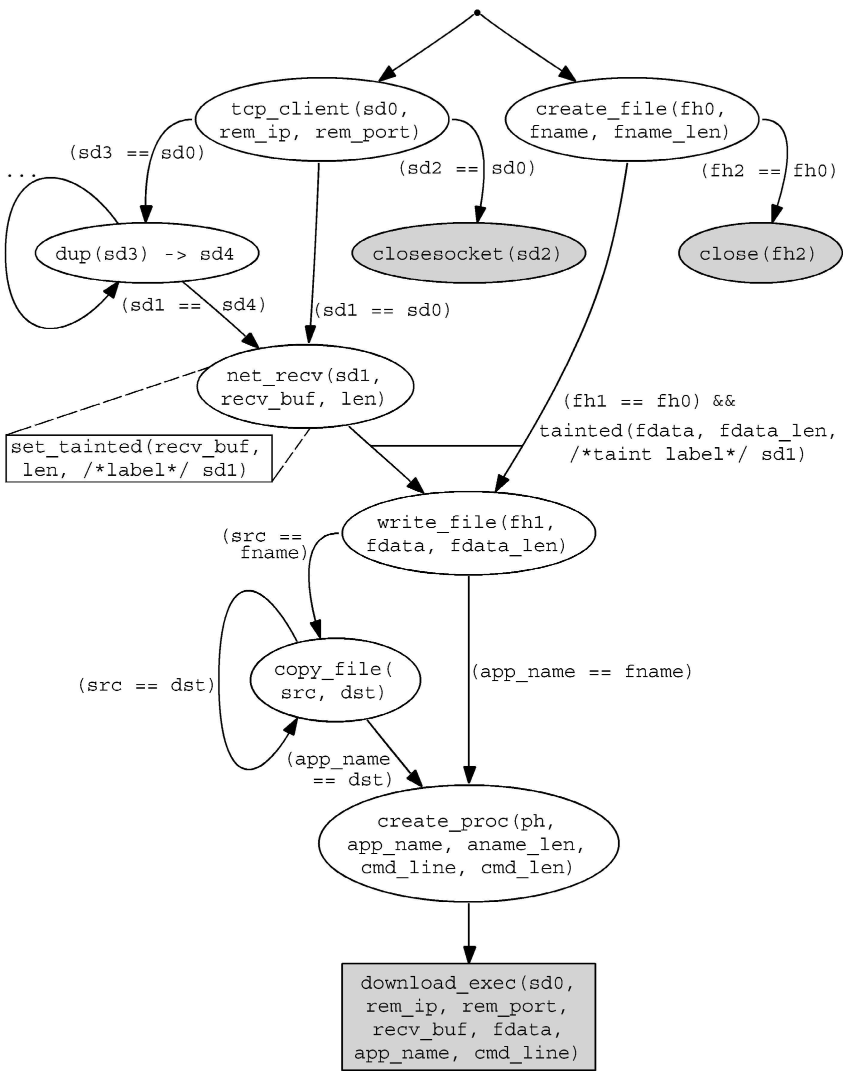

tcp_client(sd0, rem_ip, rem_port)
create_file(fh0, fname, fname_len)
(sd3 == sd0)
...
(sd2 == sd0)
(fh2 == fh0)
dup(sd3) -> sd4
closesocket(sd2)
close(fh2)
(sd1 == sd4)
(sd1 == sd0)
net_recv(sd1, recv_buf, len)
set_tainted(recv_buf, len, /*label*/ sd1)
(fh1 == fh0) &&
tainted(fdata, fdata_len, /*taint label*/ sd1)
write_file(fh1, fdata, fdata_len)
(src == fname)
copy_file(src, dst)
(src == dst)
(app_name == fname)
(app_name == dst)
create_proc(ph, app_name, aname_len, cmd_line, cmd_len)
download_exec(sd0, rem_ip, rem_port, recv_buf, fdata, app_name, cmd_line)

Fig. 2. AND/OR graph for downloading a file and executing it

been destroyed or duplicated. *Annihilator nodes* are used to represent that certain events destroy objects; e.g., calling `closesocket` on a socket descriptor `sd` releases `sd`, rendering it unable to be used in subsequent events. Annihilator nodes are represented via shaded ellipses, as with the `close(fh)` node in fig. 2. The edge from `create_file(fh0, ...)` to `close(fh2)` imposes the condition that `close` cannot be called on the newly-created file handle prior to `write_file(...)` being called on that same handle. Certain events, which we refer to as *replicators*, duplicate objects, such as socket descriptors or files. For example, calling `dup` on a socket descriptor or file handle creates a copy of the passed object; any operation that could be called on the original object can equivalently be called on the duplicate. We represent this via replicator nodes as with `dup(...)` and `copy_file(...)` in fig. 2. Since a replicator operation can be called repeatedly on its own output, replicator nodes contain a self-loop. For succinctness, some annihilators and replicators are excluded from the figures.

Edge Predicates. A directed edge can be labeled with predicates that must be satisfied to traverse the edge. Our system provides three predicate types: argument-value comparison, regular expression matching, and the `tainted` predicate. In *argument-value comparison*, we can apply any of the standard relational operators ($=$, $\neq$, $>$, $<$) to compare an argument value to a constant or to another argument. Fig. 2 contains several argument-value predicates, such as `(sd1 == sd0)` between the `tcp_client` and `net_recv` events. We can also specify that a string or buffer argument value must match a constant *regular expression* as used in the `send_email` behavior graph to identify transmission of SMTP protocol messages (e.g., `MAIL FROM`). The `tainted` predicate identifies data-flow relationships that must hold; we can require that an argument be derived from a general taint source (e.g., the network) or a specific taint source (e.g., a particular network connection). Fig. 2 includes a data-flow dependency; namely, the data written to the newly-created file (`fdata`) must be derived from data received over the specified network connection as indicated by its taint label (`sd1`).

`On-reach` **Actions.** Our monitoring system can perform an action in response to reaching a given node. An `on_reach` action is represented in the graph via a rectangle – connected to its corresponding node via dashed lines – containing the action to be performed. Fig. 2 shows that, upon reaching the `net_recv` node, the received buffer will be marked tainted with the taint label `sd1`.

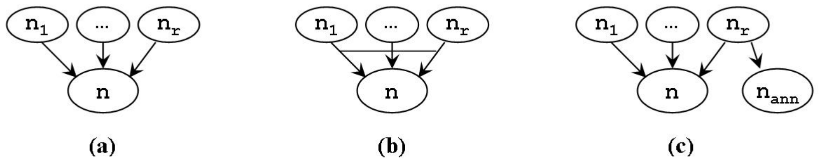


Fig. 3. Graph G with (a) OR-ed edges, (b) AND-ed edges, (c) an annihilator

Summary. A behavior graph defines a set of event sequences that match the graph, and may specify one or more on-reach events that will be generated when events match the graph in certain ways. These properties may be captured precisely in a rigorous definitions that allow us to prove properties of various algorithms. For example, a sequence $E = e_1, e_2, \ldots, e_k$ of events matches a behavior graph G if there is a function f from a subset of the nodes of G to events in E and a substitution S on variables that appear in formal parameters of the graph that satisfy the following conditions:

1. If there is an OR-edge set into a node n with $f(n) \in E$, as illustrated in fig. 3(a), then $\exists i.\, f(n_i) \in E$.

2. If there is an AND-edge set into a matched node n with $f(n) \in E$, as illustrated in fig. 3(b), then $\forall i.\, f(n_i) \in E$.
3. If there are matched nodes $n_r \rightarrow n$ with an annihilator node as illustrated in fig. 3(c), then $\nexists$ event $e \in E$ with $f(n_r) < e < f(n)$ and e matches $ev(n_{ann})$ by any $S' \supseteq S$.
4. If predicate P appears on an edge between nodes n and n' with $f(n) \in E$ and $f(n') \in E$, then the substitution instance $S(P)$ of P is true.

2.2 Behavior-Specification Language

A major contribution of our work is our behavior-specification language and monitoring system. Together, these can be used to specify then identify novel semantically-meaningful behaviors. The substrate consists of the graphs at each layer. Each of the behaviors specified by these graphs is a primitive that can be used in defining additional behaviors. Table 1 contains some primitives from our resulting behavior-specification language. We can describe "log keystrokes then send them in an email" using two of these primitives (`keylogging` and `send_email`) and correlating their arguments in a particular way, which illustrates the powerful, high-level expressiveness of our language.

2.3 Graph Construction

We developed our graphs manually and iteratively through domain knowledge and analysis of tens of gigabytes of execution traces, obtained from multiple runs of (i) around fifteen standard applications (including Googletalk, Filezilla, Firefox, putty, mIRC, Internet Explorer, Outlook, Thunderbird, SecureFX, Windows Media Player, SecureCRT, Unreal IRCd, Apple Software Update, Quicktime, etc.), (ii) over one hundred specially-crafted programs, and (iii) several malicious programs. We present our evaluation of these graphs' coverage in sect. 4.2.

Constructing $L0$ Graphs. Recall that $L0$ graphs represent successful system call invocations. The challenges here are as follows, (i) Windows implements the sockets API through a single system call, `NtDeviceIoControlFile`, (ii) we do not have source access to the target OS, and (iii) we need to be able to differentiate invocations of `listen` from invocations of `accept` and so on. We rely on analysis of process execution traces in order to identify commonalities

Table 1. Some primitives in our resulting behavior-specification language

Event	*Arguments*
tcp_client	`sd`, `loc_ip`, `loc_port`, `rem_ip`, `rem_port`
tcp_server	`sd`, `loc_ip`, `loc_port`, `cli_ip`, `cli_port`
net_send	`sd`, `buf`, `buf_len`
net_recv	`sd`, `buf`, `buf_len`
send_email	`sd`, `targ_ip`, `from_addr`, `to_addr`, `data`
keylogging	`data`, `data_len`

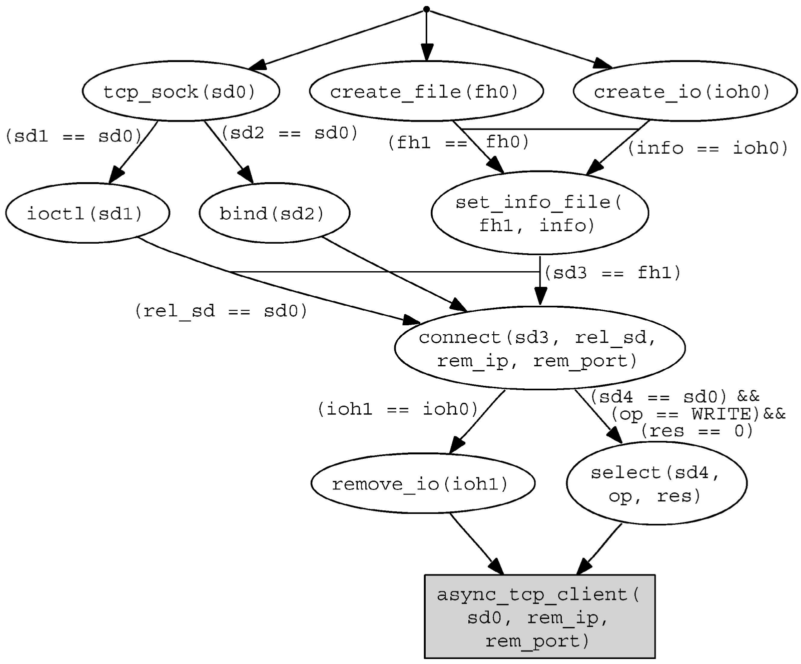


Fig. 4. $L2$ AND/OR graph for an asynchronous TCP client

(in arguments) across all invocations of a sockets function s_1 but which are not present in any invocations of all other sockets functions, s_2, s_3, ..., s_k. These commonalities are the basis of our $L0$ behavior graphs. The coverage of any graph then relies upon the diversity of process traces. Our process traces delineate entry to and return from each sockets function and identify all system calls invoked therein, including each system call's arguments and return value.

For some functions, such as `socket`, we crafted a suite of programs that invoked the function using all possible combinations of valid arguments. The execution of other sockets functions, however, is stateful in that it depends directly upon previous actions performed on the same socket descriptor; e.g. `recv`. Hence, it is not enough to provide different argument combinations to `recv`, we must also precede the invocation of `recv` with different combinations of particular sockets functions, such as `socket`, `bind`, `listen`, `connect`, and so on.

Pending System Call Invocations. A system call *sc* may not immediately return success or failure but rather return `STATUS_PENDING`; `NtWaitForSingleObject` is subsequently invoked on *sc*'s associated event object. We encode this path in our $L0$ graphs so as to identify *eventually successful* system call invocations.

Constructing $L1$ Graphs. Recall that $L1$ graphs aggregate $L0$ events that have a similar side effect. Since the system call interface is finite, we can enumerate the "relevant" effects of each system call and construct an $L1$ graph

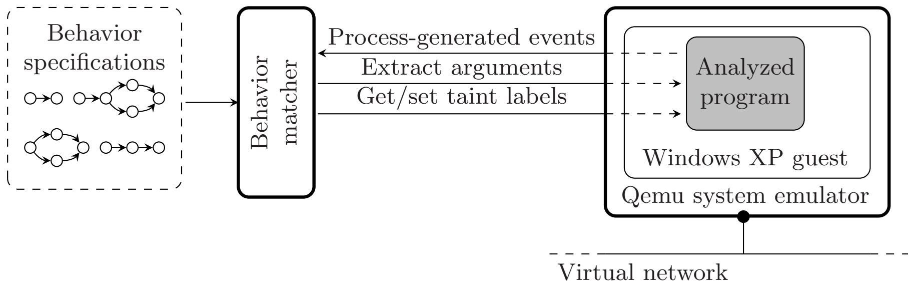


Fig. 5. Architecture of the system

for each such effect, where by "relevant" we mean "of interest". In our case, there were two effects that required $L1$ graphs: `net_send` and `net_recv`. These were immediately identifiable through domain knowledge. Note that aggregation graphs can exist at higher layers as well; e.g., we use an $L3$ graph to aggregate `async_tcp_client` and `sync_tcp_client` so that we may identify generally `any tcp_client`.

Constructing *L*2 Graphs. The graphs at $L2$ identify correlated sequences of lower-layer events which have some aggregate, composite effect, e.g. `create_write_file`. For each target $L2$ behavior, we identify the events essential to that behavior and any dependencies between those constituent events. This identification comes through (I) domain knowledge, such as encoding that in order to `connect` or `listen` on a socket, that socket must first have been (explicitly or implicitly) bound, and (II) analysis of process traces, as used to construct the graphs for asynchronous network interaction. Windows exports a rich API for performing asynchronous network interaction, including the standard polling model using `select` on a socket as well as event-based approaches, such as via `WSAEventSelect` and `WSAAsyncSelect`. We are able to represent all of these through a single asynchronous TCP client graph as in fig. 4. This graph was built by examining process traces of existing applications which use the Windows asynchronous API as well as augmenting this analysis with traces of specially-crafted programs designed to capture more execution diversity.

3 System Implementation

Figure 5 depicts the architecture of our system, which has two main components: a system-wide emulator (Qemu) and a behavior matcher. Qemu emulates and traces the execution of analyzed programs in an isolated virtual environment. We use a hybrid emulated/virtualized approach, where the execution of the process under analysis is emulated while the execution of all other processes in the system is virtualized using KQemu [38]. The behavior matcher obtains information about process events from the emulator and attempts to match this input event stream to the behavior graphs. The behavior matcher operates independently of the particular monitoring technique and, as such, could be used in concert with, e.g., a process emulator. We use system-wide rather than process emulation for reasons relating to ease of experiment execution and cleanup. In particular, Qemu offers built-in support for rollback of system state and enables easy isolation of the monitored process from the external world.

3.1 System Emulator

Our system-wide emulator extends Qemu [39], an open-source emulator based on dynamic binary translation, by adding guest-OS-aware virtual machine introspection and taint analysis capabilities [25]. Guest-OS awareness is essential as we must be able to determine: which system call was invoked, which process invoked it, and the format of the system call's argument buffers. Our system currently emulates the IA-32 architecture and supports Microsoft Windows XP.

Process-generated Events. We instrument the code executed in the emulator by hooking the `sysenter` and `sysexit` instructions, which identify, respectively, invocation of and return from system calls. The instrumentation causes the emulator to provide this event stream to the behavior matcher in real-time.

Taint Analysis. The code executed in the emulator is also instrumented to perform taint analysis. In order to propagate taint labels through data dependencies, we extend the semantics of instructions that assign a value to a register or memory location, excluding floating point operations. We set the label of an assignment instruction's destination operand to be the union of the source operands' labels. Instructions instrumented in this manner are referred to as *taint propagation instructions*. To reduce overhead, we perform taint analysis on user-space code only. Our system also includes support for custom taint propagation rules over operations at a higher level than machine code instructions. In particular, we use this support to propagate taint across system calls that participate in hostname resolution; we assign the labels from the input hostname buffer to the location storing the resolved IP address.

Local User Input Tracking. Our local user input tracking module is designed for Win32 GUI applications, which receive messages indicating keyboard or mouse input events. The receiving application invokes its handler for the input event via a call to `DispatchMessage`. Mouse input messages do not provide the data value associated with the event; hence, identifying this data is a challenge. We address this by entering *clean mode* whenever the monitored process is handling receipt of a mouse click or keystroke; we define that period as starting with select invocations of `DispatchMessage` and ending with the corresponding returns. During clean mode, all taint propagation instructions unconditionally set the labels of their destination operands to be the special `clean` label. We present evaluation details related to user input tracking in sect. 4.6.

3.2 Behavior Matcher

At startup, the behavior matcher loads the provided set of behavior graphs. The matcher maintains some state for each graph, including the graph's current set of active nodes. A node n_{act} in graph G is *active* when we have received some event sequence $<ev_1, ev_2, ..., ev_k>$ which causes us to transition from the `start` state of G to n_{act}. There may be multiple event sequences corresponding to any particular active node; these event sequences (including each event's actual parameters) are also part of a graph's state. For brevity, certain details of the matching algorithm are omitted.

The behavior matcher is notified in real-time by the emulator every time the monitored process invokes or returns from a system call. Given a new event e with name $name_e$, for each behavior graph, the matcher: (I) checks whether there is a transition from an active node to a node n_{new} whose name is the same as $name_e$; if not, discard e, (II) extracts e's actual parameters and binds them to n_{new}'s formal parameters; (III) evaluates the predicates on (n_{act}, n_{new}); if they do not hold, discard e; (IV) if there is an `on-reach` action associated with n_{new}, then execute it; (V) if there is an edge from n_{new} to this graph's output event, the matcher generates the appropriate synthetic event.

4 Evaluation

This section provides the results of testing our dynamic specification-driven system monitor on seven malicious and eleven benign applications. After describing the experimental setup, we provide results demonstrating our ability to fill the semantic gap. Additionally, in testing the bots and benign applications against seven behavior graphs (referred to as *malspecs*) corresponding to bots' most threatening behaviors, there were no false negatives and seven false positives.

4.1 Experimental Setup

We performed our evaluation of the system in the environment depicted in Fig. 5. The evaluation framework consists of a victim Qemu virtual machine VM_{vict}, which is connected to a second virtual machine VM_{gway}, which is acting as a network gateway. On VM_{vict}, the system-wide emulator monitors the target malicious or benign process. The purpose of VM_{gway} is three-fold: it isolates the emulator from the external network to prevent further infection; it provides a realistic network environment for the execution of network-aware malware; and it hosts the command-and-control (C&C) server used to direct bots' activities.

4.2 Graph Validation

To determine whether our behavior graphs adequately cover semantically -equivalent but programmatically-different execution paths, we ran a diverse suite of applications within our monitoring framework and performed matching against a set of behavior graphs corresponding to generally innocuous actions. The column headings in Table 3 identify the tested behavior graphs. We drove each application's execution via performing the actions described in Table 2.

Table 2. Actions over which benign programs were exercised

Application	**Interaction**
ftp.exe, FTP Wanderer	Connect to server, authenticate, get a file, get multiple files
Internet Explorer	Access google.com, perform FTP access, download and execute a program.
Outlook Express	Download and read email containing an external image, reply to email, download and execute an attachment.
PuTTy	Connect and authenticate with server, send commands, use as SSH tunnel.
WinSCP, pSCP	Copy a file from server to client (and vice versa) using wildcards, download and execute a program.
SDK Installer	Download and install debugging tools from Microsoft server.
mIRC	Chat on a typical channel, DCC send, DCC get.
Google Talk	Chat, start a voice call, attempt a file transfer.
EasyProxy	Start proxy, route HTTP traffic.

Table 3. Graph validation results. Blank entries indicate that the software did not perform the tested behavior.

	TCP Client	**TCP Server**	**Net Send**	**Net Recv**	**Create Proc**	**Dwnld File**	**Dwnld & Exec**	**Send Email**	**TCP Proxy**
ftp.exe	✓	✓	✓	✓		✓			
Internet Explorer	✓		✓	✓	✓	✓	✓		
Outlook Express	✓		✓	✓	✓	✓	✓	✓	
PuTTy	✓	✓	✓	✓		✓			
pSCP	✓		✓	✓		✓			
WinSCP	✓		✓	✓	✓	✓	✓		
FTP Wanderer	✓	✓	✓	✓	✓	✓	✓		
SDK Installer	✓		✓	✓		✓			
mIRC	✓	✓	✓	✓		✓			
Google Talk	✓		✓	✓		✓			
Easy Proxy	✓	✓	✓	✓					✓

Moreover, during process execution, we performed manual analysis of network traffic and OS state in order to obtain "ground truth" about a process's actions. In this way, we were able to determine which behavior specifications any particular process should match at any point in time. Table 3 shows the output of our behavior matcher on each application and for each behavior graph. In all instances, the behavior matcher's output comported with ground truth, demonstrating that our graphs identify the fundamental components of the tested behaviors. Recall that graphs at $L2$ and higher compose lower-layer graphs. Hence, our evaluation was performed over more than forty distinct graphs.

Table 4. Malspecs used for evaluation. Recall that "RI" stands for remotely-initiated. Use of "tainted" in the below refers to data received over the network.

	Name	Description
M1	RI Create and Execute File	A file with a tainted name is created, tainted data is written to the file, and a process is created from the file.
M2	RI Net Download	A connection to a tainted address or port is created, a file with a tainted name is created, and tainted data is written to the file.
M3	RI Send Email	A sequence of messages is matched using regular expressions, and found to correspond to an SMTP message sent to a tainted email address.
M4	RI Sendto	A UDP packet is sent to a tainted port or address.
M5	RI TCP Proxy	An application binds to a tainted port number, connects to a tainted address, and relays information from the tainted port to the tainted address.
M6	Keylogging	An application captures keystrokes destined for another process.
M7	Data Leak	An application sends data from either the filesystem or the registry over a network connection.

Table 5. Results on malicious bots. Blank entries denote behaviors not matched because the bot did not implement them; † entries denote behaviors which, when exercised, caused the bot to crash.

	M1	M2	M3	M4	M5	M6	M7
rBot	✓	✓	†	✓	✓		✓
Agobot	✓	✓	✓	✓	✓		✓
DSNX	✓	✓			✓	✓	✓
SpyBot	✓				✓	✓	✓
gSys	✓	✓		✓	✓		✓
rxBot	✓	✓	✓	✓	✓		✓
SDBot	✓	✓		✓	✓		✓

4.3 Specifications of Malicious Behavior

The malicious behavior specifications used in our evaluation (malspecs) reflect the targeted class of malware: bots. We targeted bots because their diverse range of behaviors encompasses the full range of behaviors performed by some other types of malware. Our malspecs (described briefly in Table 4) correspond to the most alarming threats posed by bots [2,3,4,5,6,7], including: malware install (M1, M2), spamming (M3), DoS attacks (M4), proxying (M5), and identity theft (M6, M7). Since bots act at the behest of a remote entity (the botmaster), we describe their actions as *remotely-initiated* (RI), which occurs when the values used to perform an action depend on data received over the network [16].

Table 6. Results on benign applications. "$\overline{\text{UI}}$" refers to an experiment in which user input tracking was not used, and "UI" to one with it enabled.

	M1		**M2**		**M3**		**M4**		**M5**		**M6**		**M7**	
	$\overline{\text{UI}}$	UI	$\overline{\text{UI}}$	UI	$\overline{\text{UI}}$	UI	$\overline{\text{UI}}$	UI	$\overline{\text{UI}}$	UI	$\overline{\text{UI}}$	UI	$\overline{\text{UI}}$	UI
ftp.exe													✓	✓
FTP Wanderer			✓										✓	
Internet Explorer	✓		✓								✓	✓		
Outlook Express	✓		✓										✓	✓
PuTTy													✓	✓
pSCP													✓	✓
WinSCP	✓												✓	✓
SDK Installer			✓											
mIRC			✓										✓	
Google Talk			✓				✓						✓	✓
EasyProxy														

4.4 Malware Results

We evaluated our system against seven malicious bots: rbot, Agobot, DSNX, Spybot, gSys, rxbot, and SDBot. When run in VM_{vict}, the bot connected to its C&C server (hosted in VM_{gway}), received a series of commands, and executed each. Table 5 shows the malspecs matched by each bot. From this, two conclusions can be drawn: first, we can detect when a process performs a high-level, semantically meaningful action, such as **Remotely-Initiated Net Download** (M2); and secondly, a single malspec can be used to identify a malicious behavior in a variety of bots. In one case, a command fed to a bot caused the bot to crash; consequently, we don't have results of executing the `email` command on rBot, which we expected would match **RI Send Email** malspec (M3).

4.5 Benign Application Results

To determine whether our malspecs sufficiently encode the difference between malicious behavior and benign, we tested eleven benign applications against these malspecs. We chose benign applications and actions over which to drive each application by favoring those with the greatest perceived likelihood of triggering a match on at least one malspec. Due to the black-box nature of many Win32 applications, this selection process is imperfect. Since we favored network-intensive applications and since our malspecs define remotely-initiated actions as those which use network-supplied parameters, we expect some false positives.

Table 6 provides the results of evaluating each of our benign programs against the set of malspecs. We ran each program under two scenarios: first, with user-input tracking disabled, which corresponds to the $\overline{\text{UI}}$ column; and second, with user-input tracking enabled, which correspond to the UI column. What this means is that, e.g., GoogleTalk matched M2, M4, and M7 when we performed no user input tracking and only matched M7 when this tracking was enabled.

We note that, in general, we are better able to distinguish malicious from benign when we take local user input into consideration in the manner described in 3.1.

The malspec matched by most benign programs (regardless of whether user input was taken into consideration) is **Leak** (M7). **Leak** identifies when data read from a file or the registry is subsequently sent on the network. This manifests in malicious applications when sensitive user data or product keys are transmitted to the botmaster. The deficiency of this malspec is its coarse granularity; i.e., reading data from *any* file on the system and sending any portion of that causes a match. In actuality, we would prefer to encode that, when an application reads data that *does not belong* to that application, this is considered a breach. So, in a sense, a more finely-tuned **Leak** malspec would retrofit fine-grained access control for applications on Windows systems, enabling application X to read from files and registry keys belonging to X. As proof of concept, we tuned the **Leak** malspec to exclude cookie files and certain registry keys belonging Internet Explorer (IE), which explains why IE does not match M7.

4.6 Tracking Local User Input

Since our benign results make clear the importance of identifying and tracking data which is dependent upon local user input, it is important to understand how often the system is cleaning data in response to local user input (as described in 3.1). If it is the case that our system is in "clean mode" the vast majority of the time, one might question the validity of our distinction between malicious and benign. We identified the number of instructions executed by a benign process over its lifetime as well as the number of instructions executed by that process while it was in *clean* mode. The percent of instructions executed in clean mode for three representative applications was: mIRC, 1%; Outlook, 3%; and IE, 9%. Thus, user-input tracking is performed for a very small portion of a process's lifetime and, hence, our designation of data as clean is conservative.

4.7 Additional Malware

Though our sample malspecs target malicious bots, high-level specifications can be generated to identify other classes of malware. To demonstrate this, we evaluated four Trojans (Bancos, two variants of Banker, and Delf) and three mass-mailing worms (all variants of Bagle) using our previous malspecs plus a new malspec designed to detect self-propagation through email. With no modifications to the **Leak** malspec (M7), each Trojan matched it. To identify self-propagation through email, we modified the **Remotely-Initiated Send Email** malspec (M3). Rather than requiring that the data-flow be from the network to an SMTP message, we specified that the data-flow must be from the code of the executable itself to an SMTP message, which corresponds to a process sending its own code in an email. This demonstrates that specifying signatures for entirely new classes of malware can be straightforward and intuitive.

Table 7. Performance overhead of the system. The **Tainting** column identifies the factor slowdown of running Qemu with tainting over vanilla Qemu. Each MX column identifies the factor slowdown (over vanilla Qemu) of performing both tainting and behavior matching for the given malspec. Startup time is not included and is on the order of ten seconds.

	Tainting	**M1**	**M3**	**M6**
Internet Explorer	5.25	11.53	7.19	5.64
pSCP	7.32	8.08	19.62	7.42
Agobot	3.01	16.40	23.73	16.84
rBot	9.50	11.20	11.08	9.62

4.8 Performance Overhead

We evaluated the performance overhead of our system on a subset of the malicious and benign applications used in the evaluation, including Agobot, rBot, pSCP, and Internet Explorer. We ran each application under three different scenarios: (I) Qemu with no tainting; (II) Qemu with tainting; (III) Qemu with tainting and behavior matching for each of three different malspecs. For each application under each scenario, we measured the amount of wall clock time elapsed between a set of events captured in system logs. We selected events that did not depend on user input, so as to preserve as much determinism as possible.

The **Tainting** column in Table 7 identifies the factor slowdown of using Qemu with tainting over Qemu without tainting, which we refer to as vanilla Qemu. We rely on previous work to determine the overhead of vanilla Qemu relative to native execution, which is substantial: on the order of a 7X to 23X [12]. Each MX column identifies the factor slowdown of performing both tainting and behavior matching for the given malspec. To obtain the total slowdown over native execution, we add the MX value to the numbers in [12]; e.g., running behavior matching using the M3 malspec on rBot exacts an 18X to 34X performance penalty over native execution. Our system yields rich information and would ease the analysis performed in applications which may be less performance sensitive.

5 Limitations and Future Work

Limitations. There are several approaches to evasion that we can imagine attackers would adapt against a system such as ours. In particular, since we identify correlated sequences of system calls, efforts to disrupt our ability to correlate are an obvious choice. This disruption could take the form of splitting the work required to achieve some high-level action across multiple processes or across different instantiations of the same process. Another high-level approach at evasion relates to our assumption that the malicious process interacts with the kernel. Malware that expropriate kernel functionality would disrupt our ability to see and thus correlate their events. For example, an application could use raw sockets and write its own IP and transport-layer headers rather than calling the standard `sockets` functions such as `connect`, `accept`, and so on. Malicious software could also attempt to subvert our user-input tracking. Another approach to evasion relates to breaking our assumption about data-flow; in particular, malware could convert data-flow dependencies into control-flow dependencies thus defeating our mechanism for determining when an action is remotely-initiated. Finally, because we are interposing on a process, we are vulnerable to Time-Of-Check-Time-Of-Use (TOCTOU) bugs as in [24].

Future Work. We are very interested in exploring automated ways of generating the behavior graphs at various layers of the hierarchy. At $L0$, perhaps given source code access, we could ascertain precisely the set of low-level events (and the constraints on those events) that corresponds to each `sockets` operation. Moving up the hierarchy, such access would also presumably enable us to determine all possible sequences of events which achieve some semantic effect, such as `tcp_client`. An alternative approach may be to use symbolic execution to infer these behavior graphs. In this way, we would still achieve our semantic understanding of the aggregate effect of a process's actions but would have more confidence in our coverage than can be obtained through even rigorous testing.

6 Related Work

Behavior-Based Malware Detection. Host-based behavior-based research has been done to identify rootkits, spyware, and bots [22,23,20,19,16]. In [19], Cui *et al* identify *extrusions*: stealthy outgoing network connections made by malicious processes. In the commercial sector, Sana Security's ActiveMDT [21] correlates a process's exhibition of various mostly stateless behaviors to determine whether the process is likely to be malicious. The simple behaviors include: whether a process spawns or terminates other processes, the directory from which a process executes, whether the process attempts to hide, and so on.

Egele *et al* present a method and system for detecting spyware implemented as a Browser Helper Object (BHO) in [30]. The method identifies *malicious information access and processing* when sensitive information flows (such as the list of URLs visited) are written by a BHO to the network, file system, or shared memory. Moreover, they perform static analysis to identify instructions that are control-dependent on sensitive information. Since spyware-writers could prevent the static analysis in [30] from identifying the post-dominator node, they consider failure of their static analysis to be indicative of malicious intent. This control-flow tracking is only performed for BHO code so it's unclear whether such tracking, if applied to general-purpose programs, would blur the ability to distinguish between malicious and benign. Yin *et al* developed a related malware detector, *Panorama* [18], which performs full-system, instruction level tainting and can express more diverse leakage policies than [30]. We can express the behavior identified by these systems using our specification language. As with [18], we do not currently track implicit information flows.

The behavioral specifications developed by Christodorescu *et al* [11] are similar to ours. Our specifications differ in three important ways. First, we use

AND-edges which enables expressing concurrent behaviors. Second, we introduce synthetic event nodes, in order to identify complex behaviors hierarchically. Additionally, the specifications used in Christodorescu's work were generated automatically using data mining techniques, as opposed to the manual techniques we used. This has a few significant implications. Most importantly, their specifications identify sequences of actions which happen to occur in some malicious software; the aggregate effect of such sequences is unknown as is the value to the malware of performing those actions. That is, their specifications may identify *incidental*, rather than fundamental or mission-critical, behaviors as are targeted by our work. Additionally, no effort is made to cover semantically equivalent sequences. Consequently, there may be alternative sequences of system calls which have the same effect as a mined sequence but are not identified in their graphs.

Dynamic Code Analysis. Some systems use emulation to monitor the execution of suspicious executables [27,33,34]; however, rather than attempting to infer high-level behaviors, these systems merely report the numerous low-level events, such as system calls and API invocations, generated during execution. Other research has focused on addressing the shortcomings of dynamic analysis, including using symbolic execution to explore multiple execution paths [31,32].

Semantic Gap Problem. The semantic gap problem was explored by Garfinkel *et al*, as part of an attempt to embed an intrusion detector into a virtual machine monitor [25]. Related systems include honeypots [29,28], where introspective capabilities are used to examine the state of the filesystem in order to detect hidden files. Rather than encoding semantic information about the system, Jones [35] applied implicit techniques to infer relevant state. One notable result was the use of these techniques to detect processes hidden by rootkits [36].

7 Conclusion

Bots are an extremely widespread and serious problem, allowing remote bot masters to direct the activities of millions of compromised hosts. We develop new behavioral monitoring techniques that are effective for identifying meaningful high-level actions, based on hierarchical behavior graphs. Behavior graphs provide a high-level specification language that can be used to describe semantically meaningful behaviors such as "proxying", "keystroke logging", "data leaking", and "downloading and executing a program." To address evasive malware behavior, our specifications are carefully crafted to detect alternate sequences of events that achieve the same goal.

Our experimental emulation-based detector identifies when a process performs a specified high-level actions, regardless of the process's source-code implementation of the action. We tested multiple malicious bots and benign programs and found that we were able to thoroughly identify high-level behaviors across a diverse code base. In addition, we are able to distinguish malicious execution of high-level behaviors from benign ones by distinguishing remotely-initiated from locally-initiated actions.

References

1. Symantec Internet Security Threat Report, Trends for January-June 07, Volume XII (September 2007)
2. Keizer, G.: Bot Networks Behind Big Boos. In: Phishing Attacks. TechWeb (November 2004)
3. Parizo, E.: New bots, worm threaten AIM network. SearchSecurity (December 2005)
4. Naraine, R.: Money Bots: Hackers Cas. In on Hijacked PCs. eWeek (September 2006)
5. Overton, M.: Bots and Botnets: Risks, Issues, and Prevention. In: Virus Bulletin Conference (October 2005)
6. Ianelli, N., Hackworth, A.: Botnets as a Vehicle for Online Crime. CERT Coordination Center (December 2005)
7. Ilett, D.: Most spam generated by botnets, says expert. ZDNet UK (September 22, 2004)
8. Christodorescu, M., Jha, S.: Testing Malware Detectors. In: Proc. of the International Symposium on Software Testing and Analysis (July 2004)
9. SRI Honeynet and BotHunter Malware Analysis Automatic Summary Analysis
10. Jevans, D.: The Latest Trends in Phishing, Crimeware and Cash-Out Schemes. Private correspondence
11. Christodorescu, M., Jha, S., Kruegel, C.: Mining specifications of malicious behavior. In: Proc. of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (August 2007)
12. NoAH Foundation: Containment Environment Design
13. Chen, P., Noble, B.: When Virtual is Better than Real. In: Proceedings of HotOS-VIII: 8th Workshop on Hot Topics in Operating Systems
14. Petritsch, H.: Understanding and Replaying Network Traffic in Windows XP for Dynamic Malware Analysis. Master's Thesis (February 2007)
15. Christodorescu, M., Jha, S., Seshia, S., Song, D., Bryant, R.: Semantics-Aware Malware Detection. In: IEEE Symposium on Security and Privacy (May 2005)
16. Stinson, E., Mitchell, J.: Characterizing Bots' Remote Control Behavior. In: Proc. of the 4th DIMVA Conference (July 2007)
17. Newsome, J., Song, D.: Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software. In: Network and Distributed Systems Symposium (February 2005)
18. Yin, H., Song, D., Egele, M., Kruegel, C., Kirda, E.: Panorama: capturing system-wide information flow for malware detection and analysis. In: Proc. of the 14th ACM conference on Computer and communications security (October 2007)
19. Cui, W., Katz, R., Tan, W.: BINDER: An Extrusion-based Break-in Detector for Personal Computers. In: Proc. of the 21st Annual Computer Security Applications Conference (December 2005)
20. Kirda, E., Kruegel, C., Banks, G., Vigna, G., Kemmerer, R.: Behavior-based Spyware Detection. In: Proc. of the 15th USENIX Security Symposium (August 2006)
21. United States Patent Application 20070067843 M̈ethod and apparatus for removing harmful software: Williamson, Matthew; Gorelik, Vladimir (March 22, 2007)
22. Strider GhostBuster Rootkit Detection

23. Wang, Y., Beck, D., Vo, B., Roussev, R., Verbowski, C.: Detecting Stealth Software with Strider GhostBuster. Microsoft Technical Report MSR-TR-2005-25
24. Garfinkel, T.: Traps and Pitfalls: Practical Problems in System Call Interposition Based Security Tools. In: Network and Distributed System Security (Feburary 2003)
25. Garfinkel, T., Rosenblum, M.: A Virtual Machine Introspection Based Architecture for Intrusion Detection. In: Network and Distributed Systems Symp. (Feburary 2003)
26. Nilsson, N.: Problem-Solving Methods in Artificial Intelligence. McGraw-Hill, New York (1971)
27. Bayer, U., Moser, A., Kruegel, C., Kirda, E.: Dynamic Analysis of Malicious Code. Journal in Computer Virology 2(1) (August 2006)
28. Jiang, X., Xu, D., Wang, X.: 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 (CCS 2007), Alexandria, VA (November 2007)
29. Jiang, X., Wang, X.: 'Out-of-the-box' Monitoring of VM-based High-Interaction Honeypots. In: Kruegel, C., Lippmann, R., Clark, A. (eds.) RAID 2007. LNCS, vol. 4637, pp. 198–218. Springer, Heidelberg (2007)
30. Egele, M., Kruegel, C., Kirda, E., Yin, H., Son, D.: Dynamic Spyware Analysis. In: Proceedings of Usenix Annual Technical Conference, USA (June 2007)
31. Moser, A., Kruegel, C., Kirda, E.: Exploring Multiple Execution Paths for Malware Analysis. In: Proceedings of IEEE Symposium on Security and Privacy, May 2007, IEEE Computer Society Press, USA (2007)
32. Brumley, D., Hartwig, C., Liang, Z., Newsome, J., Poosankam, P., Song, D., Yin, H.: In: Lee, W., et al. (eds.) Botnet Analysis (2007)
33. Norman Sandbox
34. Willems, C.: Automatic Behaviour Analysis of Malware. Master Thesis. University of Mannheim
35. Jones, S.: Implicit Operating System Awareness in a Virtual Machine Monitor. Ph.D. Thesis, University of Wisconsin - Madison (April 2007)
36. Jones, S., Arpaci-Dusseau, A., Arpaci-Dusseau, R.: VMM-based Hidden Process Detection and Identification using Lycosid. In: ACM International Conference on Virtual Execution Environments (March 2008)
37. Vasudevan, A., Yerraballi, R.: Cobra: Fine-grained Malware Analysis using Stealth Localized-executions. In: Proceedings of IEEE Symposium on Security and Privacy, May 2006, IEEE Computer Society Press, USA (2006)
38. Bellard, F.: QEMU Accelerator (KQEMU)
39. Bellard, F.: QEMU, a Fast and Portable Dynamic Translator

A Study of the Packer Problem and Its Solutions

Fanglu Guo, Peter Ferrie, and Tzi-cker Chiueh

Symantec Research Laboratories

Abstract. An increasing percentage of malware programs distributed in the wild are packed by packers, which are programs that transform an input binary's appearance without affecting its execution semantics, to create new malware variants that can evade signature-based malware detection tools. This paper reports the results of a comprehensive study of the extent of the packer problem based on data collected at Symantec and the effectiveness of existing solutions to this problem. Then the paper presents a generic unpacking solution called Justin (Just-In-Time AV scanning), which is designed to detect the end of unpacking of a packed binary's run and invoke AV scanning against the process image at that time. For accurate end-to-unpacking detection, Justin incorporates the following heuristics: Dirty Page Execution, Unpacker Memory Avoidance, Stack Pointer Check and Command-Line Argument Access. Empirical testing shows that when compared with SymPack, which contains a set of manually created unpackers for a collection of selective packers, Justin's effectiveness is comparable to SymPack for those binaries packed by these supported packers, and is much better than SymPack for binaries packed by those that SymPack does not support.

1 The Packer Problem

1.1 Overview

Instead of directly obfuscating malware code, malware authors today heavily rely on packers, which are programs that transform an executable binary into another form so that it is smaller and/or has a different appearance than the original, to evade detection of signature-based anti-virus (AV) scanners. In many cases, malware authors recursively apply different combinations of multiple packers to the same malware to quickly generate a large number of different-looking binaries for distribution in the wild. The fact that more and more malware binaries are packed seriously degrades the effectiveness of signature-based AV scanners; it also results in an exponential increase in AV signature size, because when an AV vendor cannot effectively unpack a packed threat, it has no choice but to create a separate signature for the threat.

The percentage of malicious programs (malware) and benign applications (goodware) that are packed is hard to measure accurately. The numbers reported vary from vendor to vendor, but it is generally accepted that over 80% of malware is packed. These malware samples are often "wrapped" rather than packed, because many packers alter the original form of input binaries in ways that don't necessarily involve compression.

Not all packed programs are malware. We took a random sample of tens of thousands of executable files that were collected over a period of several months and were packed by packers that Symantec recognizes and knows how to unpack, and ran a set of commercial anti-virus (AV) scanners from multiple vendors against them. About 65% of these executable files are known malware. The remaining 35% most likely falls into the goodware category because these samples were collected more than a year ago and today's AV scanners should be able to capture most malware programs during that period of time. Clearly, the use of packers to protect goodware is quite common too.

The number of known packers is also hard to measure accurately. Symantec has collected a large number of packers - more than 2000 variants in more than 200 families. Among them, Symantec currently can identify nearly 1200 packers spread among approximately 150 families. However, among the 150 packer families Symantec knows about, it can only unpack about 110 of them, which contain approximately 800 members. This means that Symantec has a backlog of approximately 1200 members in 90 families, and this number increases day by day.

Without doubt, UPX [1] remains the most widely used packer. The rest of the list depends on how files are collected, but it always includes the old favorites like ASPack [2], FSG [3], and UPack [4]. In addition to those known packers, analysis of the above randomly sampled file set revealed at least 30 previously unknown packers. Some were minor variations of known packers, but most were custom packers. Amazingly, some of the clean files were packed with these custom packers.

The traditional way an AV vendor such as Symantec handles packers involves the following steps:

1. Recognize a packer's family. This is not as simple as it sounds. There are plenty of packers whose code is constant, and these can be recognized using simple strings. But many packers use polymorphic code to alter their appearance, and some packers intentionally use fake strings from other packers or standard compiler code, in order to fool the recognizers.
2. Identify a packer's version. A packer is classified into an existing version or is assigned to a new version. Being able to identify a packer is essential for successful unpacking, because there can be enough variations among members of the same family such that an unpacker for one member of a family cannot be used for another member for the same family.
3. Create a recognizer. The previous two steps are usually handled by a human, or applications such as neural net programs that have been trained on packers that are assigned to known families. This step, in contrast, is the act of writing a program whose function is solely to recognize that family, and perhaps that particular member/version.
4. Create an unpacker. Unlike the recognizer, whose goal is just to recognize the packer, the unpacker actually performs the reverse actions of the corresponding packer, and restores a packed binary as much as possible to its original form, including its metadata such as PE header for Win32 binaries.

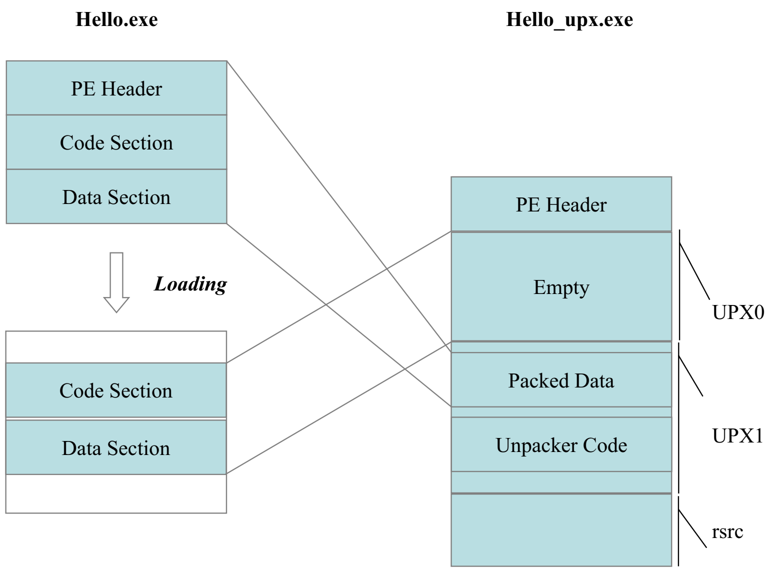


Fig. 1. A diagram that shows how an example program Hello.exe is packed by packer UPX and the layout of the resulting packed binary Hello_upx.exe

It requires a non-trivial amount of efforts to develop packer recognizers and unpackers. As noted above, Symantec has a backlog of approximately 1200 members in 90 families. To add unpacking support for a typical packer takes about six hours, on average. This means that it would take five full-time engineers about six months to clear the backlog if two unpackers are developed per day. However, in the case of complex packers such as Themida [5], it alone may take an experienced engineer up to six months to develop its unpacker. Packers with this level of complexity are not rare.

Due to these obvious disadvantages, the Justin (Just-In-Time AV scanning) solution presented in this paper takes a totally different approach. Justin leverages generic behaviors of unpacking and unpacks arbitrary packers without the need of knowing any information that is specific to the particular packer. Thus Justin doesn't have to go through any steps of the above traditional approach.

1.2 How Packers Work

Let's start with UPX, which arguably is among the most straightforward packers in use today. Figure 1 shows how UPX packs an example program Hello.exe.

When UPX compresses a PE binary, it begins by merging all of its sections into a single section, with the exception of the resource section. The combined data is then compressed into a single section of the resulting packed binary. In Figure 1, the code section and data section of hello.exe is compressed and stored in the Packed Data area of section UPX1 of the resulting binary Hello_upx.exe.

The resulting binary Hello_upx.exe contains three sections. The first section UPX0 is entirely virtual - it contains no physical data and is simply a placeholder.

It reserves the address range when Hello.exe is loaded to memory. At run time, Hello.exe will be restored to section UPX0. The second section contains the Packed Data, followed immediately by the Unpacker Code. The entry point in the PE header of Hello_upx.exe is changed to point directly to the Unpacker Code. The third section contains the resource data, if the original binary had a resource section, and a partial import table, which contains some essential imports from `kernel32.dll` as well as one imported function from every DLL used by the original binary.

The first two sections are set to read/write/execute, regardless of what they were before, and are not changed at run time. Therefore UPX is NX compatible, but it loosens up the protection for the original binary's read-only sections. The third section is simply set to read/write, since no execution should happen within that section.

After unpacking, UPX write-enables the header of the resulting binary, then changes the first two sections of the section table to read-only, and write-protects the header again. This ensures compatibility with some application programs that check in-memory section table instead of the actual section attributes, because these sections are supposed to be non-writable.

More sophisticated packers use a variety of techniques that virus writers use to defeat attempts to reverse-engineer, bypass, and disable the unpackers included in packed binaries. We discuss some of them in the following.

Multi-layer packing uses a combination of potentially different packers to pack a given binary, and makes it really easy to generate a large number of packed binaries from the same input binary. In practice, packed binaries produced by some packers may not be packed again by other packers. Also, the use of multi-layer packing itself could be used as an indication of malware, so the very presence of multiple layers - supported or not - could allow for a heuristic detection.

Anti-unpacking techniques are designed to make it difficult to uncover the logic of an unpacker, and fall into two major categories: passive and active. Passive anti-unpacking techniques are intended to make disassembly difficult, which in turns makes it difficult to identify and reverse the unpacking algorithm. Active techniques are intended to protect the running binary against having the fully unpacked image intercepted and extracted, and can be further classified into three subcategories: *anti-dumping*, *anti-debugging*, and *anti-emulating*. There are several commercial packers, such as Enigma and Themida, which promote their use of all of these techniques.

The simplest way to capture an unpacked image is to dump the address space of a running process. The simplest form of anti-dumping involves changing the value of the image size in the process environment block, and makes it difficult for a debugger to attach to the process or to dump the correct number of pages. More advanced anti-dumping methods include page-level protections, where each page is packed individually and unpacked only when accessed. It can even be packed again afterwards. This technique is used by packers such as Armadillo [6]. Shrinker [7] uses a variation of this method, by unpacking regions when they are accessed, but it is perhaps for performance reasons rather than an anti-dumping mechanism, since the unpacked pages remain in memory.

A very common way to capture an unpacked image is to use a debugger to step through the code, or to set breakpoints at particular locations. Two common forms of anti-debugging involve checking some values that the operating system supplies in the presence of a debugger. The first uses a public API, called `IsDebuggerPresent()`, which returns a Boolean value that corresponds to the presence or absence of a debugger. This technique is defeated by always setting the value to FALSE. The second anti-debugging technique checks if certain bits are set within the `NtGlobalFlag` field. The values of interest are heap tail checking (0x10), heap free checking (0x20), and heap parameter checking (0x40). They get their values from the `GlobalFlag` field of the `HKLM/System/CurrentControlSet/Control/Session Manager` registry key. A debugged process always has these values set in memory, regardless of the values in the registry. This technique can be defeated by clearing the bits in the process environment block.

Another way to capture an unpacked image to use an emulator to execute it in a protected environment. There are many ways to attack an emulator. The most common is to attempt to detect the emulator, since it is very hard to make an emulator whose behavior matches closely to real machine. However, each emulator has different capabilities, so there are multiple methods to detect different emulators [8].

Not all protection methods restore the host to its original form when executed. In particular, wrappers such as VMProtect [9] replace the host code with byte-code, and attach an interpreter to execute that byte-code. The result is that the original host code no longer exists anywhere, making it hard to analyze and essentially impossible to reverse. In addition, the byte-code have different meanings in different files. That is, the value 0x01 might mean "add" in one VMProtect-packed binary, but "xor" in another, and only the corresponding embedded interpreter knows for sure.

2 Unpacking Solutions from the Anti-Virus Industry

The AV industry has developed several approaches to tackle the packer problem, which satisfy different combinations of the following requirements:

- Effective: An ideal unpacker should restore packed binaries to their original form.
- Generic: An ideal unpacker should cover as many different types of packed binaries as possible.
- Safe: Execution of an ideal unpacker should not leave any undesirable side effects.
- Portability: An ideal unpacker should be able to run on multiple operating systems.

The first requirement enables existing signature-based AV scanners to be directly applied to an unpacker's output and detect the embedded malware if applicable. The second requirement decreases the amount of efforts required to keep up with new packers. The third requirement is crucial for at-rest file scanning, where the AV scanner initiates the unpacker and therefore has to be absolutely sure that the unpacker itself does not cause any harm. The final requirement is relevant for in-network scanning, where the unpacker and the AV scanner may need to run on different platforms than that required by the packed binaries.

The first solution to the packer problem is to the traditional way which manually creates recognizers and unpackers by reverse-engineering the unpackers in packed binaries by following steps as outlined in Section 1.1. The SymPack library [10] from Symantec falls into this category. This solution is safe, portable, largely effective but not generic. That is, one needs to develop a packer recognizer and an unpacker for each distinct packer. Given a set of packer recognizers and unpackers, one can classify packed binaries into four categories: (1) packed binaries whose packer can be recognized and that can be unpacked, (2) packed binaries whose packer can be recognized but that cannot be unpacked, (3) packed binaries whose packer cannot be recognized, and (4) non-packed binaries. Assuming all the packers that goodware programs use fall into the first category, then one can black-list all packed binaries belonging to the second and third category. To be able to distinguish between packed and non-packed binaries, one needs a technique to detect packed files generically. This can be done by, for example, calculating the entropy [11] of a particular region of an executable binary that most likely contains compressed data.

However, this general approach of handling packed binaries has several problems. First, it entails significant investments in engineering efforts, and the level of investment required is expected to increase over time as more packers appear in the wild. Second, if goodware decides to use packers in the second and third category above, false positives in the form of blocking legitimate goodware may arise. The same problem may also occur when packer recognizers and unpackers contain design or implementation bugs that, for example, treat variants of packers used by goodware as unknown packers. Finally, this approach requires continuing maintenance for existing packer recognizers and unpackers. Old packers never die. They just get rediscovered and reused, and never quite go away. For example, the self-extractor stub for RAR - the world's second-most popular archiving format after ZIP - is packed by UPX v0.50, which dates from 1999.

The second solution to the packer problem is to run a packed binary inside an emulator for a sufficiently long period of time so that the embedded binary is fully unpacked, and then invoke signature-based AV scanners against the memory image to check if it contains any malware. The x86 Emulator [12] take this approach. This solution is safe, portable and generic, but is not always effective [13] for two reasons. First, a packed binary can terminate itself before the embedded binary is unpacked if it detects that it is running inside an emulator. Second, so far there is no good heuristic to decide when it is safe to stop the emulation run of a packed binary, because it is difficult to distinguish between the following two cases: (1) the embedded binary is benign and (2) the embedded binary is malicious but is not fully unpacked. Another disadvantage of this approach is that it takes a non-trivial amount of effort to develop a high-fidelity and high-performance emulator.

The third solution to the packer problem is to invoke AV scanning against a suspicious running process's memory image either periodically or at certain security-sensitive events. Symantec's Eraser dump [10] takes this approach. This solution is generic, somewhat effective, but neither safe nor portable. Its effectiveness is compromised by the facts that certain information required by AV scanners, such as the entry point, is not available, and that the memory image being scanned is not the same as that of an embedded binary immediately after it is loaded. For example, a malware program may contain an encrypted string in its binary file, and decrypt it at run time. If the encrypted string is part of its signature, periodic memory scanning may fail to detect the malware because the encrypted string is no longer in its memory image.

PolyUnpack [14] is a generic approach to the problem. It detects newly generated code by comparing if the current instruction sequence exists in the original program. Instructions are disassembled and single stepped to achieve the detection. Because both disassembling and single stepping are expensive, this approach incurs significant performance overhead.

Renovo [15] monitors each instruction and tracks if any the memory is overwritten. If any overwritten memory is executed, it is treated as one layer of unpacking. This approach instruments instructions and also incurs significant performance overhead.

OllyBone [16] tracks write and execution too. It improves performance by overloading the user/supervisor bit and exploiting the separation of data TLB and instruction TLB in the X86 architecture. Saffron [17] combines OllyBone's technique with Intel's PIN to build a tool that detects control transfers to dynamically created or modified pages, and dumps memory images at that time.

OmniUnpack [18] also relies on OllyBone for identifying executed pages and invokes AV scanning before every "dangerous" system call. In addition, it incorporates two additional optimizations to reduce the total number of AV scans. First, it invokes an AV scan only when there is a control transfer to a dynamically modified page between the previous and current dangerous system calls. Second, whenever an AV scanner is invoked, it only scans those pages that are modified since the last dangerous system call. OmniUnpack is generic and largely effective, but neither safe nor portable. In particular, the fact that it requires whole-binary scanning is incompatible with almost all existing commercial AV scanners, which scans only a selective portion of each binary. Moreover, it only works for running processes, but is not suitable for at-rest file scanning.

One common heuristic shared among PolyUnpack, Renovo, OllyBone, Saffron, OmniUnpack and Justin (descibed in the next section) is that a necessary condition of the end of unpacking is a control transfer to a dynamically created or modified page. However, there are important differences between Justin and these previous efforts. First, Justin includes a more complete set of heuristics to detect the end of unpacking, including unpacked code region make-up, stack pointer check, and command line argument check. Second, Justin includes several counter-measures that are designed to fend off evasion techniques that existing packers use. Finally, Justin leverages NX support rather than overloads the supervisor/user bit, and is more efficient to track page executions.

3 Justin: Just-in-Time AV Scanning

3.1 Design

Justin is designed to be generic, effective and safe, but is not portable. The key idea of Justin is to detect the end of unpacking during the execution of a packed binary and invoke AV scanning at that instant. In addition to triggering AV scanning at the right moment, Justin also aims to provide the AV scanner a more complete picture about the binary being scanned, specifically its original entry point.

A packed binary logically consists of three components, the unpacker, the packed binary, and the area to hold the output of the unpacker. Different packers arrange these components into different number of PE sections. The section containing the unpacker's output typically is relatively easy to identify because its reserved size is larger than that of its initialized data contained in the binary.

The initial design goal of Justin is to enforce the invariant that no code page can be executed without being scanned first. Its design is relatively straightforward: it first scans a packed binary at load time, runs the binary, keeps track of pages that are dynamically modified or created, and scans any such page when the program's control is transferred to it. This design relies on an AV scanner that does whole-binary scanning, and is not compatible with existing commercial AV scanners, which employ a set of heuristics (e.g., file type) to select a portion of a binary and scan only bytes in that portion.

To work with commercial AV scanners, the design goal of Justin is shifted to detecting the end of unpacking during the execution of a packed binary. In addition, it makes the following two assumptions about packers: (1) The address space layout of the program embedded within a packed binary after it is unpacked is the same as that if the program is directly loaded into memory, and (2) the unpacker in a packed binary completely unpacks the embedded program before transferring control to it. The majority of packers satisfy Assumption (1) because they are supposed to work on commercially distributed executable binaries, which generally do not come with a relocation table. They also satisfy Assumption (2) because they cannot guarantee 100% static disassembly accuracy and coverage [19]. Some packers do perform simple metamorphic transformation to the input binaries before packing them. These packers inherently can evade signature-based AV scanners even without packing and are thus outside the scope of Justin. These two assumptions make it feasible to apply standard file-based AV scanners with selective scanning to a packed binary's memory image at the end of unpacking.

When the unpacker in a packed binary completes unpacking the embedded program, it sets up the import address table, unwinds the stack, and transfers control to the embedded program's entry point. Therefore the necessary conditions for the execution of a packed binary to reach the end of unpacking are

– A control transfer to a dynamically created/modified page occurs.
– The stack is similar to that when a program is just loaded into memory.
– The command-line input arguments are properly set up on the stack.

Accordingly, Justin combines these conditions into a composite heuristic for detecting the end of unpacking during the execution of a packed binary as follows. Given a binary, Justin loads it, marks all its pages as executable but non-writeable, and starts its execution. During the execution, if a write exception occurs on a non-writeable page, Justin marks this page as dirty, turns it into non-executable and writeable and continues; if a execution exception occurs on a non-executable page, Justin invokes an AV scanner to scan the whole memory image, and turns the page into executable and non-writeable if the end-of-unpacking check concludes that the unpacking is not done. Note that the whole memory image is presented as a file and scanned by the AV scanner. This is different from OmniUnpack [18] which only scans dirty pages. By presenting the whole memory image as a file, Justin's output is compatible with existing commercial AV scanners and avoids the problem in which signature straddles page boundaries. For a non-packed binary, because no code page is generated or modified during its execution, it is impossible for an execution exception to occur on a dirty page and no AV scan will be triggered at run time. So the performance overhead of Justin for non-packed binaries is insignificant. The performance overhead of Justin for packed binaries, on the other hand, depends on the number of times in which the program's control is transferred to a newly created page during its execution.

The current Justin prototype leverages virtual memory hardware to identify control transfers to dynamically created pages. More specifically, it manipulates write and execute permissions of virtual memory pages to guarantee that a page is either writeable or executable, but never both. With write protection, Justin can track which pages are modified. With execute protection, Justin can detect which pages are executed. If a binary Justin tracks needs to modify the protection attributes of its pages in ways that conflict with Justin's setting, Justin records the binary's intentions but physically keeps Justin's own setting. If the binary later on queries the protection attributes of its pages, Justin should respond with the binary's intentions, rather than the physical settings.

Whenever a virtual memory protection exception occurs, Justin takes control and first checks if this exception is owing to its setting. If not, Justin simply delivers the exception to the binary being monitored; otherwise Justin modifies the protection attributes according to the above algorithm. To ensure that Justin is the first to respond to an exception, the exception handler component of Justin must be the first in the binary's vectored exception handler list.

To ensure that the original program in a packed binary can execute in the same environment, most unpackers unwind the stack so that when the embedded program is unpacked and control is transferred to it, the stack looks identical to that when the embedded program is loaded into memory directly. For example, assume that the initial ESP at the time when a packed binary is started is `0x0012FFC4`, then right after the unpacking is done and the unpacked code is about to be executed, the ESP should point to `0x0012FFC4` again. This rule applies to many unpackers and is widely used in manually unpacking practice. Justin automates this method by recording ESP's value at the entry point of a packed binary, and compares the ESP at every exception in which the program's control is transferred to a dynamically created page. The exception context of an exception contains all CPU registers at the time when the binary raises the exception.

When a PE binary is run with a set of command-line arguments, these arguments are first placed in heap by the loader and later copied to the stack by a piece of compiler-generated code included in the binary at the program start-up time. Based on this observation, one can detect the start of execution of the original binary embedded in a packed binary, which occurs short after the end of unpacking.

3.2 Implementation Details

Justin currently is implemented for Windows only. But the idea will also work on other operating systems. The core logic of Justin is implemented in an exception handler that is registered in every binary at the time when it starts. In addition, Justin contains a kernel component that intercepts system calls related to page protection attribute manipulation and query and "lies" properly so that its page status tracking mechanism is as transparent to the binary being monitored as possible. Justin leverages NX support [20] in modern Intel X86 processors and Windows OS to detect pages that are executed at run time. In theory, it is possible to use other bits such as supervisor bits for this purpose, as is the case with OmniUnpack [18] and OllyBone [16].

Because Justin enforces the invariant that a page is either executable or writeable but not both, it could lead to a live lock for a program that contains an instruction which modifies data in the same page. The live lock is an infinite loop of interleaved execution and write exceptions. To address this issue, Justin checks if a memory-modifying instruction and its target address are in the same page when a write exception occurs. If so, Justin sets the page writeable, single-steps this instruction, and sets the page non-writeable again. This mechanism allows a page to be executable and writeable simultaneously for one instruction, but after that Justin continues to enforce the invariant.

One way to escape Justin's invariant is to map two virtual pages to the same physical page, and set one of them as executable and non-writeable and the other as writeable and non-executable. With this set-up, the unpacker can modify the underlying physical page through the writeable virtual page and jump to the underlying physical page through the executable virtual page, without triggering exceptions. To defeat this evasion technique, Justin makes sure that the protection attributes of virtual pages which are mapped to the same physical page are set in the same way.

Instead of a PE section, an unpacker can put its output in a dynamically allocated heap area. To prevent unpacked binaries from escaping Justin's tracking, Justin tracks pages in the heap, even when it grows. Similarly when a file is mapped into a process's address space, the mapped area needs to be tracked as well.

An unpacker can also put its output in a file, and spawns a process from the file later on. In this case, Justin will not detect any execution exception, because the generated code is invoked through a process creation mechanism rather than a jump instruction. Fortunately, standard AV scanners can detect this unpacked binary file when it is launched.

After recreating the embedded binary, some packers fork a new process and in the new process jumps to the embedded binary. This evasion technique is effective because page status tracking of a process is not necessarily propagated to all other processes it creates. Justin defeats this technique by tracking the page protection status of a process and that of all of its descendant processes.

Some unpackers include anti-emulation techniques that attempt to determine if they run inside an emulator or are being monitored in any way. Because Justin modifies page protection attributes in ways that may differ from the intentions of these unpackers, sometimes it triggers their anti-emulation techniques and results in program termination. For example, one unpacker detects if a page is writeable by passing a buffer that is supposedly writeable and Justin marks as non-writeable into the kernel as a system call argument. When the kernel attempts to write to the buffer, a kernel-level protection exception occurs and the program terminates. Justin never has a chance to handle this exception because it is a kernel-level exception and never gets delivered to the user level. To solve this problem, Justin intercepts this kernel-level protection exception, modifies the page protection attribute appropriately to allow it to continue, and changes it back before the system call returns.

When Justin detects the end of unpacking, it treats the target address of the control transfer instruction as the entry point of the embedded binary. However, some packers obfuscate the original entry point by replacing the first several instructions at the main entry point with a jump instruction, say Y, to a separate piece of code, which contains the original entry point instructions and a jump back to the instruction following Y. Because an unpacker can only safely replace the first several instructions, Justin can single-step the first several instructions at the supposedly entry point to specifically detect this evasion technique.

Some packers significantly transform an input binary before packing it. In general, these transformations are not always safe, because it requires 100% disassembly accuracy and coverage, which is generally not possible. Therefore, although these packers may evade signature-based AV scanners after Justin correctly produces the unpacked binary, we generally consider these packers to be too unreliable to be a real threat.

4 Evaluation

4.1 Effectiveness of Justin

To assess the effectiveness of Justin, we collect a set of known malware samples that are not packed by any known packers, then use different packers to pack

Table 1. Effectiveness comparison between Justin and manually created unpackers from SymPack when they are used together with an AV scanner

Packers	Packed	Justin Unpack Failure	Justin Detection Failure	Justin Detection	SymPack Detection	Justin Detection Improvement
ASPack	182	4	0	178	182	-4
BeroPacker	178	0	4	174	161	13
Exe32Pack	176	32	0	144	176	-32
Mew	180	1	8	171	171	0
PE-Pack	176	1	0	175	171	4
UPack	181	1	5	175	173	2

them, and run the packed binaries under Justin and Symantec's AV scanner to see if they together can detect these samples. As a comparison, we used the same procedure but replaced Justin with Symantec's SymPack library, which contains a set of unpacker routines created manually by reverse engineering the logic of known packer programs. This experiment tests if Justin can unpack packed binaries to the extent that AV signatures developed for non-packed versions of malware samples still work.

There are totally 183 malware samples used in this study. As shown in Table 1, most packers cannot pack every malware program in the test suite successfully. So only successfully packed malware programs are unpacked. The number of successfully packed malware programs for each packer is listed in Column 2 of Table 1.

Justin cannot unpack certain packed samples. By manually examining each failure case, we identify two reasons. First, some samples simply cannot run any more after being packed. Being a run-time detection technology, Justin cannot unpack something that does not run. From malware detection's standpoint, these packed samples are no longer a threat as they won't be able to cause any harm. Second, the packer Exe32Pack sometimes doesn't really modify the original binary when it produces a packed binary. For these packed binaries, no unpacking occurs at run time and Justin does not have a chance to step in and trigger the AV scan. From malware detection's standpoint, these packed samples are not a problem either. The original program in these samples are in plain-text and AV scanner can detect them without Justin. The number of packed malware programs that Justin fails to unpack is listed in Column 3 of Table 1.

Among those malware samples that Justin successfully unpacks, not all of them can be detected. By manually analyzing these undetected samples, we find that most detection failures arise because signatures developed for non-packed versions of malware programs do not work for their unpacked versions. Although Justin can detect the end of unpacking, the unpacked result it produces is not exactly the same as the original program. Because some AV signatures are too stringent to accommodate these minor differences, they fail to detect Justin's outputs. For the same reasons, none of these undetected samples cannot be detected by SymPack either. The number of unpacked but undetected samples is listed in Column 4 of Table 1.

Overall, Justin's detection rate (Column 5) is slightly higher than SymPack's (Column 6) among the malware samples that can be successfully unpacked, because Justin relies on the unpackers embedded in the packed binaries, which are generally more reliable than the manually created unpackers in SymPack, to capture the execution state of a malware before it starts to run.

To test Justin's generic unpacking capability, we select a set of 13 packers that are not supported by SymPack. Justin can successfully unpack binaries packed by 12 out of these 13 packers. The packer whose packed binaries Justin cannot unpack detects Justin's API call interception and terminates the packed binary's execution without unpacking the original program. We also test a set of malware samples packed by a packer that is not well supported by SymPack against Justin and an AV scanner. The number of these packed malware samples that can be detected by Justin/AV scanner is almost twice the number of SymPack/AV scanner.

To summarize, as long as a packed binary can run and requires unpacking at run time, Justin can unpack it successfully. Moreover, for the same malware samples packed by packers supported in SymPack, the unpacked outputs produced by Justin are more amenable to AV scanning than those produced by SymPack, thus resulting in a higher detection rate than SymPack. Finally, Justin is able to detect twice as many packed malware samples than SymPack when they are packed by packers not supported in SymPack.

4.2 Number of Spurious End-of-Unpacking Detections

When Justin detects an end of unpacking during a packed binary's execution, it invokes the AV scanner to scan the process image at that instant. The main heuristic that Justin uses to detect the end of unpacking is to monitor the first control transfer to a dirty page (called *Dirty Page Execution*). Unfortunately this heuristic triggers many spurious end-of-unpacking detections for binaries packed by certain packers and thus incurs a significant AV scanning overhead even for goodware packed by these packers. The same observation was made by Martignoni et al. [18]. Their solution to this problem is to defer AV scanning until the first "dangerous" system call. Even though this technique drastically decreases the number of spurious end-of-unpacking detections, it also loses the entry-point information, which plays an important role for commercial signature-based AV scanners.

Instead, Justin incorporates three addition heuristics to reduce the number of spurious end-of-unpacking detections. *Unpacker Memory Avoidance* limits the Dirty Page Execution technique to pages that are not likely to contain the unpacker code. *Stack Pointer Check* checks if the current stack pointer at the time of a first control transfer to a dirty page during a packed binary's run is the same as that at the very start of the run. *Command-Line Argument Access* checks if the command-line arguments supplied with a packed binary's run is moved to the stack at the time of a first control transfer to a dirty page. Each of these

Table 2. Comparison among four heuristics in their effectiveness to detect the end of unpacking, as measured by the number of times it thinks the packed binary run reaches the end of unpacking. The last three heuristics, Unpacker Memory Avoidance, Stack Pointer Check and Command-Line Argument Detection, are used together with the first heuristic, which monitors first control transfers to dirty pages.

Packers	**Dirty Page Execution**	**Unpacker Memory Avoidance**	**Stack Pointer Check**	**Command-Line Argument Access**
ACProtect	186	11	1	2
ASPack	96	12	2	3
ASProtect	1633	12	12	3
Exe32Pack	394	11	1	2
eXPressor	15	11	1	2
FSG	12	12	1	2
Molebox	3707	11	1	2
NsPack	19	11	1	2
Obsidium	not work	14	4	6
PECompact	16	12	2	3
UPack	442084	12	2	3
UPX	11	11	1	2
WWPack	12	11	1	2

three heuristics is meant to work in conjunction with the Dirty Page Execution heuristic.

We apply a set of packers to a set of test binaries, run these packed binaries under Justin, and measure the number of end-of-unpacking detections. Table 2 shows the average number of end-of-unpacking detections for each of these four heuristics. Used together, the three additional heuristics in Justin successfully reduces the number of spurious end-of-unpacking detections to the same level as Martignoni et al. [18], but in a way that still preserves the original program's entry point information.

Although the number of spurious end-of-unpacking detections produced by Unpacker Memory Avoidance is higher than the other two heuristics, it is more reliable and resilient to evasion. If Justin mistakes a normal page as an unpacker page, it will not monitor this page, and the worst that can happen is that Justin loses the original program's entry point if this page happens to contain the original entry point. If Justin mistakes an unpacker page as a normal page, it will monitor this page, and the worst that can happen is additional spurious end-of-unpacking detections. Currently, Justin is designed to err on the conservative side and therefore is tuned to treat unpacker pages as normal pages rather than the other way around.

We test the Stack Pointer Check heuristic using the packers listed in Table 2. Column 4 of Table 2 shows the average number of end-of-unpacking detections for each packer tested is decreased to just one or two for most packers. Unfortunately, this heuristic generates false negatives but no false positive. A false positive occurs when a certain execution point passes the stack pointer check but it is not the end of unpacking. This happens when the unpacker intentionally manipulates the stack pointer to evade this heuristic. None of the packers we tested exhibit this evasion behavior. A false negative happens when Justin thinks an execution point is not the end of unpacking when in fact it is. This happens when the unpacker does not clean up the stack to the exactly same state when the unpacker starts. The unpacker in ASProtect-packed binaries doesn't completely clean up the stack before transferring control to the original binary. It is possible to loosen up the stack pointer check, i.e., as long as the stack pointers are roughly the same, to mitigate this problem, but this is not a robust solution and may cause false positives.

The key idea in Command-Line Argument Access is that when the original binary embedded in a packed binary starts execution, there is a piece of compiler-generated code that will prepare the stack by fetching command-line arguments. Therefore, if at an execution point the command-line arguments supplied to a packed binary's run are already put on the stack, that execution point must have passed the end of unpacking. This command-line argument access behavior exists event if the original binary is not designed to accept any command-line arguments. Because Justin gets to choose the values for command-line arguments, it detects command-line argument access by searching the stack for pointers that point to values that it chooses as command-line arguments.

We test the Command-Line Argument Access heuristic using the packers listed in Table 2. Column 5 of Table 2 list the average number of end-of-unpacking detections for each packer tested, which is generally higher than the Stack Pointer Check heuristic for the following reason. To put command-line arguments on the stack, the original program needs to execute a couple of new generated code pages. The execution of the new generated pages causes one or two more end-of-unpacking detections. Even though its reported number of end-of-unpacking detections is slightly higher, the Command-Line Argument Access heuristic does not generate any false positive or false negative. For example, it can accurately detect the end of unpacking for ASProtect-packed binaries, but the Stack Pointer Check heuristic cannot.

4.3 Performance Overhead of Justin

Justin is designed to work with an AV scanner to monitor the execution of binaries. Its performance penalty comes from two sources: (1) additional virtual memory protection exceptions that are triggered during dirty page tracking, and (2) AV scans invoked when potential ends of unpacking are detected. We packed Microsoft Internet Explorer, whose binary size is 91KB, with a set of packers, ran the packed version, and measured its start-up delay with and without Justin on a 3.2GHz Pentium-4 machine running Windows XP. The start-up delay is defined as the interval between when the IE process is created and when it calls the Win32 API CreateWindowEx function, which creates the first window. The start-up time excludes the program load time, which involves disk access, so

Table 3. The average additional start-up delays for Microsoft Internet Explorer (IE) when it is packed by a set of packers and run under Justin and an AV scanner. The additional delay is dominated by AV scanning, which is mainly determined by the number of AV scans invoked during a packed binary's run.

Packers	**Number of AV Scans**	**Original Delay (msec)**	**Extra Delay (msec)**	**Extra Delay %**
ACProtect	2	46	4.2	9.1
ASProtect	3	62	9.0	14.6
eXPressor	2	62	5.5	8.8
FSG	2	62	4.2	6.8
Molebox	2	31	4.2	13.5
NsPack	2	46	4.5	9.9
Obsidium	6	31	12.1	38.7
PECompact	3	62	5.8	9.3
UPX	2	31	4.1	13.1

that we can focus on the CPU overhead. After the first window is created, a packed GUI application must have been fully unpacked, and there will not be any additional protection exceptions or AV scans from this point on. The AV scanner used in this study runs at 40 MB/sec on the test machine, and is directly invoked as a function call.

Table 3 shows the base start-up delay and the additional start-up delay for IE when it is packed by a set of packers and runs under Justin and an AV scanner. Overall, the absolute magnitude of the additional start-up delay is quite small. Justin only introduces several milliseconds of additional delay under most packers. The largest additional delay occurs under Obsidium and is only around 12 msec. At least for IE, the additional start-up delay that Justin introduces is too small to be visible to the end user.

Most of the additional start-up delay comes from AV scanning, because the additional delay becomes close to zero when the AV scan operation is turned into a no-op. This is why the additional start-up delay correlates very well with the number of AV scans invoked. More specifically, the additional start-up delay for a packer is the product of the AV scanning speed, the number of scans, and the size of the memory being scanned. Because the amount of memory scanned in each AV scan operation may be different for binaries packed by different packers, the additional delay is different for different packers even though they invoke the same number of AV scans.

We also try other GUI programs such as Microsoft NetMeeting, whose binary is around 1 MB, and the additional delay results are consistent with those associated with IE. The performance overhead associated with additional protection exceptions is still negligible. Because of a larger binary size, the performance cost of each AV scan is higher. On a typical Windows desktop machine, more than 80% of its executable binaries is smaller than 100 KB. This means that the additional start-up delay when they are packed and run under Justin will be similar to that of IE and thus not noticeable. Finally, for legitimate programs that are not packed, no AV scanning will be triggered when they run under Justin, so there is no performance overhead at all.

5 Conclusion

Packer poses a serious problem for the entire AV industry because it significantly raises the bar for signature-based malware detection. Existing solutions to the packer problem do not scale because they require either expensive manual reverse engineering efforts or creation of separate signatures for variants of the same malware. In this paper, we report the result of a detailed study of the packer problem and its various solutions described in the literature, taking into account practical requirements and design considerations when integrating such solutions with commercial AV products. In particular, we describe a solution to the packer problem called Justin (JUST-IN-time AV scanning), which aims to detect the end of unpacking during the execution of a packed binary in a packer-independent way and invoke AV scanning against the binary's run-time image at that moment. Towards that end, Justin incorporates the following heuristics: first control transfer to dirty pages, avoiding tracking unpacker pages, checking for stack unwinding, and detection of command-line input argument access. More concretely, this paper makes the following contributions to the field of malware detection:

- A detailed analysis of the extent of the packer problem and the packing and evasion technologies underlying state-of-the-art packers,
- A set of heuristics that collectively can effectively detect the end of unpacking during the execution of packed binaries without any a priori knowledge about their packers,
- A comprehensive set of countermeasures against anti-unpacking evasion techniques built into modern packers, and
- A fully working Justin prototype and a thorough evaluation of its effectiveness and performance overhead.

Overall, Justin's effectiveness at detecting packed malware is excellent and its performance overhead for packed goodware is minimal. However, this paper will not be the final chapter on the packer problem. If anything, experiences tell us that the packer community will sooner or later shift to a different set of tactics to evade Justin's detection techniques. So the search for better solutions to the packer problem is expected to continue for the next few years.

References

1. Oberhumer, M.F., Molnár, L., Reiser, J.F.: UPX: the Ultimate Packer for eXecutables (2007), http://upx.sourceforge.net/
2. ASPACK SOFTWARE, ASPack for Windows, (2007), http://www.aspack.com/aspack.html

3. bart, FSG: [F]ast [S]mall [G]ood exe packer, (2005), http://www.xtreeme.prv.pl/
4. Dwing, WinUpack 0.39final, (2006), http://dwing.51.net/
5. Oreans Technology, Themida: Advanced Windows Software Protection System, (2008), http://www.oreans.com/themida.php
6. Silicon Realms, Armadillo/SoftwarePassport (2008), http://www.siliconrealms.com/
7. Blinkinc,Shrinker 3.4, (2008), http://www.blinkinc.com/shrinker.htm
8. Ferrie, P.: Attacks on Virtual Machines. In: Proceedings 9th Annual AVAR International Conference (2006)
9. VMProtect, VMProtect (2008), http://www.vmprotect.ru/
10. Symantec Corporation (2008), http://www.symantec.com/
11. Lyda, R., Hamrock, J.: Using entropy analysis to find encrypted and packed malware. IEEE Security and Privacy 5(2), 40–45 (2007)
12. Prakash, C.: Design of X86 Emulator for Generic Unpacking. In: Proceedings of 10th Annual AVAR International Conference (2007)
13. Tan, X.: Anti-unpacker Tricks in Malicious Code. In: Proceedings of 10th Annual AVAR International Conference (2007)
14. Royal, P., Halpin, M., Dagon, D., Edmonds, R., Lee, W.: Polyunpack: Automating the hidden-code extraction of unpack-executing malware. In: ACSAC 2006: Proceedings of the 22nd Annual Computer Security Applications Conference on Annual Computer Security Applications Conference, pp. 289–300 (2006)
15. Kang, M.G., Poosankam, P., Yin, H.: Renovo: A hidden code extractor for packed executables. In: Proceedings of the 5th ACM Workshop on Recurring Malcode (WORM) (Oct. 2007)
16. Stewart, J.: OllyBonE v0.1, Break-on-Execute for OllyDbg (2006), http://www.joestewart.org/ollybone/
17. Quist, D., Valsmith,: Covert Debugging: Circumventing Software Armoring. In: Proceedings of Black Hat USA (2007)
18. Martignoni, L., Christodorescu, M., Jha, S.: OmniUnpack: Fast, Generic, and Safe Unpacking of Malware. In: 23rd Annual Computer Security Applications Conference (ACSAC) (2007)
19. Nanda, S., Li, W., chung Lam, L., cker Chiueh, T.: BIRD: Binary Interpretation using Runtime Disassembly. In: Proceedings of the 4th IEEE/ACM Conference on Code Generation and Optimization (CGO 2006) (2006)
20. NX bit, http://en.wikipedia.org/wiki/NX_bit

Gnort: High Performance Network Intrusion Detection Using Graphics Processors

Giorgos Vasiliadis, Spiros Antonatos, Michalis Polychronakis,
Evangelos P. Markatos, and Sotiris Ioannidis

Institute of Computer Science, Foundation for Research and Technology – Hellas,
N. Plastira 100, Vassilika Vouton, GR-700 13 Heraklion, Crete, Greece
{gvasil,antonat,mikepo,markatos,sotiris}@ics.forth.gr

Abstract. The constant increase in link speeds and number of threats poses challenges to network intrusion detection systems (NIDS), which must cope with higher traffic throughput and perform even more complex per-packet processing. In this paper, we present an intrusion detection system based on the Snort open-source NIDS that exploits the underutilized computational power of modern graphics cards to offload the costly pattern matching operations from the CPU, and thus increase the overall processing throughput. Our prototype system, called *Gnort*, achieved a maximum traffic processing throughput of 2.3 Gbit/s using synthetic network traces, while when monitoring real traffic using a commodity Ethernet interface, it outperformed unmodified Snort by a factor of two. The results suggest that modern graphics cards can be used effectively to speed up intrusion detection systems, as well as other systems that involve pattern matching operations.

Keywords: GPU, pattern matching, intrusion detection systems, network security, SIMD, parallel programming.

1 Introduction

Network security architectures such as firewalls and Network Intrusion Detection Systems (NIDS) attempt to detect break-in attempts by monitoring the incoming and outgoing traffic for suspicious payloads. Most modern network intrusion detection and prevention systems rely on a set of rules that are compared against network packets. Usually, a rule consists of a filter specification based on packet header fields, a string that must be contained in the packet payload, the approximate or absolute location where that string should be present, and an associated action to take if all the conditions of the rule are met.

Signature matching is a highly computationally intensive process, accounting for about 75% of the total CPU processing time of modern NIDSes [2,7]. This overhead arises from the fact that most of the time, every byte of every packet needs to be processed as part of the string searching algorithm that searches for matches among a large set of strings from all signatures that apply for a particular packet. For example, the rule set of Snort [26], one of the most widely used open-source NIDS, contains about 10000 strings. Searching every packet for all of these strings requires significant resources, both in terms of the computation capacity needed to process a packet, as well as the amount of memory needed to store the rules.

Several research efforts have explored the use of parallelism for improving the packet processing throughput [25,8,14,4,37]. Specialized hardware devices can be used to inspect many packets concurrently, and such devices include ASICs and Network Processors. Both are very efficient and perform well, however they are complex to modify and program. Moreover, FPGA-based architectures have poor flexibility since most of the approaches are usually tied to a specific implementation.

As Graphics Processing Units (GPUs) are becoming increasingly powerful and ubiquitous, researchers have begun exploring ways to tap their power for non-graphic or general-purpose (GPGPU) applications. The main reason behind this evolution is that GPUs are specialized for computationally-intensive and highly parallel operations—required for graphics rendering—and therefore are designed such that more transistors are devoted to data processing rather than data caching and flow control [23]. The release of software development kits (SDKs) from big vendors, like NVIDIA[1] and ATI,[2] has started a trend of using GPUs as a computational unit to offload the CPU.

In addition, many attempts have been made to use graphics processors for security purposes, including cryptography [11], data carving [20] and intrusion detection [17]. Specifically, it has been shown that GPU support can substantially increase the performance of digital forensics software that relies on binary string searches [20]. Jacob and Brodley were the first that tried to use the GPU as a pattern matching engine for NIDS in PixelSnort [17]. They used a simplified version of the Knuth-Morris-Pratt (KMP) algorithm [18], however, their performance results indicated marginal improvement.

In this paper, we explore how GPUs can be used to speed up the processing throughput of intrusion detection systems by offloading the string matching operations to the GPU. We show that single pattern matching algorithms, like KMP, do not perform well when executed on the GPU, especially when using an increased number of patterns. However, porting multi-pattern matching algorithms, like the Aho-Corasick algorithm can boost overall performance by a factor of three. Furthermore, we take advantage of DMA and the asynchronous execution of GPUs to impose concurrency between the operations handled by the CPU and the GPU. We have implemented a prototype intrusion detection system that effectively utilizes GPUs for pattern matching operations in real time.

The paper is organized as follows: In the remainder of the Introduction we will give an overview of the GPU architecture that we used for this research. In Section 2 we will briefly present a survey of related work. Section 3 and 4 presents our prototype architecture and the implementation details respectively.

[1] http://developer.nvidia.com/object/cuda.html
[2] http://ati.amd.com/technology/streamcomputing/index.html

In Section 5 we evaluate our implementation and we compare with the previous work. Finally, in Section 6 we present some conclusions as well as some ideas for future work.

1.1 Overview of the GeForce 8 Series Architecture

In this Section we briefly describe the architecture of the NVIDIA GeForce 8 Series (G80) cards, which we have used for this work, as well as the programming capabilities it offers through the Compute Unified Device Architecture (CUDA) SDK. The G80 architecture is based on a set of multiprocessors, each of which contains a set of *stream processors* operating on SIMD (Single Instruction Multiple Data) programs. When programmed through CUDA, the GPU can be used as a general purpose processor, capable of executing a very high number of threads in parallel.

A unit of work issued by the host computer to the GPU is called a *kernel*, and is executed on the device as many different *threads* organized in *thread blocks*. Each multiprocessor executes one or more thread blocks, with each group organized into *warps*. A warp is a fraction of an *active group*, which is processed by one multiprocessor in one batch. Each of these warps contains the same number of threads, called the *warp size*, and is executed by the multiprocessor in a SIMD fashion. Active warps are time-sliced: A thread scheduler periodically switches from one warp to another to maximize the use of the multiprocessors' computational resources.

Stream processors within a processor share an instruction unit. Any control flow instruction that causes threads of the same warp to follow different execution paths reduces the instruction throughput, because different executions paths have to be serialized. When all the different execution paths have reached a common end, the threads converge back to the same execution path.

A fast *shared memory* is managed explicitly by the programmer among thread blocks. The *global*, *constant*, and *texture memory spaces* can be read from or written to by the host, are persistent across kernel launches by the same application, and are optimized for different memory usages [23]. The constant and texture memory accesses are cached, so a read from them costs much less compared to device memory reads, which are not being cached. The texture memory space is implemented as a read-only region of device memory.

GPGPU programming on G80 series and later is feasible using the CUDA SDK. CUDA consists of a minimal set of extensions to the C language and a runtime library that provides functions to control the GPU from the host, as well as device-specific functions and data types. CUDA exposes several hardware features that are not available via the graphics API. The most important of these features is the read and write access to the shared memory shared among the threads, and the ability to access any memory location in the card's DRAM through the general memory addressing mode it provides. Finally, CUDA also offers highly optimized data transfers to and from the GPU.

2 Related Work

Pattern matching is the most critical operation that affects the performance of network intrusion detection systems. Pattern matching algorithms can be classified into single- and multi-pattern algorithms.

In single pattern matching algorithms, each pattern is searched in a given text individually. This means that if we have k patterns to be searched, the algorithm must be repeated k times. Knuth-Morris-Pratt [18] and Boyer-Moore [6] are some of the most widely used single pattern matching algorithms. Knuth-Morris-Pratt is able to skip characters when a mismatch occurs in the comparison phase using a partial-match table for each pattern. Each table is built by preprocessing every pattern separately. Boyer-Moore is the most widely used single-pattern algorithm. Its execution time can be sublinear if the suffix of the string to be searched for appears infrequently in the input stream, due to the skipping heuristics that it uses.

Multi-pattern string matching algorithms search for a set of patterns in a body of text simultaneously. This is achieved by preprocessing the set of patterns and building an automaton that will be used in the matching phase to scan the text. The automaton can be thought of as a state machine that is represented as a trie, a table or a combination of the two. Each character of the text will be searched only once. Multi-pattern matching scales much better than algorithms that search for each pattern individually. Multi-pattern string matching algorithms include Aho-Corasick [1], Wu-Manber [36] and Commentz-Walter [10].

Most Network Intrusion Detection Systems (NIDS) use finite automata and regular expressions [26,24,16] to match patterns. Coit *et al.* [9] improved the performance of Snort by combining the Aho-Corasick keyword trie with the skipping feature of the Boyer-Moore algorithm. Fisk and Vaghese enhance the Boyer-Moore-Horspool algorithm to simultaneously match a set of rules. The new algorithm, called Set-wise Boyer-Moore-Horspool [15], was shown to be faster than both Aho-Corasick and Boyer-Moore for sets with less than 100 patterns. Tuck *et al.* [31] optimized the Aho-Corasick algorithm by applying bitmap node and path compression.

Snort from version 2.6 and onwards uses only flavors of the Aho-Corasick for exact-match pattern detection. Specifically, it contains a variety of implementations that are differentiated by the type of the finite automaton they use (NFA or DFA), and the storage format they use to keep it in memory (full, sparse, banded, trie, *etc.*). It should be mentioned, however, that the best performance is achieved with the full version that uses a deterministic finite automaton (DFA) at the cost of high memory utilization [30].

To speed-up the inspection process, many IDS implementations are based on specialized hardware. By using content addressable memory (CAM), which is suitable to perform parallel comparison for its contents against the input value, they are very well suited for use in intrusion detection systems [37,38]. However they have a high cost per bit.

Many reconfigurable architectures have been implemented for intrusion detection. Most approaches involve building an automaton for a string to be searched, generating a specialized hardware circuit using gates and flip-flops for the automaton, and then instantiating multiple such automata in the reconfigurable chip to search the streaming data in parallel. However, the circuit implemented on the FPGA to perform the string matching is designed based on the underlying hardware architecture to adjust to a given specific rule set. To adjust to a new rule set, one must program a new circuit (usually in a hardware description language), which is then compiled down through the use of CAD tools. Any changes in the rule set requires the recompilation, regeneration of the automaton, resynthesis, replacement and routing of the circuits which is a time consuming and difficult procedure.

Sidhu and Prasanna implemented a regular expression matching architecture for FPGAs [28]. Baker *et al.* also investigated efficient pattern matching as a signature based method [4]. In [13], the authors used hardware bloom filters to match multiple patterns against network packets at constant time. Attig *et al.* proposed a framework for packet header processing in combination with payload content scanning on FPGAs [3].

Several approaches attempt to reduce the amount of memory required to economically fit it in on-chip memory [4,31,14]. However, the on-chip hardware resource consumption grows linearly with the number of characters to be searched. In [29], the authors convert a string set into many tiny state machines, each of which searches for a portion of the strings and a portion of the bits of each string.

Other approaches involve the cooperation with network processors in order to pipeline the processing stages assigned to each hardware resource [8], as well as the entire implementation of an IDS on a network processor [5,12]. Computer clusters have also been proposed to offload the workload of a single computer [19,34,33,27]. The cost however remains high, since it requires multiple processors, a distribution network, and a clustered management system.

On the contrary, modern GPUs have low design cost while their increased programmability makes them more flexible than ASICs. Most graphic cards manufacturers provide high-level APIs that offer high programming capabilities and are further ensure forward compatibility for future releases, in contrast with most FPGA implementations that are based on the underlying hardware architecture and need to be reconfigured whenever a change occurs in the rule set. Furthermore, their low design cost, the highly parallel computation and the potential that are usually underutilized, especially in hosts used for intrusion detection purposes, makes them suitable for use as an extra low-cost coprocessor for time-consuming problems, like pattern matching.

The work most related to ours is PixelSnort [17]. It is a port of the Snort IDS that offloads packet matching to an NVIDIA 6800GT. The GPU programming was complicated, since the 6800GT did not support a general purpose programming model for GPUs (as the G80 used in our work). The system encodes Snort rules and packets to textures and performs the string searching using the KMP algorithm on the 16 fragment shaders in parallel. However, PixelSnort *does not* achieve *any* speed-up under normal-load conditions. Furthermore, PixelSnort

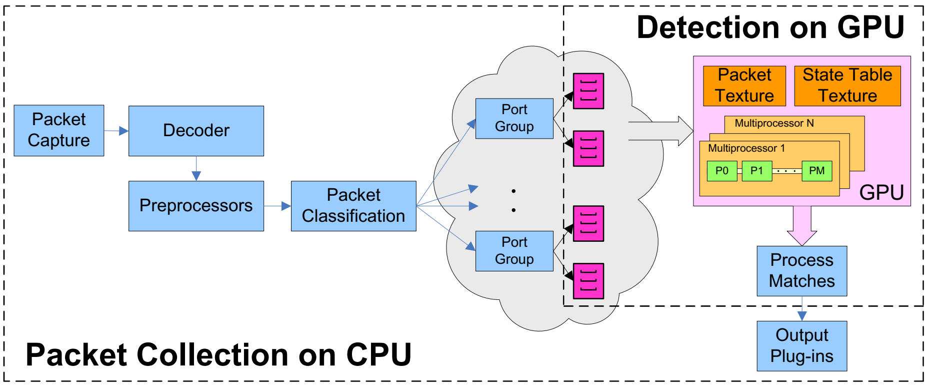


Fig. 1. Overall architecture of Gnort

did not have any multi-pattern matching algorithms ported to GPU. This is a serious limitation since multi-pattern matching algorithms are the default for Snort. In a more recent work, Marziale *et al.* [20] evaluated the effectiveness of offloading the processing of a file carving tool to the GPU. The system was implemented on the G80 architecture and the results show that GPU support can substantially increase the performance of digital forensics software that relies on binary string search.

3 Architecture

The overall architecture of Gnort, which is based on the Snort NIDS, is shown in Figure 1. We can separate the architecture of our system in three different tasks: the transfer of the packets to the GPU, the pattern matching processing on the GPU, and finally the transfer of the results back to the CPU.

3.1 Transferring Packets to the GPU

The first thing to consider is how the packets will be transferred from the network interface to the memory space of the GPU. The simplest approach would be to transfer each packet directly to the GPU for processing. However, due to the overhead associated with a data transfer operation to the GPU, batching many small transfers into a larger one performs much better than making each transfer separately [23]. Thus, we have chosen to copy the packets to the GPU in batches.

Snort organizes the content signatures in groups, based on the source and destination port numbers of each rule. A separate detection engine instance is used to search for the patterns of a particular rule group. Table 1 shows the number of rules that come with the latest versions of Snort and are enabled by default, as well as the number of groups in which they are organized. We use a separate buffer for temporarily storing the packets of each group. After a packet has been classified to a specific group, it is copied to the corresponding buffer.

Table 1. Snort Data Structures

Snort version	# Groups	# Rules
2.6	249	7179
2.7	495	8719
2.8	495	8722

Whenever the buffer gets full, all packets are transferred to the GPU in one operation. In case a buffer is not yet full after 100ms, its packets are explicitly transferred to the GPU.

The buffers are allocated as a special type of memory, called page-locked or "pinned down" memory. Page-locked memory is a physical memory area that does not map to the virtual address space, and thus cannot be swapped out to secondary storage. The use of pinned down memory results to higher data transfer throughput between the host and the device [23]. Furthermore, the copy from page-locked memory to the GPU is performed using DMA, without occupying the CPU. Thus, the CPU can continue working and collecting the next batch of packets at the same time the GPU is processing the packets of the previous batch.

To further improve parallelism, we use a double buffering scheme. When the first buffer becomes full, it is copied to a texture bounded array that can be read later by the GPU through the kernel invocation. While the GPU is performing pattern matching on the packets of the first buffer, the CPU will copy newly arrived packets in the second buffer.

3.2 Pattern Matching on the GPU

Once the packets have been transferred to the GPU, the next step is to perform the pattern matching operation. We have ported the Aho-Corasick algorithm [1] to run on the graphics card. The Aho-Corasick algorithm seems to be a perfect candidate for SIMD processors like a GPU. The algorithm iterates through all the bytes of the input stream and moves the current state to the next correct state using a state machine that has been previously constructed during initialization phase. The loop lacks any control flow instructions that would probably lead to thread divergence.

In our GPU implementation, the deterministic finite automaton (DFA) of the state machine is stored as a two-dimensional array. The dimensions of the array are equal to the number of states and the size of the alphabet (256 in our case), respectively, and each cell consists of four bytes. The first two bytes contain the next state to move, while the other two contain an indication whether the state is a final state or not. In case the state is final, the corresponding cell will contain the unique identification number (*ID*) of the matching pattern, otherwise zero. A drawback of this structure is that state machine tables will be sparsely populated, containing a significant number of zero elements and only a few non-zero elements. However, the use of more efficient storage structures, like those proposed in [22], are much more complex to map in the memory space of a GPU.

During the initialization phase, the state machine table of each rule group is constructed in host memory by the CPU, and is then copied to texture memory that is accessible directly from the GPU. At the searching phase, all state machine tables reside only in GPU memory. The use of GPU texture memory instead of generic GPU memory has the benefit that memory fetches are cached. A cache hit consumes only one cycle, instead of several hundreds in case of transfers from generic device memory. Since the Aho-Corasick algorithm exhibits strong locality of references [12], the use of texture memory for storing the state machine tables boosts GPU execution time about 19%.

We have implemented two different parallelization methods for the Aho-Corasick searching phase. In the first, each packet is splitted into fixed equal parts and each thread searches each portion of the packet in parallel. In the second, each thread is assigned a whole packet to search in parallel. Both techniques have advantages and disadvantages that will be discussed in Section 4.

3.3 Transferring the Results to the CPU

Every time a thread matches a pattern inside a packet, it reports it by appending it in an array that has been previously allocated in the device memory. The reports for each packet will be written in a separate row of the array, following the order they were copied to the texture memory. That means that the array will have the same number of rows as the number of packets contained in the batch. Each report is constituted by the *ID* of the pattern that was matched and the index inside the packet where it was found.

After the pattern matching execution has finished, the array that contains the matching pairs is copied to the host memory. Before raising an alert for each matching pair, the following extra cases should be examined in case they apply:

– *Case-sensitive patterns.* Since Aho-Corasick cannot distinguish between capital and low letters, an extra, case-sensitive, search should be made at the index where the pattern was found.
– *Offset-oriented rules.* Some patterns must be located in specific locations inside the payload of the packet, in order for the rule to be activated. For example, it is possible to look for a specified pattern within the first 5 bytes of the payload. Such ranges are specified in Snort with special keywords, like `offset`, `depth`, `distance`, etc. The index where the match was found is compared against the offset to argue if the match is valid or not.
– *Patterns with common suffix.* It is possible that if two patterns have the same suffix will also share the same final state in the state machine. Thus, for each pattern, we keep an extra list that contains the “suffix-related” *ID*s in the structure that holds its attributes. If this list is not empty for a matching pattern, the patterns that contained in the list have to be verified to find the actual matching pattern.

4 Implementation

We have implemented Gnort on the GeForce 8 Series architecture using CUDA. NVIDIA states that programs developed for the GeForce 8 series will also work without modification on all future NVIDIA video cards.

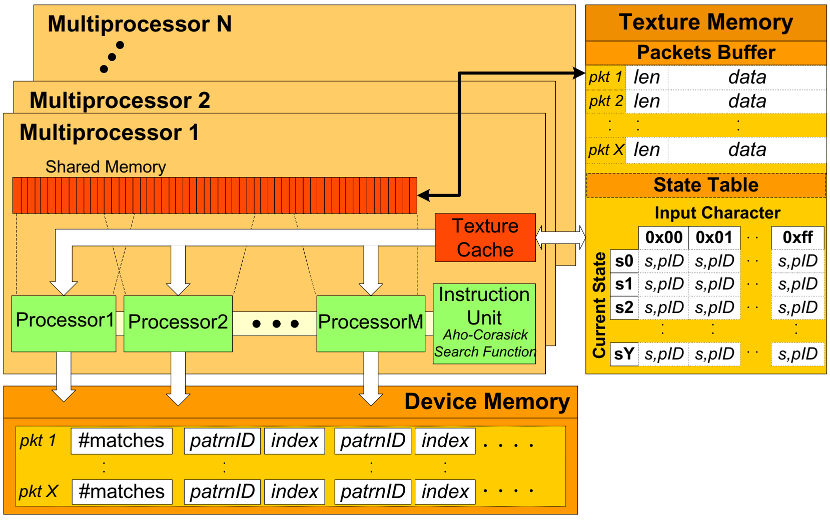
To facilitate concurrent execution between the host and the device, we associate GPU execution into streams. A *stream* is a sequence of operations that execute in order. It is created by the host and in our case includes the copying of the packets to the device memory, the kernel launch, and the transfer of the results back to the host memory. While the stream is executing, the CPU is able to collect the next batch of packets. The CPU work includes the execution flow of Snort to capture, decode, and classify the incoming packets, as well as the extra packet copies to the page-locked memory buffer that we have introduced.

The page-locked memory buffers that are used to collect the packets in batches are allocated by the CUDA runtime driver. The driver tracks the relevant virtual memory ranges and automatically accelerates calls to functions that are used to copy data to the device. The copying of the buffers to the device is asynchronous and is associated to the stream. The device memory where the packets are copied is bound to a texture reference of type `unsigned char` and dimensionality 2. Texture fetches are cached using a proprietary 2D caching scheme and cost only one clock cycle when a cache hit occurs; otherwise a fetch can take 400 to 600 clock cycles. The cache size for texture fetches is 8 KB per multiprocessor. Only the packet payloads are copied to the device, and each payload is stored in a separate row of fixed size. The actual length of the payload is stored in the first two bytes of the row.

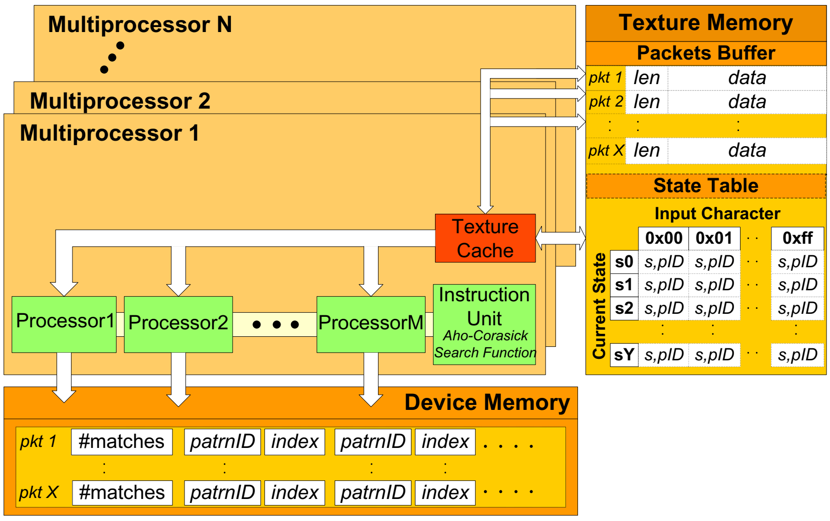
The state machine tables that are used for each group of rules are stored in a texture reference of type `unsigned short` and dimensionality 2. CUDA does not support dynamic allocation of textures yet. To overcome this limitation, all state table arrays are copied to the device at start-up and each of them is dynamically bound to the texture reference, every time a batch of packets have to be matched against.

Once the packets have been copied to the texture bound array, the kernel is initiated by the host to perform the pattern matching. The 8-Series (G8X)—as well as the 9-Series (G9X) which was recently released—contain many independent multiprocessors, each comprising eight processors that run on a SIMD fashion. However, every multiprocessor has an independent instruction decoder, so they can run different instructions.

The Aho-Corasick algorithm performs multi-pattern search, which means that all patterns of a group are searched concurrently. We have explored two different approaches for parallelizing the searching phase by splitting the computation in two ways: assigning a single packet to each multiprocessor at a time, and assigning a single packet to each stream processor at a time. The two approaches are illustrated in Figure 2.



(a) Packet per multiprocessor



(b) Packet per stream processor

Fig. 2. Different pattern matching parallelization approaches. In (a), a different packet is processed by each multiprocessor. All stream processors in the multiprocessor search the packet payload concurrently. In (b), a different packet is processed by each stream processor independently of the others.

4.1 Assigning a Single Packet to each Multiprocessor

In this approach, each packet is processed by a specific thread block, executed by one multiprocessor. The number of threads in the thread block that searchthe packet payload is fixed and equal to the warp size (currently 32). Even though each multiprocessor consists of eight stream processors, the warp size ensures that the multiprocessor's computational resources are maximized by hiding arithmetic pipeline and memory delays.

Each thread searches a different part of the packet, and thus the packet is divided in 32 equal chunks. The 32 chunks of the packet are processed by the 32 threads of the wrap in parallel. To correctly handle matching patterns that span consecutive chunks of the packet, each thread searches in additional X bytes after the chunk it was assigned to search, where X is the maximum pattern length in the state table. To reduce further communication costs due to the overlapping computations, each packet is also copied to the shared memory of the multiprocessor (besides the texture memory)—all threads copy a different chunk in parallel, so this additional copy does not add significant overhead.

An advantage of this method is that all threads are assigned the same amount of work, so execution does not diverge, which would hinder the SIMD execution. Moreover, the texture cache is entirely used for the state tables, as shown in Figure 2(a). A drawback of this approach is that extra processing is needed for the chunk overlaps, especially in case of small packets.

4.2 Assigning a Single Packet to each Stream Processor

In this approach, each packet is processed by a different thread. The number of blocks that are created is equal to the number of multiprocessors the GPU has, so all are working. Each thread block processes X/N packets using an equal number of threads, where X is the number of packets in the batch sent to the GPU, and N is the number of multiprocessors. However, the maximum number of threads that can be created per block is currently 512. So if the number of threads per thread block is greater, more thread blocks are created to keep the number of threads under this limit. The disadvantage of this method is that the amount of work per thread will not be the same since packet sizes will vary. This means that threads of a warp will have to wait until all have finished searching the packet that was assigned to them. However, no additional computation will occur since every packet will be processed in isolation.

Whenever a match occurs, regardless of the implementation used, the corresponding *ID* of the pattern and the index where the match was found are stored in an array allocated in device memory. Each row of the array contain the matches that were found per packet. We use the first position of each row as a counter to know where to put the next match. Every time a match occurs, the corresponding thread increments the counter and writes the report where the counter points to. The increment is performed using an atomic function supplied by CUDA, to overcome possible race conditions for the first parallelization method.

5 Evaluation

In this section, we explore the performance of our implementation. First, we measure the scalability of the various algorithms for different number of patterns and packet sizes, and how they affect overall performance. We then examine how these algorithms perform in a realistic scenario as a function of the traffic load.

In our experiments we used an NVIDIA GeForce 8600GT card, which contains 32 stream processors organized in 4 multiprocessors, operating at 1.2GHz with 512 MB of memory. The CPU in our system was a 3.40 GHz Intel Pentium 4 processor with 2 GB of memory.

In order to directly compare with prior work, we re-implemented the KMP algorithm on the NVIDIA G80 GPU architecture using CUDA. In our implementation, the patterns to be searched, and the partial-match tables that KMP uses, are stored in two 2D texture arrays. Each packet is assigned to a different thread block, while each thread in a block is responsible for searching a specific pattern in the entire packet. This way, each warp of threads performs pattern matching against each packet in parallel, as long as the number of patterns is equal with the number of processors. If the number of patterns is greater than 512, the pattern matching is bundled in groups of 512 patterns each time, due to the limitation of the 512 threads that can be created per block.

We also did a GPU implementation of the Boyer-Moore algorithm, which performs better than KMP. The patterns to be searched, as well as the bad-character shift tables, are stored in two 2D texture arrays similarly to the KMP implementation. Each packet is assigned to a different thread block, while each thread in a block is responsible for searching a specific pattern in the whole packet.

For all experiments conducted, we disregard the time spent in the initialization phase of Snort as well as the logging of the alerts to the disk. Even though it only takes less than just a few seconds to load the patterns and build its internal structures in all cases, there is no practical need to include this time in our graphs. For all experiments we measure the performance of the default Snort using the full Aho-Corasick implementation. We conducted experiments with other implementations as well, however they performed worse in every case. Some information on the different implementations of Aho-Corasick that Snort uses can be found in [30].

5.1 Microbenchmarks

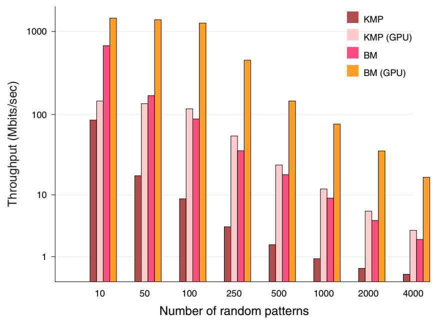
We start by investigating the effect that the size of the batch of packets that are transferred to the GPU has on the overall system performance. We used a synthetic payload trace that contains 1344330 UDP packets with random payload, each 800 bytes in length. The detection engine was disabled, so no execution would take place on the GPU. This way, we measured the time needed for the packets to transferred to the device in batches using the double buffer technique described in Section 3. The times include the capture, decode and classification phases performed by Snort as well as the copying of each packet to our buffer. Table 2 shows the time needed for a packet to copied to the memory of the device for various buffer sizes. We can see that the cost per packet increased as the size of the buffer decreased. For bigger sizes the cost remained somewhat constant. This may be due to the PCI startup overhead of each transaction. As

Table 2. Transfer times per packet as a function of the buffer size for 800-bytes packets

Buffer size (# packets)	Transfer time (ms)
4	0.035547
32	0.008218
512	0.004626
1024	0.004472
4096	0.004326
32768	0.004296

the size of the buffer increases, the number of transaction decreases, resulting in lower startup overheads. For all subsequent experiments we used a buffer of 1024 packets size, which we think is optimal considering the available memory of the host computer we used for the evaluation.

In the next experiment we evaluated how each detection algorithm scales with the number of patterns. We created Snort rules of randomly generated patterns which size varied between 5 and 25 bytes and gave as input to Snort a payload trace that contains UDP packets with random payload, each of 800 bytes in length. All rules are matched against every packet. This is the worst case scenario for a pattern matching engine, as in most cases each packet has to be checked only against a few hundred rules. Figures 3 and 4 show the maximum throughput achieved for single- and multi-pattern matching algorithms respectively, to perform string searches through rule-sets of sizes 10 up to 4000 rules. As shown in Figure 3, single pattern algorithms do not scale as the rule-set size increases. Performance of the CPU implementations of both KMP and BM decreases linearly with the number of patterns. KMP achieves nearly 100 Mbit/s for 10 patterns but its performance for 4000 patterns drops under 1 Mbit/s. BM presents better results but still for a large number of patterns it can only achieve up to 5 Mbit/s. The GPU implementation of these algorithms boosts their performance by up to an order of magnitude. For the case of 50, 100 and 250



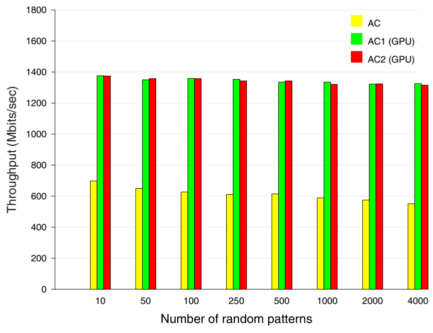


Fig. 3. Throughput sustained for single-pattern matching algorithms

Fig. 4. Throughput sustained for multiple-pattern matching algorithms

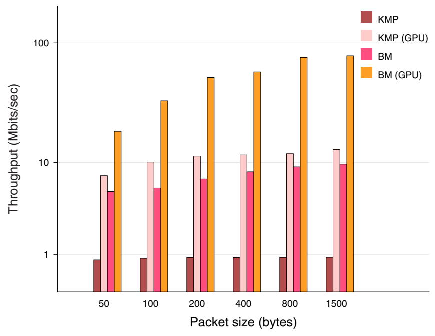


Fig. 5. Throughput sustained for single-pattern matching algorithms

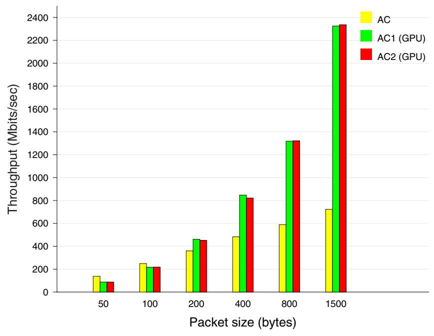


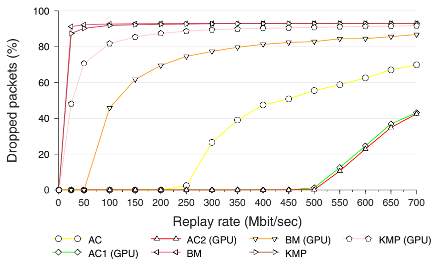
Fig. 6. Throughput sustained for multiple-pattern matching algorithms

patterns we can see that GPU versions of algorithms are an order of magnitude faster than the CPU ones, while for the case of 4000 patterns the improvements reaches a factor of 3.

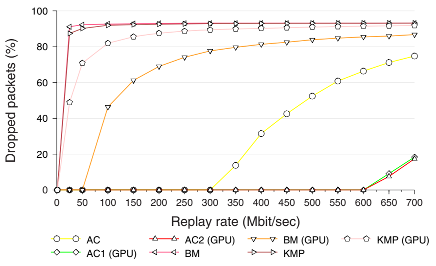
An interesting observation is that the throughput of the GPU implementations for both KMP and BM remained constant for up to 100 patterns. Even though there are 32 processors available, the thread scheduler can pipeline threads execution to effectively utilizes available resources. To verify it, we changed the kernels to return immediately performing a null computation and we observed the same behavior. Performance of the system remained constant for up to 100 patterns and then began decreasing linearly.

In the case of Aho-Corasick algorithm, the throughput remains constant independently of the number of patterns, a behavior expected for a multi-pattern approach. The results are shown in Figure 4. For the CPU implementation, Aho-Corasick achieves nearly 600 Mbit/s throughput, while the GPU implementation reaches up to 1.4 Gbit/s, yielding a 2.4 times improvement. Our two different approaches for implementing Aho-Corasick (displayed as AC1 and AC2 in the graph) do not present significant differences in performance.

Figures 5 and 6 show the throughput achieved for various UDP packet sizes. Snort was loaded with 1000 random patterns which size varied between 5 and 25 bytes. Each packet contains random data, a property that favors the BM algorithm as it will skip most of the payload. CPU implementations of KMP and BM presented a stable performance of around 1 and 10 Mbit/s respectively, independently of the packet size. Their GPU implementations yield a speedup from 2 up to 10 times. The throughput of Aho-Corasick reached over 2.3 Gbit/s for 1500-byte packets, giving a total speed-up of 3.2 compared to the respective CPU implementation. It is important to notice that it is worthless to process small packets on GPU. As it can be seen in Figure 6, for small packet sizes (under 100), the CPU implementation performs better than the GPU. However, for sizes larger than 100 bytes, the GPU implementation outperforms the CPU one in all cases.

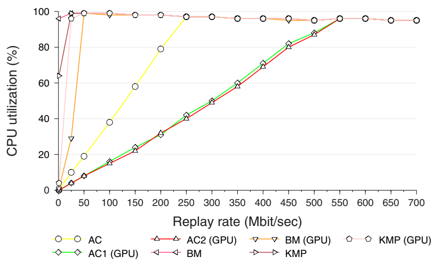


(a) simple pcap

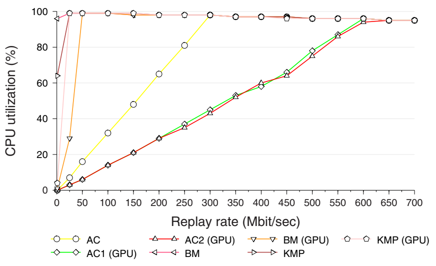


(b) pcap-mmap

Fig. 7. Packet loss ratio as a function of the traffic speed



(a) simple pcap



(b) pcap-mmap

Fig. 8. CPU utilization as a function of the traffic speed

5.2 Macrobenchmarks

In this section we present the evaluation of our prototype implementation using real rules from the current Snort rule set on real network traffic. Our experimental environment consists of two PCs connected via a 1 Gbit/s Ethernet switch. The first PC is equipped with a NVIDIA GeForce 8600GT card and runs our modified version of Snort, while the second is used for replaying real network traffic traces using tcpreplay [32]. We used a full payload trace captured at the access link that connects an educational network with thousands of hosts to the Internet. By rewriting the source and destination MAC addresses in all packets, the generated traffic can be sent to the first PC.

We ran Snort with a custom configuration in which preprocessors and regular expression pattern matching were disabled, as both processes are executed only on the CPU. Snort loaded 5467 rules that contain about 7878 content patterns.

Figure 7 shows the packet loss ratio while replaying the trace at different speeds for two versions of pcap: the default one [21] and the pcap-mmap [35]. The pcap-mmap is a modified version of libpcap that implements a shared memory ring buffer to store captured packets. In this fashion user-space applications are able to read them directly, without trapping to kernel mode and copying them to a user buffer. The use of pcap-mmap gave both unmodified Snort and our system an increase of 50 to 100 Mbit/s to the overall performance. We can

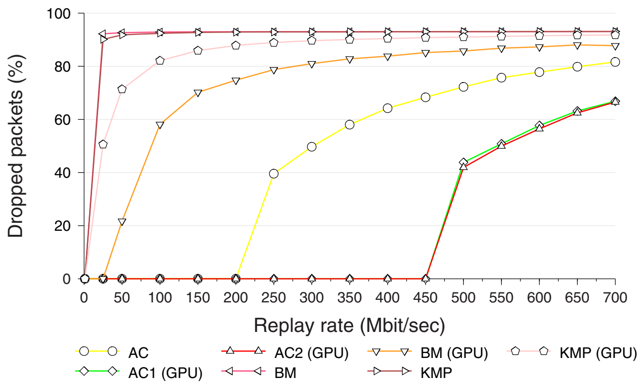


Fig. 9. Packet loss ratio as a function of the traffic speed under heavy CPU load

see that conventional Snort cannot process all packets in rates higher than 300 Mbit/s, so a significant percentage of packets is being lost. On the other hand, our GPU-assisted Snort is twice as fast as the original one. Packet loss for our approach starts at 600 Mbit/s, a 200% improvement to the processing capacity of Snort. The two different GPU implementations of the Aho-Corasick algorithm achieve almost the same performance. For completeness, in Figure 8, we plot the corresponding CPU utilization. Packet loss starts when CPU reaches 100% utilization.

Figure 9 plots the packets dropped by the kernel when CPU was overloaded synthetically. We used a simple program in an infinite tight loop, performing basic math operations to increase CPU usage to 100%. Snort was executing simultaneously. We observe that the performance decreased even when the matching process was executing on GPU. This can be explained by the fact that as the CPU controls the execution of the GPU, by overloading the former the execution flow is affected directly. However, performance degradation did not converge to that of default Snort, in contrast with [17].

6 Conclusions

In this paper, we presented Gnort, an intrusion detection system that utilizes the GPU to offload pattern matching computation. We ported the classic Aho-Corasick algorithm to run on the GPU exploiting the SIMD instructions. Our prototype system was able to achieve a maximum throughput of 2.3 Gbit/s, while in a real world scenario outperformed conventional Snort by a factor of two.

As future work we plan on eliminating the extra copy we introduced in order to transfer the packets to the GPU in batches. One way to accomplish this, is to transfer the packets directly from the kernel buffer. This would require that the buffer will be allocated from the application and will be shared between the user and kernel spaces. We believe that by modifying the pcap-mmap, that already implements this shared buffer capability, we can benefit from the lack of copies of both from kernel to user space as well as the one to our defined buffer. An even more efficient way would be to DMA directly the packets from the NIC to the GPU, without occupying the CPU at all. Currently, this is not supported but it may be in the future.

Finally, we plan on utilizing multiple GPUs instead of a single one. Modern motherboards support dual GPUs, and there are PCI Express backplanes that support multiple slots. We believe that building such *"clusters"* of GPUs will be able to support multiple Gigabit per second Intrusion Detection Systems.

Acknowledgments

This work was supported in part by the project CyberScope, funded by the Greek Secretariat for Research and Technology under contract number PENED 03ED440 and by the Marie Curie Actions - Reintegration Grants project PASS. G. Vasiliadis, S. Antonatos, M. Polychronakis, E. P. Markatos and S. Ioannidis are also with the University of Crete.

References

1. Aho, A.V., Corasick, M.J.: Efficient string matching: an aid to bibliographic search. Communications of the ACM 18(6), 333–340 (1975)
2. Antonatos, S., Anagnostakis, K., Markatos, E.: Generating realistic workloads for network intrusion detection systems. In: Proceedings of the 4th ACM Workshop on Software and Performance (January 2004)
3. Attig, M., Lockwood, J.: A framework for rule processing in reconfigurable network systems. In: Proceedings of the 13th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM 2005), Washington, DC, USA, 2005, pp. 225–234. IEEE Computer Society Press, Los Alamitos (2005)
4. Baker, Z.K., Prasanna, V.K.: Time and area efficient pattern matching on FPGAs. In: Proceedings of the 2004 ACM/SIGDA 12th International Symposium on Field Programmable Gate Arrays (FPGA 2004), pp. 223–232. ACM, New York (2004)
5. Bos, H., Huang, K.: Towards software-based signature detection for intrusion prevention on the network card. In: Valdes, A., Zamboni, D. (eds.) RAID 2005. LNCS, vol. 3858, pp. 102–123. Springer, Heidelberg (2006)
6. Boyer, R.S., Moore, J.S.: A fast string searching algorithm. Communications of the Association for Computing Machinery 20(10), 762–772 (1977)
7. Cabrera, J.B.D., Gosar, J., Lee, W., Mehra, R.K.: On the statistical distribution of processing times in network intrusion detection. In: 43rd IEEE Conference on Decision and Control, December 2004, pp. 75–80 (2004)
8. Clark, C., Lee, W., Schimmel, D., Contis, D., Kone, M., Thomas, A.: A hardware platform for network intrusion detection and prevention. In: Proceedings of the 3rd Workshop on Network Processors and Applications (NP3) (2004)
9. Coit, C., Staniford, S., McAlerney, J.: Towards faster string matching for intrusion detection or exceeding the speed of Snort. In: Proceedings of DARPA Information Survivability Conference & Exposition II (DISCEX 2001) (June 2001)
10. Commentz-Walter, B.: A string matching algorithm fast on the average. In: Proceedings of the 6th International Colloquium on Automata, Languages and Programming, pp. 118–131.

11. Cook, D.L., Ioannidis, J., Keromytis, A.D., Luck, J.: Cryptographics: Secret key cryptography using graphics cards. In: Proceedings of RSA Conference, Cryptographer's Track (CT-RSA), pp. 334–350 (2005)
12. de Bruijn, W., Slowinska, A., van Reeuwijk, K., Hruby, T., Xu, L., Bos, H.: SafeCard: a Gigabit IPS on the network card. In: Zamboni, D., Krügel, C. (eds.) RAID 2006. LNCS, vol. 4219, pp. 311–330. Springer, Heidelberg (2006)
13. Dharmapurikar, S., Krishnamurthy, P., Sproull, T.S., Lockwood, J.W.: Deep packet inspection using parallel bloom filters. IEEE Micro 24(1), 52–61 (2004)
14. Dharmapurikar, S., Lockwood, J.: Fast and scalable pattern matching for content filtering. In: Proceedings of the 2005 ACM symposium on Architecture for networking and communications systems (ANCS 2005), pp. 183–192. ACM, New York (2005)
15. Fisk, M., Varghese, G.: Applying fast string matching to intrusion detection. Technical Repor In preparation, successor to UCSD TR CS2001-0670, University of California, San Diego (2002)
16. C. IOS. IPS deployment guide, `http://www.cisco.com`
17. Jacob, N., Brodley, C.: Offloading IDS computation to the GPU. In: Security Applications Conference on Annual Computer Security Applications Conference (ACSAC 2006), Washington, DC, USA, pp. 371–380. IEEE Computer Society, Los Alamitos (2006)
18. Knuth, D.E., Morris, J., Pratt, V.: Fast pattern matching in strings. SIAM Journal on Computing 6(2), 127–146 (1977)
19. Kruegel, C., Valeur, F., Vigna, G., Kemmerer, R.: Stateful intrusion detection for high-speed networks. In: Proceedings of the IEEE Symposium on Security and Privacy, May 2002, pp. 285–294 (2002)
20. Lodovico Marziale, G.G.R.I., Roussev, V.: Massive threading: Using GPUs to increase the performance of digital forensics tools. Digital Investigation 1, 73–81 (2007)
21. McCanne, S., Leres, C., Jacobson, V.: libpcap. Lawrence Berkeley Laboratory, Berkeley, `http://www.tcpdump.org/`
22. Norton, M.: Optimizing pattern matching for intrusion detection (July 2004), `http://docs.idsresearch.org/OptimizingPatternMatchingForIDS.pdf`
23. NVIDIA. NVIDIA CUDA Compute Unified Device Architecture Programming Guide, version 1.1, `http://developer.download.nvidia.com/compute/cuda/1_1/NVIDIA_CUDA_Programming_Guide_1.1.pdf`
24. Paxson, V.: Bro: A system for detecting network intruders in real-time. In: Proceedings of the 7th conference on USENIX Security Symposium (SSYM 1998), Berkeley, CA, USA, p. 3. USENIX Association (1998)
25. Paxson, V., Sommer, R., Weaver, N.: An architecture for exploiting multi-core processors to parallelize network intrusion prevention. In: Proceedings of the IEEE Sarnoff Symposium (May 2007)
26. Roesch, M.: Snort: Lightweight intrusion detection for networks. In: Proceedings of the 1999 USENIX LISA Systems Administration Conference (November 1999)
27. Schaelicke, L., Wheeler, K., Freeland, C.: SPANIDS: a scalable network intrusion detection loadbalancer. In: CF 2005: Proceedings of the 2nd conference on Computing frontiers, pp. 315–322. ACM, New York (2005)
28. Sidhu, R., Prasanna, V.: Fast regular expression matching using FPGAs. In: IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM 2001) (2001)

29. Tan, L., Brotherton, B., Sherwood, T.: Bit-split string-matching engines for intrusion detection and prevention. ACM Transactions on Architecture and Code Optimization 3(1), 3–34 (2006)
30. The Snort Project. Snort users manual 2.8.0, http://www.snort.org/docs/snort_manual/2.8.0/snort_manual.pdf
31. Tuck, N., Sherwood, T., Calder, B., Varghese, G.: Deterministic memory-efficient string matching algorithms for intrusion detection. In: Proceedings of the IEEE Infocom Conference, pp. 333–340 (2004)
32. Turner, A.: Tcpreplay, http://tcpreplay.synfin.net/trac/
33. 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)
34. Watanabe, K., Tsuruoka, N., Himeno, R.: Performance of network intrusion detection cluster system. In: Proceedings of The 5th International Symposium on High Performance Computing (ISHPC-V) (2003)
35. Wood, P.: libpcap-mmap, http://public.lanl.gov/cpw/
36. Wu, S., Manber, U.: A fast algorithm for multi-pattern searching. Technical Report TR-94-17 (1994)
37. Yu, F., Katz, R.H., Lakshman, T.V.: Gigabit Rate Packet Pattern-Matching Using TCAM. In: Proceedings of the 12th IEEE International Conference on Network Protocols (ICNP 2004), Washington, DC, USA, October 2004, pp. 174–183. IEEE Computer Society, Los Alamitos (2004)
38. Yusuf, S., Luk, W.: Bitwise optimised CAM for network intrusion detection systems. In: Proceedings of International Conference on Field Programmable Logic and Applications, pp. 444–449 (2005)

Predicting the Resource Consumption of Network Intrusion Detection Systems

Holger Dreger[1], Anja Feldmann[2], Vern Paxson[3,4], and Robin Sommer[4,5]

[1] Siemens AG, Corporate Technology
[2] Deutsche Telekom Labs / TU Berlin
[3] UC Berkeley
[4] International Computer Science Institute
[5] Lawrence Berkeley National Laboratory

Abstract. When installing network intrusion detection systems (NIDSs), operators are faced with a large number of parameters and analysis options for tuning trade-offs between detection accuracy versus resource requirements. In this work we set out to assist this process by understanding and predicting the CPU and memory consumption of such systems. We begin towards this goal by devising a general NIDS resource model to capture the ways in which CPU and memory usage scale with changes in network traffic. We then use this model to predict the resource demands of different configurations in specific environments. Finally, we present an approach to derive site-specific NIDS configurations that maximize the depth of analysis given predefined resource constraints. We validate our approach by applying it to the open-source Bro NIDS, testing the methodology using real network data, and developing a corresponding tool, `nidsconf`, that automatically derives a set of configurations suitable for a given environment based on a *sample* of the site's traffic. While no automatically generated configuration can ever be optimal, these configurations provide sound starting points, with promise to significantly reduce the traditional trial-and-error NIDS installation cycle.

1 Introduction

Operators of network intrusion detection systems (NIDSs) face significant challenges in understanding how to best configure and provision their systems. The difficulties arise from the need to understand the relationship between the wide range of analyses and tuning parameters provided by modern NIDSs, and the resources required by different combinations of these. In this context, a particular difficulty regards how resource consumption intimately relates to the specifics of the network's traffic—such as its application mix and its changes over time—as well as the internals of the particular NIDS in consideration. Consequently, in our experience the operational deployment of a NIDS is often a trial-and-error process, for which it can take weeks to converge on an apt, stable configuration.

In this work we set out to assist operators with understanding resource consumption trade-offs when operating a NIDS that provides a large number of tuning options. We begin towards our goal by devising a general NIDS resource model to capture the ways in which CPU and memory usage scale with changes in network traffic. We then use this model to predict the resource demands of different configurations for specific environments. Finally, we present an approach to derive site-specific NIDS configurations that maximize the depth of analysis given predefined resource constraints.

A NIDS must operate in a *soft real-time* manner, in order to issue timely alerts and perhaps blocking directives for intrusion prevention. Such operation differs from *hard* real-time in that the consequences of the NIDS failing to "keep up" with the rate of arriving traffic is not catastrophe, but rather *degraded performance* in terms of some traffic escaping analysis ("drops") or experiencing slower throughput (for intrusion prevention systems that forward traffic only after the NIDS has inspected it). Soft real-time operation has two significant implications in terms of predicting the resource consumption of NIDSs. First, because NIDSs do not operate in hard real-time, we seek to avoid performance evaluation techniques that aim to prove compliance of the system with rigorous deadlines (e.g., assuring that it spends no more than T microseconds on any given packet). Given the very wide range of per-packet analysis cost in a modern NIDS (as we discuss later in this paper), such techniques would severely reduce our estimate of the performance a NIDS can provide in an operational context. Second, soft real-time operation means that we also cannot rely upon techniques that predict a system's performance solely in terms of aggregate CPU and memory consumption, because we must also pay attention to *instantaneous* CPU load, in order to understand the degree to which in a given environment the system would experience degraded performance (packet drops or slower forwarding).

When modeling the resource consumption of a NIDS, our main hypothesis concerns *orthogonal decomposition*: i.e., the major subcomponents of a NIDS are sufficiently independent that we can analyze them in isolation and then extrapolate aggregate behavior as the composition of their individual contributions. In a different dimension, we explore how the systems' overall resource requirements correlate to the volume and the mix of network traffic. If orthogonal decomposition holds, then we can systematically analyze a NIDS' resource consumption by capturing the performance of each subcomponent individually, and then estimating the aggregate resource requirements as the sum of the individual requirements. We partition our analysis along two axes: type of analysis, and proportion of connections within each class of traffic. We find that the demands of many components scale directly with the prevalence of a given class of connections within the aggregate traffic stream. This observation allows us to accurately estimate resource consumption by characterizing a site's traffic "mix." Since such mixes change over time, however, it is crucial to consider both short-term and long-term fluctuations.

We stress that, by design, our model does *not* incorporate a notion of detection quality, as that cannot reasonably be predicted from past traffic as resource usage can. We focus on identifying the types of analyses which are *feasible* under given resource constraints. With this information the operator can assess which option promises the largest gain for the site in terms of operational benefit, considering the site's security policy and threat model.

We validate our approach by applying it to Bro, a well-known, open-source NIDS [7]. Using this system, we verify the validity of our model using real network data, and develop a corresponding prototype tool, `nidsconf`, to derive a set of configurations suitable for a given environment. The NIDS operator can then examine these configurations and select one that best fits with the site's security needs. Given a relatively small *sample* of a site's traffic, `nidsconf` performs systematic measurements on it, extrapolates a set of possible NIDS configurations and estimates their performance and resource implications. In a second stage the tool is also provided with a longer-term connection-level log file (such as produced by NetFlow). Given this and the results from the systematic measurements, the tool can project resource demands of the NIDS' subcomponents without actually running the NIDS on long periods of traffic. Thus the tool can be used not only to derive possible NIDS configurations but also to estimate when, for a given configuration and a given estimation of traffic growth, the resources of the machine running the NIDS will no longer suffice. While we do not claim that `nidsconf` always produces optimal configurations, we argue that it provides a sound starting point for further fine-tuning.

We structure the remainder of this paper as follows. In §2 we use an example to demonstrate the impact of resource exhaustion. In §3 we introduce our approach, and validate its underlying premises in §4 by using it to predict the resource usage of the Bro NIDS. In §5 we present our methodology for predicting the resource consumption of a NIDS for a specific target environment, including the automatic derivation of suitable configurations. We discuss related work in §6 and conclude in §7.

2 Impact of Resource Exhaustion

We begin with an examination of how resource exhaustion affects the quality of network security monitoring, since this goes to the heart of the problem of understanding the onset and significance of degraded NIDS performance. We do so in the context of the behavior of the open-source Bro NIDS [7] when it runs out of available CPU cycles or memory.

CPU Overload. The primary consequence of CPU overload are packet drops, and thus potentially undetected attacks. As sketched above, a NIDS is a *soft real-time* system: it can buffer packets for a certain (small) amount of time, which enables it to tolerate sporadic processing spikes as long as traffic arriving in the interim fits within the buffer. On average, however, processing needs to keep up with the input stream to avoid chronic overload and therefore packets drops. To understand the correlation between packet drops and CPU load, we run the Bro NIDS live on a high-volume network link (see §4) using a configuration that deliberately overloads the host CPU in single peaks. We then correlate the system's CPU usage with the observed packet drops.

Figure 1 shows the real-time (Y-axis) that elapses while Bro processes each second of network traffic (X-axis). The vertical lines denote times at which the packet capture facility (libpcap) reports drops; the corresponding CPU samples are shown with a *filled* circle.

The NIDS can avoid drops as long as the number of processing outliers remains small—more precisely, as long as they can be compensated by buffering of captured packets. For example, the 20MB buffer used in our evaluations enabled us to process an extreme outlier—requiring 2.5 s for one real-time second worth of network traffic—without packet drops. Accordingly, we find that the first packet drop occurs only after a spike in processing real time of more than 4s. Closer inspection shows that the loss

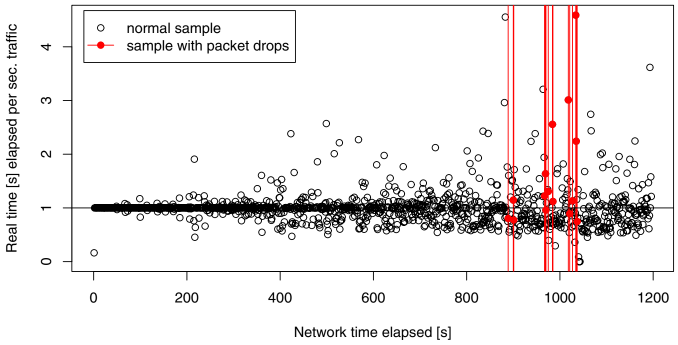


Fig. 1. Relation between elapsed real-time and packet drops

does not occur immediately during processing the "expensive" traffic but rather six network seconds later. It is only at that point that the buffer is completely full and the *lag* (i.e., how far the NIDS is behind in its processing) exceeds 5.5s. Such a large amount of buffering thus makes it difficult to predict the occurrence of drops and their likely magnitude: *(i)* the buffer can generally absorb single outliers, and *(ii)* the buffer capacity (in seconds) depends on the traffic volume yet to come. But clearly we desire to keep the lag small.

Memory Exhaustion. When a stateful NIDS completely consumes the memory available to it, it can no longer effectively operate, as it cannot store additional state. It can, however, try to reclaim memory by expiring existing state. The challenges here are *(i)* how to recognize that an exhaustion condition is approaching prior to its actual onset, *(ii)* in the face of often complex internal data structures [3], and then *(iii)* locating apt state to expire that minimizes the ability for attackers to leverage the expiration for evading detection.

One simple approach for limiting memory consumption imposes a limit on the size of each internal data structure. Snort [8], for example, allows the user to specify a maximum number of concurrent connections for its TCP preprocessor. If this limit is reached, Snort randomly picks some connections and flushes their state to free up memory. Similarly, Snort addresses the issue of variable stream reassembly size by providing an option to limit the total number of bytes in the reassembler. Bro on the other hand does not provide a mechanism to limit the size of data structures to a fixed size; its state management instead relies on timeouts, which can be set on a per-data structure basis, and with respect to when state was first created, or last read or updated. However, these do not provide a guarantee that Bro can avoid memory exhaustion, and thus it can crash in the worst case. Bro does however include extensive internal memory instrumentation [3] to understand its consumption, which we leverage for our measurements.

Memory consumption and processing lag can become coupled in two different ways. First, large data structures can take increasingly longer to search as they grow in size, increasing the processing burden. Second, in systems that provide more virtual memory than physical memory, consuming the entire physical memory does not crash the system but instead degrades its performance due to increased paging activity. In the worst case, such systems can thrash, which can enormously diminish real-time performance.

3 Modeling NIDS Resource Usage

In this section we consider the high-level components that determine the resource usage of a NIDS. We first discuss the rationale that leads to our framing of the components, and then sketch our resulting distillation. The next section proceeds to evaluate the framework against the Bro NIDS.

3.1 The Structure of NIDS Processing

Fundamental to a NIDS's operation is tracking communication between multiple network endpoints. All major NIDS's today operate in a *stateful* fashion, decoding network communication according to the protocols used, and to a degree mirroring the state maintained by the communication endpoints. This state naturally grows proportional to the number of active connections[1], and implementations of stateful approaches are naturally aligned with the network protocol stack. To reliably ascertain the semantics of an application-layer protocol, the system first processes the network and transport layers of the communication. For example, for HTTP the NIDS first parses the IP header (to verify checksums, extract addresses, determine transport protocol, and so on) and the TCP header (update the TCP state machine, checksum the payload), and then reassembles the TCP byte stream, before it can finally parse the HTTP protocol.

A primary characteristic of the network protocol stack is its extensive use of *encapsulation*: individual layers are independent of each other; while their input/output is connected, there ideally is no exchange of state between layers. Accordingly, for a NIDS structured along these lines its protocol-analyzing components can likewise operate independently. In particular, it is plausible to assume that the total resource consumption, in terms of CPU and memory usage, is the sum of the demands of the individual components. This observation forms a basis for our estimation methodology.

In operation, a NIDS's resource usage primarily depends on the characteristics of the network traffic it analyzes; it spends its CPU cycles almost exclusively on analyzing input traffic, and requires memory to store results as it proceeds. In general, network packets provide the only sustained stream of input during operation, and resource usage therefore should directly reflect the volume and content of the analyzed packets.[2]

We now hypothesize that for each component of a NIDS that analyzes a particular facet or layer of network activity—which we term an *analyzer*—the relationship between input traffic and the analyzer's resource demands is linear. Let t_0 be the time when NIDS operation begins, and P_t the number of input packets seen up to time $t \geq t_0$. Furthermore, let C_t be the *total* number of transport-layer connections seen up to time t, and c_t the number of connections *currently active* at time t. Then we argue: *Network-layer* analyzers operate strictly on a per-packet basis, and so should require $O(P_t)$ CPU time, and rarely store state. (One exception concerns reassembly of IP fragments; however, in our experience the memory required for this is

[1] For UDP and ICMP we assume flow-like definitions, similar to how NetFlow abstracts packets.

[2] In this work, we focus on stand-alone NIDSs that analyze traffic and directly report alerts. In more complex setups (e.g., with distributed architectures) resource consumption may depend on other sources of input as well.

negligible even in large networks.) *Transport-layer* analyzers also operate packet-wise. Thus, their amortized CPU usage will scale as $O(P_t)$. However, transport-layer analyzers can require significant memory, such as tracking TCP sequence numbers, connection states, and byte streams. These analyzers therefore will employ data structures to store all currently active connections, requiring $O(max(c_t))$ memory. For stream-based protocols, the transport-layer performs payload reassembly, which requires memory that scales with $O(max(c_t \cdot m_t))$, where m_t represents the largest chunk of unacknowledged data on any active connection at time t (cf. [1]). Finally, *application-layer* analyzers examine the payload data as reconstructed by the transport layer. Thus, their CPU time scales proportional to the number of connections, and depends on how much of the payload the analyzer examines. (For example, an HTTP analyzer might only extract the URL in client requests, and skip analysis of the much larger server reply.) The total size of the connection clearly establishes an upper limit. Accordingly, the state requirements for application analyzers will depend on the application protocol and will be kept on a per-connection basis, so will scale proportional to the protocol *mix* (how prevalent the application is in the traffic stream) and the number of connections c_t.

In addition to protocol analyzers, a NIDS may perform *inter-connection* correlation. For example, a scan detector might count connections per source IP address, or an FTP session analyzer might follow the association between FTP client directives and subsequent data-transfer connections. In general, the resource usage of such analyzers can be harder to predict, as it will depend on specifics of the analysis (e.g., the scan detector above requires $O(C_t)$ CPU and memory if it does not expire any state, while the FTP session analyzer only requires CPU and memory in proportion to the number of FTP client connections). However, in our experience it is rare that such analyzers exceed CPU or memory demands of $O(C_t)$, since such analysis quickly becomes intractable on any high-volume link. Moreover, while it is possible that such inter-connection analyzer may depend on the results of other analyzers, we find that such analyzers tend to be well modular and decoupled (e.g., the scan detector needs the same amount of memory independent of whether the NIDS performs HTTP URL extraction or enables FTP session-tracking).

3.2 Principle Contributors to Resource Usage

Overall, it appears reasonable to assume that for a typical analyzer, resource usage is *(i)* linear with either the number of input packets or the number of connections it processes, and *(ii)* independent of other analyzers. In this light, we can frame two main contributors to the resource usage of a NIDS:

1. The specific analyzers enabled by the operator for the system's analysis. That these contribute to resource usage is obvious, but the key point we want to make is that most NIDSs provide options to enable/disable certain analyzers in order to trade off resource requirements. Yet NIDSs give the operators almost no concrete guidance regarding the trade-offs, so it can be extremely hard to predict the performance of a NIDS when enabling different sets of analyzers. This difficulty motivated us to build our tool `nidsconf` (per §5.2) that provides an understanding of resource usage trade-offs to support configuration decisions.

2. The traffic *mix* of the input stream—i.e., the prevalence of different types of application sessions—as this affects the number of connections examined by each type of analyzers.

The above reasoning need not hold universally. However, we examined the architecture of two popular open source NIDS, Snort and Bro, and found that their resource consumption indeed appears consistent with the model discussed above. We hypothesize that we can characterize most operational NIDSs in this fashion, and thus they will lend themselves well to the kind of performance prediction we outline in §5. To support our claims, we now explore the resource usage of the Bro NIDS in more depth.

4 Example NIDS Resource Usage

To assess our approach of modeling a NIDS's resource demands as the sum of the requirements of its individual analyzers, and scaling linearly with the number of application sessions, we now examine an example NIDS. Among the two predominant open-source NIDSs, Snort and Bro, we chose to examine Bro for two reasons: *(i)* Bro provides a superset of Snort's functionality, since it includes both a signature-matching engine and an application-analysis scripting language; and *(ii)* it provides extensive, fine-grained instrumentation of its internal resource consumption; see [3] for the specifics of how the system measures CPU and memory consumption in real-time. Snort does not provide similar capabilities. For our examination we have to delve into details of the Bro system, and we note that some of the specifics of our modeling are necessarily tied to Bro's implementation. While this is unavoidable, as discussed above we believe that similar results will hold for Snort and other modern NIDSs.

For our analysis we captured a 24-hour *full* trace at the border router of the Münchener Wissenschaftsnetz (MWN). This facility provides 10 Gbps upstream capacity to roughly 50,000 hosts at two major universities, along with additional research institutes, totaling 2-4 TB a day. To avoid packet drops, we captured the trace with a high-performance Endace DAG card. The trace encompasses 3.2 TB of data in 6.3 billion packets and 137 million connections. 76% of all packets are TCP. In the remainder of this paper, we refer to this trace as `MWN-full` .

4.1 Decomposition of Resource Usage

We first assess our hypothesis that we can consider the resource consumption of the NIDS's analyzers as independent of one another. We then check if resource usage generally scales linearly with the number of connections on the monitored network link.

Independence of Analyzer Resource Usage. For our analysis we use Bro version 1.1, focusing on 13 analyzers: *finger*, *frag*, *ftp*, *http-request*, *ident*, *irc*, *login*, *pop3*, *portmapper*, *smtp*, *ssh*, *ssl*, and *tftp*. To keep the analyzed data volume tractable, we use a 20-minute, TCP-only excerpt of `MWN-full`, which we refer to as `Trace-20m`,

We run 15 different experiments. First, we establish a base case (BroBase), which only performs generic connection processing. In this configuration, Bro only analyzes connection control packets, i.e., all packets with any of the TCP flags SYN, FIN or

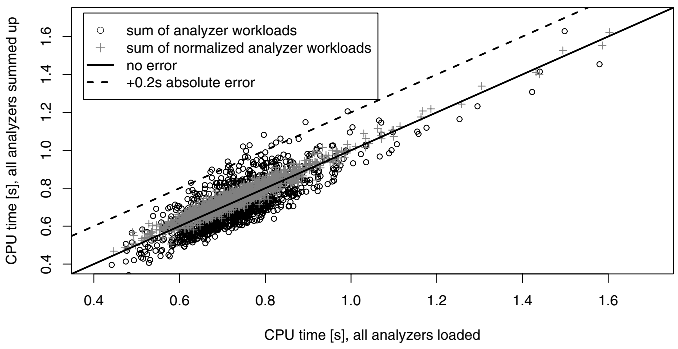


Fig. 2. Scatter plot of accumulated CPU usages vs. measured CPU usage

RST set. This suffices for generating one-line summaries of each TCP connection in the trace. BROBASE thus reflects a minimal level of still-meaningful analysis. Next, we run a fully loaded analysis, BROALL, which enables all analyzers listed above, and by far exceeds the available resources. Finally, we perform 13 additional runs where we enable a single one of the analyzers on top of the BROBASE configuration. For each test, Bro is supplied with a trace prefiltered for the packets the configuration examines. This mimics live operation, where this filtering is usually done in the kernel and therefore not accounted to the Bro process.

We start with examining CPU usage. For each of the 13 runs using BROBASE plus one additional analyzer, we calculate the contribution of the analyzer as the difference in CPU usage between the run and that for the BROBASE configuration. We then form an estimate of the time of the BROALL configuration as the sum of the contributions of the individual analyzers plus the usage of the BROBASE configuration. We term this estimate BROAGG.

Figure 2 shows a scatter plot of the measured CPU times. Each point in the plot corresponds to the CPU time required for one second of network input. The circles reflect BROAGG (Y-axis) versus BROALL (X-axis), with five samples between 1.7s and 2.6s omitted from the plot for legibility. We observe that there is quite some variance in the matching of the samples: The mean relative error is 9.2% (median 8.6%) and for some samples the absolute error of BROAGG's CPU time exceeds 0.2s (20% CPU load). There is also a systematic bias towards slight underestimation by BROAGG, with about 64% of its one-second intervals being somewhat lower than the value measured during that interval for BROALL.

To understand the origin of these differences, we examine the relative contribution of the individual analyzers. We find that there are a number of analyzers that do not add significantly to the workload, primarily due to those that examine connections that are not prevalent in the analyzed network trace (e.g., *finger*). The resource consumption with these analyzers enabled is very close to that for plain BROBASE. Furthermore, due to the imprecision of the operating system's resource accounting, two measurements of the same workload are never exactly the same; in fact, when running the BROBASE configuration ten times, the per-second samples differ by $M_R = 18$ msec on average. This means that if an analyzer contributes very little workload, we cannot soundly distinguish its contribution to CPU usage from simple measurement variation. The fluctuations of all individual runs with just one additional analyzer may well accumulate to the total variation seen in Figure 2.

To compensate for these measurement artifacts, we introduce a normalization of CPU times, as follows. For each single-analyzer configuration, we first calculate the differences of all its CPU samples with respect to the corresponding samples of BroBase. If the mean of these differences is less than the previously measured value of M_R then we instead predict its load based on aggregation across 10-second bins rather than 1-second bins. The '+' symbols in Figure 2 show the result: we both reduce overall fluctuation considerably, and no sample of BroAgg exceeds BroAll by more than 0.2s. The mean relative error drops to 3.5% (median 2.8%), indicating a good match. As in the non-normalized measurements, for most samples (71%) the CPU usage is extrapolated to slightly lower values than in the actual BroAll measurement. The key point is we achieve these gains solely by aggregating the analyzers that introduce very light additional processing. Thus, we conclude that *(i)* these account for the majority of the inaccuracy, *(ii)* correcting them via normalization does not diminish the soundness of the prediction, and *(iii)* otherwise, analyzer CPU times do in fact sum as expected.

Turning to memory usage, we use the same approach for assessing the additivity of the analyzers. We compute the difference in memory allocation between the instance with the additional analyzer enabled versus that of BroBase. As expected, summing these differences and adding the memory consumption of BroBase yields 465 MB, closely matching the memory usage of BroAll (461 MB).

Overall, we conclude that we can indeed consider the resource consumption of the analyzers as independent of one another.

Linear Scaling with Number of Connections. We now assess our second hypothesis: that a NIDS resource consumption scales linearly with the number of processed connections. For this evaluation, we run Bro with identical configurations on traces that differ mainly in the number of connections that they contain at any given time. To construct such traces, we randomly subsample an input trace using per-connection sampling with different sampling factors, run Bro on the subtrace, and compare the resulting resource usage in terms of both CPU and memory. To then extrapolate the resource usage on the full trace, we multiply by the sample factor.

To sample a trace with a sample factor P, we hash the IP addresses and port numbers of each packet into a range $[0; P-1]$ and pick all connections that fall into a particular bucket. We choose a prime for the sample factor to ensure we avoid aliasing; this approach distributes connections across all buckets in a close to uniform fashion as shown in [11]. For our analysis we sampled `Trace-20m` with sampling factors $P = 7$ (resulting in STrace7) and $P = 31$ (resulting in STrace31).

CPU Usage. Figure 3 shows a scatter plot of the CPU times for BroBase on `Trace-20m` without sampling, vs. extrapolating BroBase on STrace7 (circles) and STrace31 (triangles). We notice that in general the extrapolations match well, but are a bit low (the mean is 0.02 sec lower). Unsurprisingly, the fluctuation in the deviation from the originally measured values grows with the sampling factor (further measurements not included in Figure 3 with sampling factors between 7 and 31 confirm this).

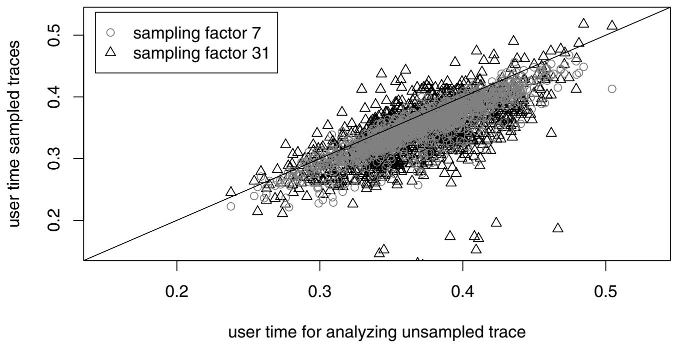


Fig. 3. Scatter plot of BroBase configuration on sampled traces vs. non-sampled trace

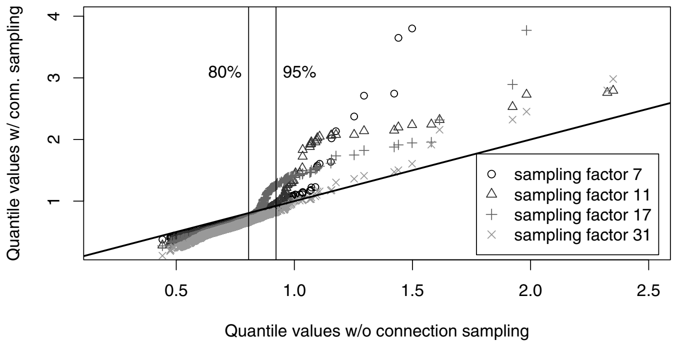


Fig. 4. QQ plot of analyzer workload without sampling vs. with different sampling factors

Naturally, the measured CPU times are very small if only a few connections are analyzed. For example, in the unsampled trace Bro consumes on average 370 msec for one second of network traffic when analyzing all the connections. With a sampling factor of 31, we would expect consumption to drop to $370/31 = 12$ msec, at which point we are at the edge of the OS's accounting precision. In fact, however, we find that extrapolation still works fairly well: for sample factor 31, the median of the extrapolated measurements is only 28 msec lower than the median of the measurements for the full trace. We have verified that similar observations hold for other segments of `MWN-full`, as well as for other traces.

Next we check if this finding still holds for more complex configurations. To this end, the QQ-plot in Figure 4 compares the distribution of CPU times for BroAll (i.e., 13 additional analyzers) on the full `Trace-20m` (X-axis) vs. sub-sampled traces

Table 1. Memory scaling factors: 10 BroBase runs (left) / 10 BroAll runs (right)

Sampling factor	1	7	11	17	31
Memory ratio	1	7.0	11.0	16.6	30.7

Sampling factor	1	7	11	17	31
Memory ratio	1	3.64	4.87	6.30	9.34

(Y-axis, with sample factors of 7, 11, 17, and 31). Overall, the majority of the samples match fairly well, though with a bias for smaller values towards underestimation (left of the 80th percentile line), and with unstable upper quantiles (usually overestimates).

Memory Usage. Turning to memory consumption, for each sampling factor we conducted 10 runs with BroBase on `Trace-20m`, measuring the maximum consumption figures on the sampled traces as reported by the OS.[3] Table 1 (left) shows the ratio between the memory consumption on the entire trace versus that for the sampled traces. Ideally, this figure would match the sampling factor, since then we would extrapolate perfectly from the sample. We see that in general the ratio is close, with a bias towards being a bit low. From this we conclude that for BroBase, predicting memory use from a sampled trace will result in fairly accurate, though sometimes slightly high, estimates.

As discussed in Section 3, we would not expect memory usage of application-layer analyzers to always scale linearly with the number of connections, since some analyzers accumulate state not on a per-connection basis but rather according to some *grouping* of the connections (e.g., Bro's HTTP analyzer groups connections into "sessions"). In such cases the memory estimate we get by scaling with the connection sample factor can be a (potentially significant) overestimation. This effect is visible in Table 1 (right), which shows the same sort of analysis as above but now for BroAll. We see that the extrapolation factors can be off by more than a factor of three. By running each analyzer separately, we identified the culprits: both the HTTP and SSL analyzers associate their state per session, rather than per connection. However, we note that at least the required memory never *exceeds* the prediction, and thus we can use the prediction as a conservative upper bound.

In summary, we find that both CPU and memory usage can generally be predicted well with a model linear in the number of connections. We need to keep in mind however that it can overestimate the memory demand for some analyzers.

5 Resource Prediction

After confirming that we can often factor NIDS resource usage components with per-analyzer and per-connection scaling, we now employ these observations to derive suggestions of reasonable configurations for operating in a specific network environment.

We start by devising a methodology for finding a suitable configuration based on a snapshot of an environment's network traffic. Then we turn to estimating the *long-term* performance of such a configuration given a coarser-grained summary of the network traffic that contains the time-of-day and day-of-week effects. The latter is crucial, as traffic characteristics, and therefore resource consumption, can change significantly over time.

5.1 From Traffic Snapshots to Configurations

In this section we consider the degree to which we can analyze a short sample trace from a given environment in order to identify suitable NIDS configurations, in terms of

[3] As in the case of CPU usage, we find inherent fluctuation in memory usage as well: running instances under identical conditions exhibits some noticeable, though not huge, variation.

maximizing the NIDS's analysis while leaving enough "head room" to avoid exhausting its resources. More generally, we wish to enable the network operator to make informed decisions about the prioritization of different types of analysis. Alternatively, we can help the operator decide whether to upgrade the machine if the available resources do not allow the NIDS to perform the desired analysis.

We stress that due to the variability inherent in network traffic, as well as the measurement limitations discussed in §4, no methodology can aim to suggest an *optimal* configuration. However, automating the process of exploring the myriad configuration options of a NIDS provides a significant step forward compared to having to assess different configurations in a time-consuming, trial-and-error fashion.

Capturing an Appropriate Trace. Our approach assumes access to a packet trace from the relevant network with a duration of some tens of minutes. We refer to this as the *main analysis trace*. At this stage, we assume the trace is "representative" of the busiest period for the environment under investigation. Later in this section we explore this issue more broadly to generalize our results.

Ideally, one uses a *full* packet trace with all packets that crossed the link during the sample interval. However, even for medium-sized networks this often will not be feasible due to disk capacity and time constraints: a 20-minute recording of a link transferring 400 Mbit/s results in a trace of roughly 60 GB; running a systematic analysis on the resulting trace as described below would be extremely time consuming. In addition, full packet capture at these sorts of rates can turn out to be a major challenge on typical commodity hardware [9].

We therefore leverage our finding that in general we can decompose resource usage on a per-connection basis and take advantage of the connection sampling methodology discussed in Section 4. Given a disk space budget as input, we first estimate the link's usage via a simple libpcap application to determine a suitable sampling factor, which we then use to capture an accordingly sampled trace. We can perform the sampling itself using an appropriate kernel packet filter [2], so it executes quite efficiently and imposes minimal performance stress on the monitoring system.

Using this trace as input, we then can scale our results according to the sample factor, as discussed in §4, while keeping in mind the most significant source of error in this process, which is a tendency to overestimate memory consumption when considering a wide range of application analyzers.

Finding Appropriate Configurations. We now leverage our observation that we can decompose resource usage per analyzer to determine analysis combinations that do not overload the system when analyzing a traffic mix and volume similar to that extrapolated from the captured analysis trace. Based on our analysis of the NIDS resource usage contributors (§3.2) and its verification (§4), our approach is straight-forward. First we derive a baseline of CPU and memory usage by running the NIDS on the sampled trace using a minimal configuration. Then, for each potentially interesting analyzer, we measure its additional resource consumption by individually adding it to the minimal configuration. We then calculate which combinations of analyzers result in feasible CPU and memory loads.

The main challenge for determining a suitable level of *CPU* usage is to find the right trade-off between a good detection rate (requiring a high average CPU load) and leaving sufficient head-room for short-term processing spikes. The higher the load budget, the more detailed the analysis; however, if we leave only minimal head-room then the system will likely incur packet drops when network traffic deviates from the typical load, which, due to the long-range dependent nature of the traffic [12] will doubtlessly happen. Which trade-off to use is a *policy decision* made by the operator of the NIDS, and depends on both the network environment and the site's monitoring priorities. Accordingly, we assume the operator specifies a target CPU load c together with a quantile q specifying the percentage of time the load should remain below c. With, for example, $c = 90\%$ and $q = 95\%$, the operator asks our tool to find a configuration that keeps the CPU load below 90% for 95% of all CPU samples taken when analyzing the trace.

Two issues complicate the determination of a suitable level of *memory* usage. First, some analyzers that we cannot (reasonably) disable may consume significant amounts of memory, such as TCP connection management as a precursor to application-level analysis for TCP-based services. Thus, the option is not whether to enable these analyzers, but rather how to *parameterize* them (e.g., in terms of setting timeouts). Second, as pointed out in §4, the memory requirements of some analyzers do not scale directly with the number of connections, rendering their memory consumption harder to predict.

Regarding the former, parameterization of analyzers, previous work has found that connection-level timeouts are a primary contributor to a NIDS's memory consumption [3]. Therefore, our first goal is to derive suitable timeout values given the connection arrival rate in the trace. The main insight is that the NIDS needs to store different amounts of state for different connection types. We can group TCP connections into three classes: *(i)* failed connection attempts; *(ii)* fully established and then terminated connections; and *(iii)* established but not yet terminated connections. For example, the Bro NIDS (and likely other NIDSs as well) uses different timeouts and data structures for the different classes [3], and accordingly we can examine each class separately to determine the corresponding memory usage. To predict the effect of the individual timeouts, we assume a constant arrival rate for new connections of each class, which is reasonable given the short duration of the trace. In addition, we assume that the memory required for connections within a class is roughly the same. (We have verified this for Bro.) This then enables us to estimate appropriate timeouts for a given memory budget.

To address the second problem, analyzer memory usage which does not scale linearly with the sampling factor, we can identify these cases by "subsampling" the main trace further, using for example an additional sampling factor of 3. Then, for each analyzer, we determine the total memory consumption of the NIDS running on the subsampled trace and multiply this by the subsampling factor. If doing so yields approximately the memory consumption of the NIDS running the same configuration on the main trace, then the analyzer's memory consumption does indeed scale linearly with the sampling factor. If not, then we are able to flag that analysis as difficult to extrapolate.

5.2 A Tool for Deriving NIDS Configurations

We implemented an automatic configuration tool, `nidsconf`, for the Bro NIDS based on the approach discussed above. Using a sampled trace file, it determines a set of Bro configurations, including sets of feasible analyzers and suitable connection timeouts. These configurations enable Bro to process the network's traffic within user-defined limits for CPU and memory.

We assessed `nidsconf` in the MWN environment on a workday afternoon with a disk space budget for the sampled trace of 5 GB; a CPU limit of $c = 80\%$ for $q = 90\%$ of all samples; a memory budget of 500 MB for connection state; and a list of 13 different analyzers to potentially activate (mostly the same as listed previously, but also including *http-reply* which examines server-side HTTP traffic).

Computed over a 10-second window, the peak bandwidth observed on the link was 695 Mbps. A 20-minute full-packet trace would therefore have required approximately 100 GB of data. Consequently, `nidsconf` inferred a connection sampling factor of 23 as necessary to stay within the disk budget (the next larger prime above the desired sampling factor of 21). The connection-sampled trace that the tool subsequently captured consumed almost exactly 5 GB of disk space. `nidsconf` then concluded that even by itself, full HTTP request/reply analysis would exceed the given c and q constraints. Therefore it decided to disable server-side HTTP analysis. Even without this, the combination of all other analyzers still exceeded the constraints. Therefore, the user was asked to chose one to disable, for which we selected *http-request*. Doing so turned out to suffice. In terms of memory consumption, `nidsconf` determined that the amount of state stored by three analyzers (HTTP, SSL, and the scan detector) did not scale linearly with the number of connections, and therefore could not be predicted correctly. Still, the tool determined suitable timeouts for connection state (873 secs for unanswered connection attempts, and 1653 secs for inactive connections).

Due to the complexity of the Bro system, there are quite a few subtleties involved in the process of automatically generating a configuration. Due to limited space, here we only outline some of them, and refer to [2] for details. One technical complication is that not all parts of Bro are sufficiently instrumented to report their resource usage. Bro's scripting language poses a more fundamental problem: a user is free to write script-level analyzers that consume CPU or memory in unpredictable ways (e.g., not tied to connections). Another challenge arises due to individual connections that require specific, resource-intensive analysis. As these are non-representative connections any sampling-based scheme must either identify such outliers, or possibly suggest overly conservative configurations. Despite these challenges, however, `nidsconf` provides a depth of insight into configuration trade-offs well beyond what an operator otherwise can draw upon.

5.3 From Flow Logs to Long-Term Prediction

Now that we can identify configurations appropriate for a short, detailed packet-level trace, we turn to estimating the *long-term* performance of such a configuration. Such extrapolation is crucial before running a NIDS operationally, as network traffic tends to exhibit strong time-of-day and day-of-week effects. Thus, a configuration suitable for a short snapshot may still overload the system at another time, or unnecessarily forsake some types of analysis during less busy times.

For this purpose we require long-term, coarser-grained logs of connection information as an abstraction of the network's traffic. Such logs can, for example, come from

NetFlow data, or from traffic traces with tools such as tcpreduce [10], or perhaps the NIDS itself (Bro generates such summaries as part of its generic connection analysis). Such connection-level logs are much smaller than full packet traces (e.g., $\ll 1\%$ of the volume), and thus easier to collect and handle. Indeed, some sites already gather them on a routine basis to facilitate traffic engineering or forensic analysis.

Methodology. Our methodology draws upon both the long-term connection log and the systematic measurements on a short-term, (sampled) full-packet trace as described above. We proceed in three steps: First, we group all connections (in both the log and the packet trace) into classes, such that the NIDS resource usage scales linearly with the class size. Second, for different configurations, we measure the resources used by each class based on the packet trace. In the last step, we project the resource usage over the duration of the connection log by scaling each class according to the number of such connections present in the connection log.

In the simplest case, the overall resource usage scales linearly with the *total* number of connections processed (for example, this holds for TCP-level connection processing without any additional analyzers). Then we have only one class of connections and can project the CPU time for any specific time during the connection log proportionally: if in the packet trace the analysis of N connections takes P seconds of CPU time, we estimate that the NIDS performing the same analysis for M connections uses $\frac{P}{N}M$ seconds of CPU time. Similarly, if we know the memory required for I concurrent connections at some time T_1 for the packet trace, we can predict the memory consumption at time T_2 by determining the number of active connections at T_2.

More complex configurations require more than one connection class. Therefore we next identify how to group connections depending on the workload they generate. Based on our observation that we can decompose a NIDS's resource requirements into that of its analyzers (§3), along with our experience validating our approach for Bro (§4), we identified three dimensions for defining connection classes: duration, application-layer service, and final TCP state of the connection (e.g., whether it was correctly established). Duration is important for determining the number of active connections in memory at each point in time; service determines the analyzers in use; and the TCP state indicates whether application-layer analysis is performed.

As we will show, this choice of dimensions produces resource-consumption predictions with reasonable precision for Bro. We note, however, that for other NIDSs one might examine a different decomposition (e.g., data volume per connection may have a strong impact too). Even if so, we anticipate that a small number of connection classes will suffice to capture the principle components of a NIDS's resource usage.

Predicting Long-Term Resource Use. We now show how to apply our methodology to predict the long-term resource usage of a NIDS, again using Bro as an example. We first aggregate the connection-level data into time-bins of length T, assigning attributes reflecting each of the dimensions: TCP state, service, and duration. We distinguish between five TCP states (attempted, established, closed, half-closed, reset), and consider 40 services (one for each Bro application-layer analyzer, plus a few additional well-known service ports, plus the service "other"). We discretize a connection's duration D by assigning it to a duration category $C \leftarrow \lfloor log_{10} D \rfloor$. Finally, for each time-bin we count the number of connections with the same attributes.

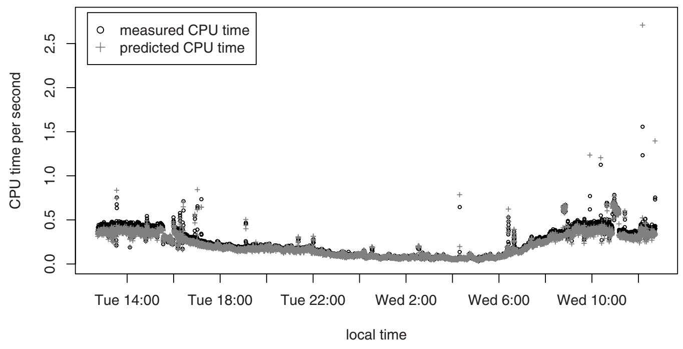


Fig. 5. Measured CPU time vs. predicted CPU time with BroBase

Simple CPU Time Projection. To illustrate how we then project performance, let us first consider a simple case: the BroBase configuration. As we have seen (§4), for this configuration resource consumption directly scales with the total number of connections. In Figure 5 we plot the actual per-second CPU consumption exhibited by running BroBase on the complete `MWN-full` trace (circles) versus the per-second consumption projected by using connection logs plus an independent 20-minute trace (crosses). We see that overall the predicted CPU time matches the variations in the measured CPU time quite closely. The prediction even correctly accounts for many of the outliers. However, in general the predicted times are somewhat lower than the measured ones with a mean error of -25 msec of CPU time per second, and a mean relative error of -9.0%.

CPU Time Projection for Complex Configurations. Let us now turn to predicting performance for more complex configurations. We examine BroAll$^-$, the BroAll configuration except with *ssl* deactivated (since the analyzer occasionally crashes the examined version of Bro in this environment). In this case, we group the connections into several classes, as discussed above. To avoid introducing high-variance effects from minimal samples, we discard any connections belonging to a service that comprises less than 1% of the traffic. (See below for difficulties this can introduce.) We then predict overall CPU time by applying our projection first individually to each analyzer and for each combination of service and connection state, and then summing the predicted CPU times for the base configuration and the predicted additional CPU times for the individual analyzers.

Figure 6 shows the resulting predicted CPU times (crosses) and measured BroAll$^-$ CPU times (circles). Note that this configuration is infeasible for a live setting, as the required CPU regularly exceeds the machine's processing capacity. We see, however, that our prediction matches the measurement fairly well. However, we underestimate some of the outliers with a mean error of -29 msec of CPU time and a mean relative error of -4.6%. Note that the mean relative error is smaller than for predicting BroBase performance since the absolute numbers of the measured samples are larger for the complex configuration.

Above we discussed how we only extrapolate CPU time for connections that contribute a significant portion ($> 1\%$) of the connections in our base measurement. Doing

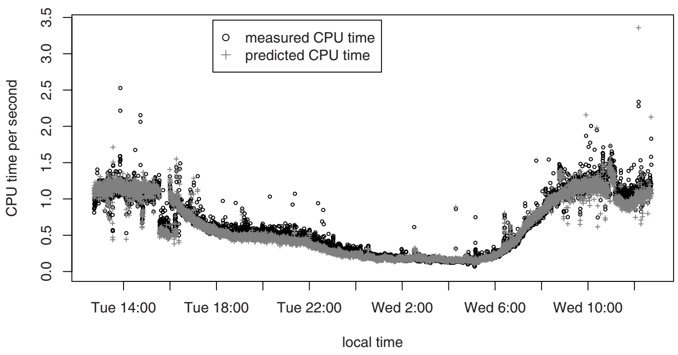


Fig. 6. Measured CPU time vs. predicted CPU time with BROALL^{-}

so can result in underestimation of CPU time when these connection types become more prominent. For example, during our experiments we found that SSH and Telnet connections did not occur frequently in the 20-minute trace on which the systematic measurements are performed. Yet the long-term connection log contains sudden surges of these connections (likely due to brute-force login attempts). `nidsconf` detects such cases and reports a warning, but at this point it lacks sufficient data to predict the CPU time usage, since it does not have an adequate sample in the trace from which to work.

Memory Projection. Our approach for predicting *memory* consumption is to derive the number of active connections per class at any given time in the connection log, and then extrapolate from this figure to the overall memory usage. However, Bro's resource profiling is not currently capable of reporting precise per-connection memory usage for application-layer analyzers, so here we limit ourselves to predicting the *number* of TCP connections in memory, rather than the actual memory consumption. To do so, we draw upon the dimensions of connection *duration* and *state*. These two interplay directly since Bro keeps its per connection state for the lifetime of the connection plus a timeout that depends on the state. To determine the relevant timeout, we use the states discussed above (attempted, established, etc.), binning connections into time intervals of length T and then calculating their aggregate memory requirements.

However, a problem with this binning approach arises due to connections with durations shorter than the bin size (since we use bin sizes on the order of tens of seconds, this holds for the majority of connections). Within a bin, we cannot tell how many of these are *concurrently* active. Therefore, we refine our basic approach, as follows. We pick a random point in the base trace and compute the average number N of short-lived connections per second occurring in the trace up to that point. We also measure the number F of these short-lived connections instantaneously in memory at the arbitrary point. Let N_i be the number of short-lived connections per second for each bin i in the connection log. Assuming that F is representative, we can then scale N_i/N by F to estimate the number of short-lived connections concurrently active in each bin.

Figure 7 shows the results of our prediction for the number of established connections in memory (crosses) assuming Bro's default inactivity timeout of 300s, along with the the actual number of in-memory connections when running on `MWN-full` (circles).

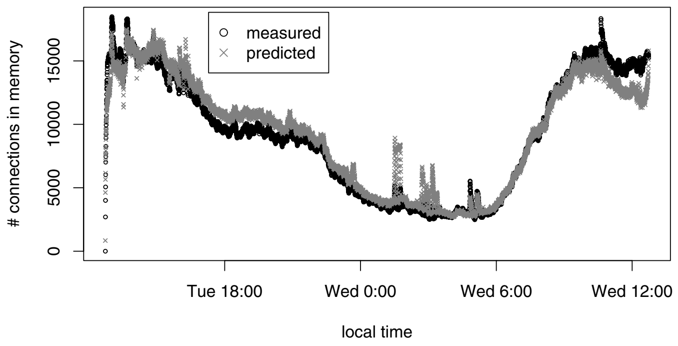


Fig. 7. Predicted number of established connections in memory for `MWN-full`

We observe that the prediction matches the measurements well, with a mean relative error of +5.0%. While not shown on the plot, we obtain similar prediction results for other classes of connections, e.g., unanswered connection attempts.

6 Related Work

Numerous studies in the literature investigate IDS detection quality, generally analyzing the trade-off between false positives and false negatives. Some studies [6,4,5] take steps towards analyzing how the detection quality and detection coverage depends on the cost of the IDS configuration and the attacks the network experiences. Gaffney and Ulvila [4] focus on the costs that result from erroneous detection, developing a model for finding a suitable trade-off between false positives and false negatives dependent on the cost of each type of failure. In contrast, Lee et al. [6,5] focus on developing and implementing high-level cost models for operating an IDS, enabling dynamic adaptation of a NIDS's configuration to suit the current system load. The models take as input both metrics of the benefits of a successful detection and (self-adapting) metrics reflecting the cost of the detection. Such metrics may be hard to define for large network environments, however. To adapt to the cost metrics, they monitor the performance of their prototype systems (Bro and Snort) using a coarse-grained instrumentation of packet counts per second. As was shown by Dreger et al. [3], this risks oversimplifying a complex NIDS. While the basic idea of adapting NIDS configurations to system load is similar to ours, we focus on predicting resource usage of the NIDS depending on both the network traffic and the NIDS configuration.

In the area of general performance prediction and extrapolation of systems (not necessarily NIDSs), three categories of work exam *(i)* performance on different hardware platforms, *(ii)* distribution across multiple systems, and *(iii)* predicting system load. These studies relate to ours in the sense that we use similar techniques for program decomposition and for runtime extrapolation. We omit details of these here due to limited space, but refer the reader to [2] for a detailed discussion. In contrast to this body of work, our contributions are to predict performance for *soft* real-time systems, both at a fine-grained resolution (prediction of "head room" for avoiding packet drops) and over long time scales (coupling a short, detailed trace with coarse-grained logs to extrapolate performance over hours or days), with an emphasis on memory and CPU trade-offs available to an operator in terms of depth of analysis versus limited resources.

7 Conclusion

In this work we set out to understand and predict the resource requirements of network intrusion detection systems. When initially installing such a system in a network environment, the operator often must grapple with a large number of options to tune trade-offs between detection rate versus CPU and memory consumption. The impact of such parameters often proves difficult to predict, as it potentially depends to a large degree on the internals of the NIDS's implementation, as well as the specific characteristics of the target environment. Because of this, the installation of a NIDS often becomes a trial-and-error process that can consume weeks until finding a "sweet spot."

We have developed a methodology to *automatically* derive NIDS configurations that maximize the systems' detection capabilities while keeping the resource load feasible. Our approach leverages the modularity likely present in a NIDS: while complex systems, NIDSs tend to be structured as a set of subcomponents that work mostly independently in terms of their resource consumption. Therefore, to understand the system as a whole, we can decompose the NIDS into the main contributing components. As our analysis of the open-source Bro NIDS shows, the resource requirements of these subcomponents are often driven by relatively simple characteristics of their input, such as number of packets or number and types of connections.

Leveraging this observation, we built a tool that derives realistic configurations for Bro. Based on a short-term, full-packet trace coupled with a longer-term, flow-level trace—both recorded in the target environment—the tool first models the resource usage of the individual subcomponents of the NIDS. It then simulates different configurations by adding together the contributions of the relevant subcomponents to predict configurations whose execution will remain within the limits of the resources specified by the operator. The operator can then choose among the feasible configurations according to the priorities established for the monitoring environment. While no automatically generated configuration can be optimal, these provide a sound starting point, with promise to significantly reduce the traditional trial-and-error NIDS installation cycle.

Acknowledgments

We would like to thank Christian Kreibich for his feedback and the fruitful discussions that greatly helped to improve this work. We would also like to thank the Leibnitz-Rechenzentrum München. This work was supported by a grant from the Bavaria California Technology Center, and by the US National Science Foundation under awards STI-0334088, NSF-0433702, and ITR/ANI-0205519, for which we are grateful. Any opinions, findings, conclusions or recommendations expressed in this material are those of the authors or originators and do not necessarily reflect the views of the National Science Foundation.

References

1. Dharmapurikar, S., Paxson, V.: Robust TCP Stream Reassembly In the Presence of Adversaries. In: Proc. USENIX Security Symposium (2005)
2. Dreger, H.: Operational Network Intrusion Detection: Resource-Analysis Tradeoffs. PhD thesis, TU München (2007), http://www.net.in.tum.de/~hdreger/papers/thesis_dreger.pdf
3. Dreger, H., Feldmann, A., Paxson, V., Sommer, R.: Operational Experiences with High-Volume Network Intrusion Detection. In: Proc. ACM Conference on Computer and Communications Security (2004)
4. Gaffney Jr., J.E., Ulvila, J.W.: Evaluation of Intrusion Detectors: A Decision Theory Approach. In: Proc. IEEE Symposium on Security and Privacy (2001)
5. Lee, W., Cabrera, J.B., Thomas, A., Balwalli, N., Saluja, S., Zhang, Y.: Performance Adaptation in Real-Time Intrusion Detection Systems. In: Proc. Symposium on Recent Advances in Intrusion Detection (2002)
6. Lee, W., Fan, W., Miller, M., Stolfo, S.J., Zadok, E.: Toward Cost-sensitive Modeling for Intrusion Detection and Response. Journal of Computer Security 10(1-2), 5–22 (2002)
7. Paxson, V.: Bro: A System for Detecting Network Intruders in Real-Time. Computer Networks 31(23–24), 2435–2463 (1999)
8. Roesch, M.: Snort: Lightweight Intrusion Detection for Networks. In: Proc. Systems Administration Conference (1999)
9. Schneider, F., Wallerich, J., Feldmann, A.: Packet Capture in 10-Gigabit Ethernet Environments Using Contemporary Commodity Hardware. In: Proc. Passive and Active Measurement Conference (2007)
10. tcp-reduce, http://ita.ee.lbl.gov/html/contrib/tcp-reduce.html
11. Vallentin, M., Sommer, R., Lee, J., Leres, C., Paxson, V., Tierney, B.: The NIDS Cluster: Scalable, Stateful Network Intrusion Detection on Commodity Hardware. In: Proc. Symposium on Recent Advances in Intrusion Detection (2007)
12. Willinger, W., Taqqu, M.S., Sherman, R., Wilson, D.V.: Self-Similarity Through High-Variability: Statistical Analysis of Ethernet LAN Traffic at the Source Level. IEEE/ACM Transactions on Networking 5(1) (1997)

High-Speed Matching of Vulnerability Signatures

Nabil Schear[1], David R. Albrecht[2], and Nikita Borisov[2]

[1] Department of Computer Science
[2] Department of Electrical and Computer Engineering
University of Illinois at Urbana–Champaign
{nschear2,dalbrech,nikita}@uiuc.edu

Abstract. Vulnerability signatures offer better precision and flexibility than exploit signatures when detecting network attacks. We show that it is possible to detect vulnerability signatures in high-performance network intrusion detection systems, by developing a matching architecture that is specialized to the task of vulnerability signatures. Our architecture is based upon: i) the use of high-speed pattern matchers, together with control logic, instead of recursive parsing, ii) the limited nature and careful management of implicit state, and iii) the ability to avoid parsing large fragments of the message not relevant to a vulnerability.

We have built a prototype implementation of our architecture and vulnerability specification language, called VESPA, capable of detecting vulnerabilities in both text and binary protocols. We show that, compared to full protocol parsing, we can achieve 3x or better speedup, and thus detect vulnerabilities in most protocols at a speed of 1 Gbps or more. Our architecture is also well-adapted to being integrated with network processors or other special-purpose hardware. We show that for text protocols, pattern matching dominates our workload and great performance improvements can result from hardware acceleration.

1 Introduction

Detecting and preventing attacks is a critical aspect of network security. The dominant paradigm in network intrusion detection systems (NIDS) has been the *exploit signature*, which recognizes a particular pattern of misuse (an *exploit*). An alternative approach is to use a *vulnerability signature*, which describes the *class* of messages that trigger a vulnerability on the end system, based on the behavior of the application. Vulnerability signatures are exploit-generic, as they focus on how the end host interprets the message, rather than how the particular exploit works, and thus can recognize polymorphic and copycat exploits.

Exploit signatures are represented using byte-string patterns or regular expressions. Vulnerability signatures, on the other hand, usually employ protocol parsing to recover the semantic content of the communication and then decide whether it triggers a vulnerability. The semantic modeling allows vulnerability signatures to be both more general and more precise than exploit signatures. However, this comes at a high performance cost. To date, vulnerability signatures have only been considered for user on end hosts, severely limiting their deployment.

In our work, we observe that full and generic protocol parsing is *not necessary* for detecting vulnerability signatures. Using custom-built, hand-coded vulnerability signature recognizers, we show that these signatures can be detected 3 to 37 times faster than the speed of full protocol parsing. Therefore, there is no *inherent* performance penalty for using vulnerability signatures instead of exploit signatures.

Motivated by this, we design an architecture, called VESPA[1], for matching vulnerability signatures at speeds adequate for a high-performance enterprise NIDS, around 1 Gbps. We build our architecture on a foundation of fast string and pattern matchers, connected with control logic. This allows us to do deep packet inspection and model complex behavior, while maintaining high performance. We also minimize the amount of implicit state maintained by the parser. By avoiding full, in-memory semantic representation of the message, we eliminate much of the cost of generic protocol parsing. Finally, in many cases we are able to eliminate the recursive nature of protocol analysis, allowing us to skip analysis of large subsections of the message.

We have implemented a prototype of VESPA; tests show that it matches vulnerability signatures about three times faster than equivalent full-protocol parsing, as implemented in binpac [1]. Our architecture matches most protocols in software at speeds greater than 1 Gbps. Further, we show that our text protocol parsing is dominated by string matching, suggesting that special-purpose hardware for pattern matching would permit parsing text protocols at much higher speeds. Our binary protocol parsing is also well-adapted to hardware-aided implementation, as our careful state management fits well with the constrained memory architectures of network processors.

The rest of this paper is organized as follows: Section 2 gives some background on vulnerability signatures and discusses the context of our work. Sections 3 and 4 describe the design of VESPA and the vulnerability signature language. We present the implementation details of VESPA in Section 5. Section 6 contains a performance evaluation of our prototype. We discuss some future directions in Section 7 and related work in Section 8. Finally, Section 9 concludes.

2 Background

2.1 Vulnerability Signatures

Vulnerability signatures were originally proposed by Wang et al. [2] as an alternative to traditional, exploit-based signatures. While exploit signatures describe the properties of the exploit, vulnerability signatures describe how the vulnerability gets triggered in an application. Consider the following exploit signature for Code Red [3]:

```
urlcontent:"ida?NNNNNNNNNNNNN..."
```

The signature describes how the exploit operates: it uses the ISAPI interface (invoked for files with extension "`.ida`") and inserts a long string of `N`'s, leading to a buffer overflow. While effective against Code Red, this signature would not match Code Red II [4]; that variant used `X`'s in place of the `N`'s. A vulnerability signature, on the other hand, does not specify how the worm works, but rather how the application-level vulnerability is triggered. An extract from the CodeRed signature in Shield [2] is:

[1] VulnErability Signature Parsing Architecture.

```
c = MATCH_STR_LEN(>>P_Get_Request.URI,"id[aq]\?(.*)$",limit);
IF (c > limit)
  # Exploit!
```

This signature captures *any* request that overflows the ISAPI buffer, making it effective against Code Red, Code Red II, and any other worm or attack that exploits the ISAPI buffer overflow. In fact, this signature could well have been written before the release of either of the Code Red worms, as the vulnerability in the ISAPI was published a month earlier [5]. Thus, while exploit signatures are reactive, vulnerability signatures can proactively protect systems with known vulnerabilities until they are patched (which can take weeks or months [6]).

2.2 Protocol Parsing

Traditionally, exploit signatures are specified as strings or regular expressions. Vulnerability signatures, on the other hand, involve some amount of protocol parsing. Shield [2] used a language for describing C-like binary structures, and an extension for parsing text protocols. The follow-on project, GAPA [7], designed a generic application-level protocol analyzer to be used for matching vulnerability signatures. GAPA represented both binary and text protocols using a recursive grammar with embedded code statements. The generated GAPA parser, when guided by code statements, performed context-sensitive parsing. GAPA aimed to provide an easy-to-use and safe way to specify protocols and corresponding vulnerabilities.

Binpac [1], another protocol parser, was designed to be used in the Bro intrusion detection system [8]. Binpac is similar to GAPA: both use a recursive grammar and embedded code for parsing network protocols, and both are intended to minimize the risks of protocol parsing. Binpac, however, is designed only for parsing, with other parts of Bro performing checks for alarms or vulnerabilities. Binpac uses C++ for its embedded code blocks, and compiles the entire parser to C++ (similar to yacc), whereas GAPA uses a restricted, memory-safe interpreted language capable of being proven free of infinite loops. Binpac trades some of GAPA's safety for parsing speed; consequently, it achieves speeds comparable to hand-coded parsers written for Bro.

Since the implementation of GAPA is not freely available, we use binpac as our prototypical generic protocol parser generator in comparing to our work. Binpac is significantly faster than GAPA, yet it is not able to parse many protocols at speeds of 1 Gbps (though sparing use of binpac, where most data passing through the NIDS is not analyzed, can be supported.)

2.3 Vulnerability Complexity

Although Shield and GAPA used protocol parsing for vulnerability signatures, Brumley et al. suggest that vulnerability signatures could be represented across a spectrum of complexity classes [9]. They consider the classes of regular expressions, constraint satisfaction languages, and Turing machines, and provide algorithms to derive automatic vulnerability signatures of each class. As increasingly complex specifications of signatures are used, the precision of signature matching improves.

We make a different observation: most vulnerability signatures can be matched *precisely* without full protocol parsing. And such precise matching can be carried out at much greater speeds. In Table 1, we compare the performance of binpac to hand-coded implementations of several vulnerability signatures. We wrote the hand-coded implementations in C and designed them to match one specific vulnerability only. These would fall into the Turing machine class according to Brumley et al., but they are optimized for speed. Notice that the hand-coded implementations operate about *3x to 37x faster* than equivalent binpac implementation.

Table 1. The throughput (Mbits/s) of binpac parsers vs. hand-coded vulnerability matchers

Protocol	binpac	hand-coded
CUPS/HTTP	5,414	20,340
DNS	71	2,647
IPP	809	7,601
WMF	610	14,013

To see why this is the case, consider the following CUPS vulnerability (CVE-2002-1368 [10]). CUPS processes the IPP protocol, which sends messages embedded inside HTTP requests. CUPS would crash if a negative `Content-Length` were specified, presenting a denial-of-service opportunity. Our binpac implementation to check for this vulnerability is based on the binpac HTTP specification, which parses the HTTP header into name–value pairs. We add a constraint that looks for header names that match `Content-Length` and verifies that a non-negative value is used. Our hand-coded implementation, on the other hand, is built upon an Aho–Corasick [11] multi-string matcher, which looks for the strings "`Content-Length:`" and "`\r\n\r\n`" (the latter indicating the end of the headers). If "`Content-Length:`" is found, the following string is parsed as an integer and checked for being non-negative.

The parsers operate with equal precision when identifying the vulnerability, yet the hand-coded approach performs much less work per message, and runs more than 3 times as quickly. Of course, not all vulnerabilities can be matched with a simple string search. However, what this vulnerability demonstrates is that an efficient vulnerability signature matching architecture must be able to handle such simple vulnerabilities quickly, rather than using heavy-weight parsing for all vulnerabilities, regardless of complexity. The architecture will surely need to support more complex constructs as well, but they should only be used when necessary, rather than all the time. We next present a new architecture for specifying and matching vulnerability signatures that follows this principle. Our architecture shares some of the goals of binpac and GAPA; however, it puts a stronger focus on performance, rather than generality (GAPA) or ease-of-authoring (binpac).

3 Design

To make vulnerability signatures practical for use in network intrusion detection systems, we developed VESPA, an efficient vulnerability specification and matching architecture. The processes of writing a protocol specification and writing a vulnerability signature are coupled to allow the parser generator to perform optimizations on the generated code that specialize it for the vulnerabilities the author wishes to match.

Our system is based on the following design principles:

– Use of fast matching primitives
– Explicit state management
– Avoiding parsing of irrelevant message parts

Since text and binary protocols require different parsing approaches, we describe our design of each type of parser and how we apply the design principles listed above. We first give a brief outline of how the system works, and then go into detail in the subsequent sections on how our approach works.

We use fast matching primitives—string matching, pattern matching (regular expressions), and binary traversal—that may be easily offloaded to hardware. The signature author specifies a number of matcher primitive entries, which correspond to fields needed by the signature to evaluate the vulnerability constraint. Each matcher contains embedded code which allows the matching engine to automatically extract a value from the result of the match. For example, the HTTP specification includes a string matcher for "`Content-Length:`", which has an extraction function that converts the string representation of the following number to a integer.

Along with each matcher, the author also specifies a handler function that will be executed following the extraction. The handlers allow the signature author to model the protocol state machine and enable additional matchers. For example, if a matcher discovers that an HTTP request message contains the POST command, it will in turn enable a matcher to parse and extract the message body. We also allow the author to define handlers that are called when an entire message has been matched.

The author checks vulnerability constraints inside the handler functions. Therefore constraint evaluation can be at the field level, intra-message level, and inter-message level. Depending on the complexity of the vulnerability signature, the author can choose where to evaluate the constraint most efficiently.

3.1 Text Protocols

We found that full recursive parsing of text protocols is both too slow and unnecessary for detecting vulnerabilities. However, simple string or regular expression matching is often insufficient to express a vulnerability constraint precisely in cases where the vulnerability depends on some protocol context. In our system, we combine the benefits of the two approaches by connecting multiple string and pattern matching primitives with control logic specialized to the protocol.

Matching Primitives. To make our design amenable to hardware acceleration we built it around simple matching primitives. At the core, we use a fast multi-string matching algorithm. This allows us to approximate the performance of simple pattern-based IDSes for simple vulnerability signatures. Since our system does not depend on any specific string matching algorithm, we have identified several well-studied algorithms [11,12] and hardware optimizations [13] that could be employed by our system.

Furthermore, hardware-accelerated regular expression matching is also becoming a reality [14]. As discussed later, this would further enhance the signature author's ability to locate protocol fields.

Minimal Parsing and State Managment. We have found that protocol fields can be divided into two categories: core fields, which define the structure and semantics of the protocol, and application fields, which have meaning to the application, but are not necessary to understand the rest of the message. An example of a core field is the `Content-Length` in HTTP, as it determines the size of the message body that follows in the protocol, whereas a field such as `Accept-Charset` is only relevant to the application.

Our approach in writing vulnerability signatures is to parse and store only the core fields, and the application fields relevant to the vulnerability, while skipping the rest. This allows us to avoid storing irrelevant fields, focusing our resources on those fields that are absolutely necessary.

Although many text protocols are defined in RFCs using a recursive BNF grammar, we find that protocols often use techniques that make identification of core fields possible without resorting to a recursive parse. For example, HTTP headers are specified on a separate line; as a result, a particular header can be located within a message by a simple string search. Header fields that are not relevant to a vulnerability will be skipped by the multi-string matcher, without involving the rest of the parser. Other text protocols follow a similar structure; for example, SMTP uses labeled commands such as "`MAIL FROM`" and "`RCPT TO`", which can readily be identified in the message stream.

3.2 Binary Protocols

While some of the techniques we use for text protocol parsing apply to binary protocols as well, binary protocols pose special challenges that must be handled differently from text.

Matching Primitives. Unlike text protocols, binary protocols often lack explicit field labeling. Instead, a parser infers the meaning of a field from its position in the message—relative to either the message start, or to other fields. In simple cases, the parser can use fixed offsets to find fields. In more complicated cases, the position of a field varies based on inter-field dependencies (e.g., variable-length data, where the starting offset of a field in a message varies based on the length of earlier fields), making parsing data-dependent. Thus, parsers must often traverse many or all of the preceding fields. This is still simpler than a full parse, since the parser only examines the lengths and values of structure-dependent fields.

Since binary protocols are more heavily structured than text protocols, we need a matching primitive that is sufficiently aware of this structure while still maintaining high performance. We call this type of parser a binary traverser.

Designing an efficient binary protocol traverser is difficult because binary protocol designs do not adhere to any common standard. In our study of many common binary protocols, we found that they most often utilize the following constructs: C structures, arrays, length-prefixed buffers, sentinel-terminated buffers, and field-driven case evaluation (switch). The binpac protocol parser generator uses variations on these constructs as building blocks for creating a protocol parser. We found binpac to have sufficient expressive power to generate parsers for complex binary protocols. However, binpac parsers perform a full protocol parse rather than a simple binary traversal, so we use a modification to improve their performance.

Minimal Parsing and State Management. We reduced overhead of original binpac parsers for state management and skipped parsing unimportant fields. Because binpac carefully separates the duties of the protocol parser and the traffic analysis system which uses it, we were able to port binpac specifications written for the Bro IDS to our system. We retain the protocol semantics and structure written in the Bro versions but use our own system for managing state and expressing constraints. While we feel that additional improvements may be made in generating fast binary traversers, we were able to obtain substantial improvements in the performance of binpac by optimizing it to the task of traversal rather than full parsing. Furthermore, the binpac language provides exceptional expressiveness for a wide range of protocols, allowing our system to be more easily deployed on new protocols.

3.3 Discussion

By flattening the protocol structure, we can ignore any part of a message which does not directly influence properly processing the message or matching a specific vulnerability. However, some protocols *are* heavily recursive and may not be flattened completely without significantly reducing match precision. We argue that it is rarely necessary to understand and parse *each and every* field and structural construct of a protocol message to match a vulnerability. Consider an XML vulnerability in the skin processing of Trillian (CVE-2002-2366 [10]). An attacker may gain control of the program by passing an over-length string in a `file` attribute, leading to a traditional buffer overflow. Only the `file` attribute, in the `prefs/control/colors` entity can trigger the vulnerability, while instances of `file` in other entities are not vulnerable. To match this vulnerability with our system, the signature author can use a minimal recursive parser which only tracks entity open and close tags. The matcher can use a stack of currently open tags to tell whether it is in the `prefs/control/colors` entity and match `file` attributes which will cause the buffer overflow. The generated parser is recursive but only for the specific fields that are needed to match the vulnerability. This type of signature is a middle-ground for our system—it will provide higher performance than a full parser while requiring the user to manipulate more state than a simpler vulnerability.

In rare cases it may be necessary to do full protocol parsing to properly match a vulnerability signature. While our system is designed to enhance the performance of simpler vulnerability signatures, it is still able to generate high-performance full recursive parsers. The drawback to our approach versus binpac or GAPA in this situation is that the user must manage the parser state manually, which may be error prone.

We do not yet address the problem of protocol detection. However, our system can be integrated with prior work [15] in an earlier stage of the intrusion detection system.

```
parser HTTP_Request {
  dispatch() %{   deploy(vers);   deploy(is_post);   deploy(crlf);   }%

  int vers = str_matcher "HTTP/1."
        handler handle_vers()
        %{      end = next_whitespace(rest);
                vers = str_to_int(rest,end);    }%

 handle_vers() %{  // handle differently depending on version... }%

  bool is_post = str_matcher "POST"
        handler handle_post()
        %{   is_post=true;   }%

  handle_post() %{    if(is_post) { deploy(content_length);  }   }%

  int content_length = str_matcher "Content-Length:"
        handler handle_cl()
        %{   end = next_line(rest);
             content_length = str_to_int(rest,end);   }%

  handle_cl() %{   if(this->content_length < 0) { // EXPLOIT! }
                   else  { deploy(body); }      }%

  bool crlf = str_matcher "\r\n\r\n" || "\n\n"
        %{ // do nothing explicit here }%

  Buffer body = extended_matcher crlf
        handler handle_body()
        %{   body = Buffer(rest,this->content_length);
             stopMachine();   }%

  handle_body() %{  // process body using another layer  }%
}
```

Fig. 1. Sample Specification for HTTP Requests (simplified)

Furthermore, the high-speed matching primitives used by VESPA may also be used to match protocol detection signatures.

4 Language

We have developed a vulnerability signature expression language for use with our system. We give an example vulnerability specification for the CUPS negative content length vulnerability in Figure 1.

Writing a signature involves specifying the matchers for the core fields of the protocol message and then specifying additional matchers to locate the vulnerability. We specify a single protocol message using a *parser* type. The code generator maps this message parser to a C++ class that will contain each state field as a member variable. Inside a message parser, the vulnerability signature author defines handler function declarations and field variable declarations with matching primitives. The author can specify additional member variables that are not directly associated with a matcher using `member_vars %{ ... }%`.

Each underlying matching primitive always searches for *all* the requested strings and fields with which the matcher is initialized. For example, an HTTP matcher might search for "`Content-Type:`" in a message even though this string should only be expected in certain cases. This allows the primitive matcher to run in parallel with the state machine and constraint evaluation, though we have not yet implemented this. It also prevents the matching primitives from needing to back up to parse a newly desired field. We provide a utility for keeping track of which fields the matcher should expect and perform extraction and which to ignore. This state is controlled using the `deploy(`*`var`*`)` function. This function may be called from any handler function, and initially by the `dispatch` function. `deploy` marks a variable as expected in a state mask stored inside the parser. This will cause the matcher to execute the variable extraction function and handler when it is matched. A handler function may in turn enable additional matchers (including re-enabling itself) using the `deploy` function. The parser ignores any primitive match that is not set to be active using `deploy`.

The parser automatically calls the `dispatch` function each time the parser starts parsing a new protocol message. This allows the author to define which fields should be matched from the start of parsing. It also allows the initialization of member variables created using `member_vars`. Conversely, the parser automatically calls `destroy` to allow any resources allocated in `dispatch` to be freed.

4.1 Matcher Primitives

Protocol fields and matcher primitives are the heart of a vulnerability specification. The format of matcher primitive specification is:

```
var_type symbol = matching_primitive meta-data
        handler handler_func_name()
        %{
                // embedded C++ code to extract the value
        }%
```

The `var_type` specifies the storage type of the field; e.g., `uint32`. The symbol is the name of the field that will be stored as a member of the C++ parser class. There are three types of matching primitives.

1. `str_matcher` (string matcher primitive): The meta-data passed to this matcher are a string or sequence of strings separated by ||, and this instructs the underlying multi-string matching engine to match this string and then execute its extraction function. It supports matching multiple different strings that are semantically identical using *or* ("||").
2. `bin_matcher` (binary traversal primitive): The meta-data passed to this matcher are the file name of a binpac specification. This is followed by a colon and the name of a binpac `record` type. The meta-data end with the name of a field inside that `record` that the author wishes to extract (*e.g.*, IPP.binpac: IPP_Message.version_num). The generated binpac parser will then call back to our system to perform the extraction and run the handler for the requested field.
3. `extended_matcher` (extension to another matcher): This construct allows us to perform additional extractions after matching a single string or binary field. This is often useful when multiple fields are embedded after a single match. It also allows

the author to specify a different extraction function depending on which state is expected. The meta-data passed to this primitive are the name of another variable that uses a standard matching primitive.

Each variable match also specifies an extraction function within braces, $\%\{$ and $\}\%$, which extracts a relevant field from the message. We have provided a number of helper functions that the author can use in the extraction function, such as string conversion and white space elimination. In a string matcher extraction function, there are two predefined variables the signature author can use and modify: `rest` and `end`. The `rest` variable points to the first byte of input after the string that was matched. The parser also defines `end`, which allows the extraction function to store where the extraction ends. Extended matchers run immediately following the extraction function of the string matcher on which they depend and in the same context. Hence, any changes to the state of `rest` and `end` should be carefully accounted for in extended matcher extraction functions.

There are two additional functions that the author can use inside the extraction function of a string matcher: `stopMachine()` and `restartMachine(`*`ptr`*`)`. These functions suspend and restart pattern matching on the input file. This is useful, for example, to prevent the system from matching spurious strings inside the body of an HTTP message. The `restartMachine(`*`ptr`*`)` function restarts the pattern matching at a new offset specified by *`ptr`*. This allows the matcher to skip portions of the message.

4.2 Handlers

Each matcher may also have an associated handler function. The handler function is executed after the extraction and only if the matcher is set to be active with `deploy`. The signature author defines the body of the handler function using C++ code. In addition to calling the `deploy` function, handler bodies are where vulnerability constraints can be expressed. We do not yet address the reporting mechanism when a vulnerability is matched. However, since any C++ code may be in the handler, the author may use a variety of methods, such as exceptions or integer codes. The author may also use the handler functions to pass portions of a protocol message to another parser to implement layering and encapsulation.

While structurally different from existing protocol parser generators like GAPA and binpac, our language is sufficiently expressive to model many text and binary protocols and vulnerabilities. Porting a protocol specification from an RFC or an existing spec in another language (like binpac or GAPA) is fairly straightforward once the author understands the protocol semantics.

5 Implementation

5.1 Compiler

We designed a compiler to generate machine-executable vulnerability signature matchers from our language. We implemented the compiler using the Perl programming language. Our implementation leverages the “Higher Order Perl” [16] Lexer and Parser classes, which kept down the implementation complexity: the entire compiler is 600 lines. Approximately 70% of the compiler code specifies the lexical and grammatical structures of our language; the balance performs symbol rewriting, I/O stream management, and boilerplate C++ syntax.

Our compiler operates on a single parser file (e.g., `myparser.p`), which defines a signature matcher. The generated code is a C++ class which extends one of the parser super classes. The class definition consists of two files (following the example above, `myparser.h` and `myparser.cc`), which jointly specify the generated parser subclass.

5.2 Parser Classes

Generated C++ classes for both binary and text parsers are structurally very similar, but differ in how they interface with the matching primitives. We have optimized the layout and performance of this code. We use inlined functions and code whenever possible. Many extraction helper functions are actually macros to reduce unnecessary function call overhead. We store the expected state set with `deploy` using a bit vector.

For string matchers, we use the sfutil library from Snort [17], which efficiently implements the Aho–Corasick (AC) algorithm [11]. Because the construction of a keyword trie for the AC algorithm can be time-consuming, we generate a separate reusable class which contains the pre-built AC trie. Our text matcher is not strongly tied to this particular multi-string matching implementation, and we have also prototyped it with the libSpare AC implementation [18].

We use binpac to generate a binary traverser for our parsers. As input, the compiler expects a binpac specification for the binary protocol. This should include all the `record` types in the protocol as well as the basic `analyzer`, `connection`, and `flow` binpac types. We then use the `refine` feature of binpac to embed the extraction functions and callbacks to our parser. Since binpac does simple extractions automatically, it is often unnecessary to write additional code that processes the field before it is assigned. Like the AC algorithm for text parsers, the binary parser is not heavily tied to the binary traversal algorithm or implementation. For a few protocols, we have developed hand-coded replacements for binpac binary traversal.

5.3 Binary Traversal-Optimized Binpac

We have made several modifications to the binpac parser generator to improve its performance for binary traversal. The primary enhancement we made is to change the default model for the in-memory structures binpac keeps while parsing. The original binpac allocated a C++ class for each non-primitive type it encountered while parsing. This resulted in an excessive number of calls to `new`, even for small messages. To alleviate this problem, we changed the default behavior of binpac to force all non-primitive types to be pre-allocated in one object. We use the `datauint` type in binpac to store all the possible subtypes that binpac might encounter. To preserve binpac semantics, we added a new function, `init(`*`params...`*`)`, to each non-primitive type in binpac. The `init` function contains the same code as the constructor, and we call it wherever a new object would have been created. It also accepts any arguments that the constructor takes to allow fields to be propagated from one object to another. We restrict binpac specifications to be able to pass only primitive types from object to object. While this reduces our compatibility with existing binpac specifications, it is easy to change them to support this limitation.

Some objects in binpac *must* be specified using a pointer to a dynamically created object and cannot be pre-allocated. For example, in the Bro DNS binpac specification, a `DNS_name` is composed of `DNS_labels`. A `DNS_label` type also contains a `DNS_name` object if the label is a pointer to another name. This circular dependency is not possible with statically sized classes. We added the `&pointer` attribute modifier to the binpac language to allow the author to specifically mark objects that must be dynamically allocated.

The final modification we made to binpac was to change the way that it handled arrays of objects. The original version of binpac created a vector for each array and stored each element separately. Because binary traversal only needs to access the data as it is being parsed, we do not need to store the entire array, only the current element. We eliminated the vector types entirely and changed binpac to only store the current element in the array using a pre-allocated object. If the author needs to store data from each element in the array, he must explicitly store it outside of binpac in the VESPA parser class using a handler function.

6 Evaluation

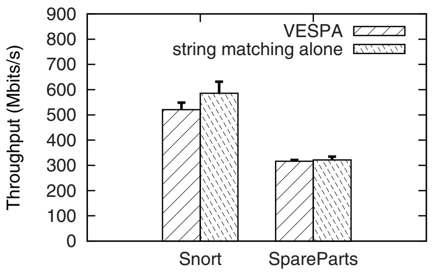
We evaluated VESPA with vulnerabilities in both text and binary protocols. We implemented matchers for vulnerabilities in the HTTP, DNS, and IPP protocols. We searched for exploitable bugs in network-facing code, focusing especially on scenarios where traditional exploit signatures would fail. Like Cui et al. did with GAPA [19], we found the process of writing a vulnerability signature for a protocol very similar to writing one for a file format. Thus, we used our system develop to a binary parser for the Windows Meta-file Format (WMF).

We ran all our experiments on an Ubuntu 7.10 Linux (2.6.22-14-x86_64) system with a dual-core 2.6 GHz AMD Athlon 64 processor and 4GB of RAM (our implementation is single-threaded so we only utilized one core). We ran the tests on HTTP and DNS on traces of real traffic collected from the UIUC Coordinated Science Laboratory network. We collected WMF files from freely available clipart websites. Since we did not have access to large volumes of IPP traffic, we tested using a small set of representative messages. We repeated the trace tests 10 times, and we repeated processing the IPP messages 1 million times to normalize any system timing perturbations. We show the standard deviation of these runs using error bars in the charts.

6.1 Micro-benchmarks of Matching Primitives

To evaluate the performance of using fast string matching primitives, we implemented our parser using two different implementations of the Aho–Corasick (AC) algorithm and compared their performance (Figure 2a). We used the sfutil library, which is part of the Snort IDS [17], and the Spare Parts implementation of AC [18]. We used those base implementations to search for the same strings as our vulnerability matcher does, but without any of the control logic or constraint checking. We found that for either AC implementation, the performance of a basic HTTP vulnerability matcher (which handles optional bodies and chunking) was very close to that of the string matching primitive.

The performance of string matching alone approximates (generously) the performance of a simple pattern-based IDS. If the vulnerability signature is simple enough to be expressed using a simple string match (e.g., the IPP vulnerability for a negative `Content-Length`), our system is able to match it with comparable performance to a pattern based IDS.



(a) Comparison between string matching primitive and parsing for HTTP requests

Parser Type	Bytes allocated	Num calls to `new`
DNS (binpac)	15,812	539
DNS (traversal)	2,296	14
IPP (binpac)	1,360	33
IPP (traversal)	432	6
WMF (binpac)	3,824	94
WMF (traversal)	312	6

(b) Dynamic memory usage for a single message for standard binpac vs. binary traversal

Fig. 2. Micro-benchmarks

We next investigated the performance of binary traversal in binpac. One of the primary changes we made to binpac was to change its default memory and allocation behavior. We instrumented the original version of binpac and a parser built with our binary traversal-optimized version to assess the effectiveness of this change (Figure 2b). We saw an overall reduction in memory usage despite pre-allocating types that may not be present in the message. We were also able to cut the number of calls to `new` by a substantial factor for all three binary protocols we implemented. Our IPP and WMF traversers do not contain any explicit pointer types (specified with `&pointer`), so the number of allocated blocks is constant for *any* protocol message. The number of times the DNS parser calls the `new` allocator is proportional to the number of name pointers in the message.

6.2 Signature Matching Performance

We evaluated the throughput of our vulnerability signature matching algorithms compared to the binpac parser generator. Binpac is the most efficient freely available automated protocol parser generator. We do not evaluate against GAPA because it has not been publicly released. Furthermore, binpac far exceeds GAPA in performance because it directly generates machine code rather than being interpreted [1]. Since binpac is not specifically designed for vulnerability signatures, we added vulnerability constraint checking to the binpac protocol specifications. In each of the following sections we describe the protocol and vulnerabilities we tested against. We show the results in Figure 3.

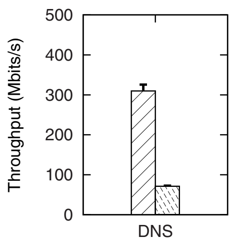
HTTP/IPP. The Common Unix Printing System (CUPS), with its protocol encapsulation and chunk-capable HTTP parser, illustrates several design choices which confound exploit-signature writers. The vulnerability given in CVE-2002-0063 [10] occurs because of the way the Internet Printing Protocol (IPP) specifies a series of textual key–value pairs, called attributes. The protocol allows attribute lengths to vary, requiring the sender to use a 16-bit unsigned integer to specify the length of each attribute. CUPS reads the specified number of bytes into a buffer on the stack, but the buffer is only 8192 bytes long, allowing an attacker to overflow the buffer and execute arbitrary code with the permissions of the CUPS process. A signature for this attack must check that each attribute length is less than 8192. IPP is a binary protocol but it is encapsulated inside of chunked HTTP for transport. Attackers can obfuscate the exploit by splitting it across an arbitrary number of HTTP chunks, making it very hard to detect this attack with pattern-based signatures. We also tested the negative content length vulnerability that we have discussed previously.

We designed a text-based vulnerability signature matcher for HTTP. In addition to vulnerabilities in HTTP itself, many protocols and file formats which are encapsulated inside of HTTP also have vulnerabilities. We use VESPA to match the `Content-Length` vulnerability in CUPS/IPP, as well as to extract the body of the message to pass it to another layer for processing. We support standard and chunked message bodies and pass them to a null processing layer. Unfortunately, we were unable to make a direct comparison to binpac for chunked HTTP messages due to a bug in binpac's buffering system: binpac will handle such a message but fail to extract data from each individual chunk. Despite this, we found that VESPA was considerably faster than the equivalent binpac parser. Since much of the HTTP message body is ignored by both VESPA and binpac, the throughputs we observed are very high because the size of the body contributes to the overall number of bytes processed. We also measured the message processing rates for various types of HTTP messages and found them to be adequate to process the traffic of a busy website (Table 2).

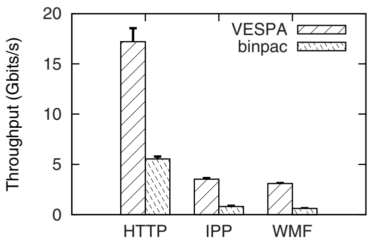
We implemented a binary IPP vulnerability matcher to be used in conjunction with our HTTP parser. The VESPA IPP matcher ran four times as fast as the binpac version, largely due to the improved state management techniques we described earlier. We also developed a hand-coded drop-in replacement for our binpac binary traverser of the IPP protocol. Using this replacement, we were able to achieve an order of magnitude improvement over the performance of the binpac binary traversal (see Table 1).

Table 2. HTTP Message Rate

HTTP Message Type	Message Rate (msgs per sec)
Requests	370,005
Responses	196,897
Chunked	41,644
Overall	314,797



(a) DNS Throughput (Mbps)



(b) Parser Throughput (Gpbs)

Fig. 3. Vulnerability Signature Matcher Performance

Therefore, our architecture stands to benefit from further improvements of the base matching primitives of binary traversal as well.

DNS. The DNS protocol includes a compression mechanism to avoid including a common DNS suffix more than once in the same message. Parsing these compressed suffixes, called name pointers, is best done with a recursive parser, but doing so introduces the possibility of a "pointer cycle," where a specially-crafted message can force a parser to consume an infinite amount of stack space, leading to a denial of service [20].

DNS name pointers can occur in many different structures in DNS, so the binary traversal must parse and visit many of the fields in the protocol. Therefore, parsing DNS is usually much slower than other protocols. Indeed, DNS is the worst-performing of our vulnerability signature matchers, though it is still several times faster than binpac, as can be seen in Figure 3. Pang et al. suggest that this is due to an inherent difficulty of parsing DNS, pointing to the comparable performance of their hand-implemented parser to binpac [1]. We have found this not to be the case, as our hand-implemented DNS parser that finds pointer cycles can operate at nearly 3 Gbps (see Table 1). As part of our future work, we will investigate what part of our current design is responsible for the much worse performance of DNS; our hope is that we will be able to achieve speeds in excess of 1 Gbps with an automatically-generated parser.

WMF. Vulnerabilities are increasingly being found in file formats (so called "data-driven attacks") rather than just network messages. The WMF format allows specification of a binary "abort procedure," called if the rendering engine is interrupted. Attackers began to misuse this feature in late 2005, using the abort handler for "drive-by downloads," where an attacker could run arbitrary code on a victim's computer by simply convincing them to render a WMF, requiring only a website visit for clients using Internet Explorer (CVE-2005-4560 [10]).

This vulnerability has been problematic for intrusion detection systems, Snort in particular. Snort normally processes only the first few hundred bytes of a message when looking for vulnerabilities; however, a WMF vulnerability can be placed at the end of a very large media file. However, matching the Snort rule set over an entire message exhausts the resources of most intrusion detection systems, requiring most sites to resort to a convoluted configuration with two Snort processes running in concert. Our architecture allows for a much cleaner approach: after an HTTP header has been parsed, the WMF vulnerability matcher would be called in the body handler, while other string matchers and handlers would be turned off. Figure 3 shows that WMF files can be parsed at multi-gigabit rates, so this would not put a significant strain on the CPU resources of the NIDS.

7 Future Directions

Although our prototype shows that high-performance vulnerability signature matching is possible in software, to achieve speeds in excess of 1 Gbps for all protocols, a hardware-accelerated approach is likely needed. Our plan is to use hardware implementations of fast pattern-matching algorithms [14,21] to replace the software implementations. This should dramatically increase the performance of text protocol parsing, as discussed in Section 6.1. We will also investigate the use of network processors, such as the Intel IXP family [22], to bring vulnerability processing closer to the network interface, and to exploit the inherent parallelism in matching signatures. Previous work has shown that using network processors can be nearly two orders of magnitude faster than similar implementations in software [23]. Network processors achieve such speedups in part by using a complex memory hierarchy; our careful management of limited state makes our architecture well-adapted to being ported to a network processor.

There are also performance gains yet to be realized in software matching as well. Our hand-coded matchers for vulnerabilities in binary protocols, in particular, are significantly fasters than those implemented using VESPA (see Table 1). The extra performance is likely due to eliminating the abstractions that ensue from representing a binary protocol structure in binpac. Our future work includes faster implementation of those abstractions, as well as the design of abstractions better suited to fast matching. One challenge that we will face is the fact that binary protocols exhibit much less consistency of design than text protocols.

Our eventual goal is to create a network intrusion *prevention* system (NIPS), which will sit as a "bump in the wire" and filter attacking traffic. In addition to throughput, another challenge that a NIPS will face is reducing latency, since, unlike intrusion detection systems, filtering decisions must be complete before the traffic can be forwarded to its destination. Furthermore, a NIPS must be able to recognize a large collection of vulnerability signatures at once. Our use of multi-pattern search as a base primitive will make parallel matching of several signatures easier to implement, but our design will need to incorporate constructs that will allow the reuse of common components (e.g., HTTP `Content-Length` extraction) between multiple signatures.

Authoring of effective signatures is a complex and error-prone process; this is true for exploit signatures, and more so for vulnerability signatures. Although our architecture was optimized for performance, rather than ease of authorship, we have found that expressing vulnerability constraints using VESPA was not appreciably more difficult than using binpac or GAPA. However, as we gain more experience with VESPA, we plan to improve the interface between the programmer and our architecture by, for example, introducing more reusable constructs and modularity. We also plan to develop better architectures for testing vulnerability signatures, to ensure that they do not generate false positives or false negatives.

Finally, automatic generation of vulnerability signatures can make them useful for not only known vulnerabilities, but new ones just observed ("zero-day"). Previous work has used annotated protocol structure [24,19], program analysis [9,25], or data flow analysis [26] to automatically generate vulnerability signatures. We will explore to what extent these approaches may be used to automatically generate signatures in our architecture. This will present a significant challenge to an automated approach, given that our architecture relegates more of state management to the programmer.

8 Related Work

8.1 Pattern Matching

The Wu–Manber [12], Boyer–Moore [27], and Aho–Corasick [11] algorithms provide fast searching for multiple strings. Their superior performance has made them natural candidates for IDS pattern-matching; in addition to our system, Snort [17] uses Aho–Corasick to match static strings.

Although slower than string matching, regular expression-based matching provides considerably more expressive power. Regular-expression matching is well-studied in the literature; broadly, deterministic matching (e.g., flex [28]) offers linear time but exponential space complexity, while nondeterministic matching (e.g., pcre [29]) offers linear space but exponential time complexity. Smith et al. attempt to combine the advantages of deterministic and nondeterministic matching using Extended Finite Automata [30]. Rubin et al. have developed protomatching to heuristically reduce matching complexity by discarding non-matching packets as quickly as possible, while keeping a low memory footprint [31]. Special-purpose hardware achieves sustained pattern matching at 4 Gbps [14]. Clark et al. [13] used application-specific FPGA cores to exploit the parallelism inherent in searching for many patterns simultaneously in a single body of text.

8.2 Vulnerability Signatures

The Shield project at Microsoft Research [2] pioneered the idea of vulnerability signatures; Borisov et al. extended the idea with a generic protocol parser generator [7]. Brumley et al. explained the complexity of various approaches to matching [9].

The binpac project at UC Berkeley and the International Computer Science Institute [1] focused on implementing a yacc-like tool for generating efficient protocol parsers from high-level definitions. binpac abstracts away much error-inducing complexity (e.g., network byte ordering). Its performance for many protocols is adequate for many intrusion detection tasks, but the VESPA architecture significantly improves on it, as shown in our evaluation.

The ongoing NetShield project [32] shares our goals of high-speed vulnerability signature detection. It has resulted in novel techniques for fast binary traversal, as well as efficient multi-signature matching, which may provide promising approaches for addressing some of the same challenges in VESPA.

8.3 Intrusion Detection

Intrusion detection requires attention to both algorithmic efficiency, and systems / implementation issues. Ptacek and Newsham [33] have detailed several strategies for evading intrusion detection by shifting packet TTLs, among others. Snort [34,17] and Bro [8], two popular IDS platforms, have addressed many systems-level issues, but are intended only to detect, not prevent intrusion. So-called intrusion prevention systems go further, by being deployed inline with the forwarding path; these systems take a more active stance against hostile traffic by dropping malicious or otherwise anomalous packets. The SafeCard [35] project used an Intel IXP network processor to perform intrusion protection in real-time up to 1 Gbps. It used high-speed matching of regular expressions, as well as an early implementation of Prospector [26] signatures, finding vulnerabilities within HTTP headers. The project shows that special-purpose hardware is a promising direction for high-performance intrusion prevention systems.

9 Conclusion

We have proposed an architecture, called VESPA, for fast matching of vulnerability signatures. VESPA relies on the fact that full protocol parsing is often not necessary to match vulnerability signatures and as a result is able to match signatures several times faster than existing work. We have built a prototype implementation of our architecture, and we showed that we can match vulnerabilities in many protocols at speeds in excess of 1 Gbps, thus demonstrating that vulnerability signatures are practical for high-performance network intrusion detection systems. We plan to continue to improve the performance of our system by improved implementation of base primitives and hardware acceleration, and to develop a full-fledged implementation of a high-performance network intrusion prevention system based on vulnerability signatures.

Acknowledgments

We would like to thank David Nicol and William Sanders for their guidance on this project and the anonymous referees for their suggestions on an earlier version of this draft. This work was supported by NSF grant CNS 06–27671.

References

1. Pang, R., Paxson, V., Sommer, R., Peterson, L.: binpac: A yacc for Writing Application Protocol Parsers. In: Proceedings of the Internet Measurement Conference (2006)
2. Wang, H.J., Guo, C., Simon, D.R., Zugenmaier, A.: Shield: Vulnerability-Driven Network Filters for Preventing Known Vulnerability Exploits. In: ACM SIGCOMM Computer Communications Review (2004)
3. CERT: “Code Red” Worm Exploiting Buffer Overflow in IIS Indexing Service DLL. CERT Advisory CA-2001-19 (July 2001), www.cert.org/advisories/CA-2001-19.html
4. Friedl, S.: Analysis of the New “Code Red II” Variant (August 2001), http://www.unixwiz.net/techtips/CodeRedII.html

5. Microsoft: Unchecked Buffer in ISAPI Extension Could Enable Compromise of IIS 5.0 Server. Microsoft Security Bulletin MS01-033 (June 2001), www.microsoft.com/technet/security/bulletin/ms01-023.mspx
6. Rescorla, E.: Security Holes... Who Cares?. In: Paxson, V. (ed.) USENIX Security Symposium (August 2003)
7. Borisov, N., Brumley, D.J., Wang, H.J., Dunagan, J., Joshi, P., Guo, C.: A Generic Application-Level Protocol Parser Analyzer and its Language. In: Proceedings of the 14th Annual Network and Distributed System Security Symposium (2007)
8. Paxson, V.: Bro: A System for Detecting Network Intruders in Real-time. Comput. Netw. 31(23-24), 2435–2463 (1999)
9. Brumley, D., Newsome, J., Song, D., Wang, H., Jha, S.: Towards Automatic Generation of Vulnerability-Based Signatures. In: Proceedings of the 2006 IEEE Symposium on Security and Privacy (2006)
10. CVE: Common Vulnerabilities and Exposures, http://cve.mitre.org/
11. Aho, A.V., Corasick, M.J.: Efficient String Matching: an Aid to Bibliographic Search. Commun. ACM 18(6), 333–340 (1975)
12. Wu, S., Manber, U.: A Fast Algorithm for Multi-Pattern Searching. Technical Report TR-94-17, Department of Computer Science, University of Arizona (1994)
13. Clark, C., Lee, W., Schimmel, D., Contis, D., Koné, M., Thomas, A.: A Hardware Platform for Network Intrusion Detection and Prevention. In: Proceedings of the Third Workshop on Network Processors and Applications (2004)
14. Brodie, B.C., Taylor, D.E., Cytron, R.K.: A Scalable Architecture For High-Throughput Regular-Expression Pattern Matching. In: ISCA, pp. 191–202 (2006)
15. Dreger, H., Feldmann, A., Mai, M., Paxson, V., Sommer, R.: Dynamic Application-layer Protocol Analysis for Network Intrusion Detection. In: USENIX-SS 2006: Proceedings of the 15th conference on USENIX Security Symposium, Berkeley, CA, USA, p. 18. USENIX Association (2006)
16. Dominus, M.J.: Higher Order Perl: Transforming Programs with Programs. Morgan Kaufmann, San Francisco (2005)
17. Sourcefire, Inc.: Snort, www.snort.org
18. Watson, B.W., Cleophas, L.: SPARE Parts: a C++ Toolkit for String Pattern Recognition. Softw. Pract. Exper. 34(7), 697–710 (2004)
19. Cui, W., Peinado, M., Wang, H.J., Locasto, M.E.: ShieldGen: Automatic Data Patch Generation for Unknown Vulnerabilities with Informed Probing. In: Pfitzmann, B., McDaniel, P. (eds.) IEEE Symposium on Security and Privacy, May 2007, pp. 252–266 (2007)
20. NISCC: Vulnerability Advisory 589088/NISCC/DNS (May 2005), http://www.cpni.gov.uk/docs/re-20050524-00432.pdf
21. Clark, C.R., Schimmel, D.E.: Scalable Pattern Matching for High-Speed Networks. In: IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM), Napa, California, pp. 249–257 (2004)
22. Intel: Intel Network Processors, www.intel.com/design/network/products/npfamily/index.htm
23. Turner, J.S., Crowley, P., DeHart, J., Freestone, A., Heller, B., Kuhns, F., Kumar, S., Lockwood, J., Lu, J., Wilson, M., Wiseman, C., Zar, D.: Supercharging PlanetLab: A High Performance, Multi-application, Overlay Network Platform. SIGCOMM Computing Communications Review 37(4), 85–96 (2007)
24. Liang, Z., Sekar, R.: Fast and Automated Generation of Attack Signatures: A Basis for Building Self-protecting Servers. In: Meadows, C. (ed.) ACM Conference on Computer and Communications Security, November 2005, pp. 213–222. ACM, New York (2005)

25. Brumley, D., Wang, H., Jha, S., Song, D.: Creating Vulnerability Signatures Using Weakest Pre-conditions. In: Proceedings of the 2007 Computer Security Foundations Symposium, Venice, Italy (July 2007)
26. Slowinska, A., Bos, H.: The Age of Data: Pinpointing Guilty Bytes in Polymorphic Buffer Overflows on Heap or Stack. In: Samarati, P., Payne, C. (eds.) Annual Computer Security Applications Conference (December 2007)
27. Boyer, R.S., Moore, J.S.: A Fast String Searching Algorithm. Commun. ACM 20(10), 762–772 (1977)
28. Flex: The Fast Lexical Analyzer, http://www.gnu.org/software/flex
29. PCRE: Perl Compatible Regular Expression Library, http://www.pcre.org
30. Smith, R., Estan, C., Jha, S.: XFA: Faster Signature Matching with Extended Automata. In: Proceedings of the 2008 IEEE Symposium on Security and Privacy (2008)
31. Rubin, S., Jha, S., Miller, B.P.: Protomatching Network Traffic for High Throughput Network Intrusion Detection. In: Proceedings of the 13th ACM conference on Computer and communications security (2006)
32. Li, Z., Xia, G., Tang, Y., He, Y., Chen, Y., Liu, B., West, J., Spadaro, J.: NetShield: Matching with a Large Vulnerability Signature Ruleset for High Performance Network Defense (manuscript) (2008)
33. Ptacek, T.H., Newsham, T.N.: Insertion, Evasion, and Denial of Service: Eluding Network Intrusion Detection. Technical report, Secure Networks, Inc., Suite 330, 1201 5th Street S.W, Calgary, Alberta, Canada, T2R-0Y6 (1998)
34. Roesch, M.: Snort—Lightweight Intrusion Detection for Networks. In: Parter, D. (ed.) Proceedings of the 1999 USENIX LISA Systems Administration Conference, Berkeley, CA, USA, November 1999, pp. 229–238. USENIX Association (1999)
35. de Bruijn, W., Slowinska, A., van Reeuwijk, K., Hruby, T., Xu, L., Bos, H.: SafeCard: A Gigabit IPS on the Network Card. In: Proceedings of the 9th International Symposium On Recent Advances in Intrusion Detection (2006)

Swarm Attacks against Network-Level Emulation/Analysis

Simon P. Chung and Aloysius K. Mok[*]

Department of Computer Sciences,
University of Texas at Austin, Austin TX 78712, USA
{phchung,mok}@cs.utexas.edu

Abstract. It is always assumed that if the attackers can achieve their goal by exploiting a vulnerability once, they won't exploit it twice. This assumption shapes our view of what attacks look like, and affects the design of many security systems. In this work, we propose the swarm attack, in which the attacker deliberately exploits the same vulnerability multiple times, each intended to carry out only a small part of the attack goal. We have studied eight systems that detect attacks using network-level emulation/analysis, and find them surprisingly vulnerable to attacks based on this strategy.

Keywords: Decoder detection; network-level emulation; network IDS; evasion; swarm attacks.

1 Introduction

In its simplest, most common form, a control hijacking attack works as follow: the attacker sends in **one single malicious input** with the proper "protocol frame" to trigger the targeted vulnerability, together with **a self contained payload** that will achieve the attacker's goal once executed. When the malicious input is processed, certain control data structure will be overwritten, and this results in **an almost instant transfer of control** to the attacker's payload. We believe many security systems are designed with this simple model of attacks in mind, and it is usually implicitly assumed that the attacker gains nothing by making the attack more complicated (or less "efficient"). In other word, if they can get all their attack code executed with one instance of control hijacking, they will not divide their code into multiple pieces and execute them through multiple exploitations of the vulnerability. Similarly, the attacker will overwrite the piece of control data that leads to the control hijacking with the minimum delay.

In this paper, we propose the attack strategy where the attacker violates the above assumption and be deliberately "inefficient" in their attacks, and study the implications of such strategy to systems that try to locate executable code within network traffic and determine if those are attack payload. We call our proposed attack **the swarm attack**, and will refer to target systems described

[*] The research reported here is supported partially by a grant from the Office of Naval Research under contract number N00014-03-1-0705.

above **network-level emulation/analysis systems**. Surprisingly, we find that by deliberately dividing the attack code into many pieces and have each executed through a different exploitation of the same vulnerability, the attacker can evade at least seven out of the eight network-level emulation/analysis systems that we have studied [1,3,13,14,15,21,24,25] (we believe the third one, [13], may detect our attack if specifically trained to, but can only do so at the cost of high false positives). The design of our attack is simple; the attack will be divided into n+1 instances of control hijacking. Each of the first n instances will have a small payload to write part of the real decoder to a predetermined area in the attacked process' address space, and the $(n+1)^{st}$ instance will direct the hijacked control to the decoder we just constructed. Under this attack, the number of unencoded instructions in each attack instance can be reduced to below 10, and all these unencoded payload will appear to serve no useful purpose for an attack. Note that the need to have multiple instances of control hijacking on the target system places certain constraints on our swarm attack. However, we will argue in Sect. 4.1 that the attack can be used against many vulnerable network servers, and there are techniques to overcome this constraint even if the target system is single-threaded. Finally, we believe if network-level emulation/analysis systems continue to consider traffic separately, such small, simple payload will be very hard to detect with low false positives; the payload behavior is so simple that the chance of finding such behavior in random data by coincidence is non-negligible.

The rest of the paper will be organized as follow: in Sect. 2, we will present related work in the area of network-level emulation/analysis based detection, and attacks against other types of network intrusion detection systems. In Sect. 3, we will present the details of the proposed swarm attack, and address some practical issues that may arise in the implementation of the attack in Sect. 4. Analysis of how the proposed attack evade network-level emulation/analysis will be given in Sect. 5, and in Sect. 6, we will discuss whether it is possible to improve existing systems to detect the attack. Finally, we conclude in Sect. 7.

2 Related Work

There are generally three approaches for network intrusion detection, and the most traditional of which is signature matching. The second approach is anomaly detection, which compares properties of observed traffic against properties of known good traffic. Network anomaly detection systems usually treat the traffic under analysis as a bag of bytes, and use statistical methods to determine if this bag of bytes appears to be an attack. For example, PAYL [23] distinguishes normal traffic from attacks based on byte frequency distribution. The last approach, which we call the network-level emulation/analysis, is the focus of our work. The main idea behind this approach is to locate executable code within the incoming traffic, and analyze the extracted code to determine if it is random data that coincidentally appears to be syntactically correct machine instructions or actual attack code. We note that any useful attack strategy must be able to defeat all three kinds of detectors. However, in this work, we will focus on evading systems based on network-level emulation/analysis, which is the least attacked among the three approaches. As for the evasion of the other two mechanisms, we will rely on existing techniques against them.

2.1 Analyzing Code within Network Traffic

The earliest network-level emulation/analysis systems are designed specifically for buffer overflow attacks. In particular, they are designed to detect the sled in these attacks; since the attacker does not know the exact address where their payload will be found, the hijacked control is usually directed to an area filled with NOPs that precedes the actual payload. This technique allows the attack to succeed even though the attackers only have a rough estimate of where their payload will be located on the stack or the heap, and the area of NOP is called the sled. Sled-detection systems are usually very simple. For example, [21] scans through the incoming traffic and declares it as malicious if it finds 30 or more consecutive valid instructions in the traffic. Similarly, [1] considers the incoming traffic malicious if it contains an instruction sequence that spans at least 230 bytes, with each of its suffix also being a valid instruction sequence.

The obvious problem with sled-detection is that not all attacks contain sleds. In fact, with the use of register springs, many buffer overflow attacks can avoid using sled. Thus a second generation of detection systems is developed to identify "meaningful" code within network traffic. For example, [24] will classify incoming traffic as malicious if: (1) it contains two or more separate instruction sequences for function calls (including the instructions for placing arguments on the stack and the actual control transfer), or (2) it contains a sequence of 14 or more "useful" instructions that does not cause any data flow anomaly (i.e. they define data values once before use). As another example, [3] defines malicious traffic as one that contains either obvious library/system calls (identified by hardcoded jump target and interrupt instructions after initializing eax), return/indirect control transfer with target address being properly set up by preceding instructions found in the traffic, or a proper loop structure that appears to be a decoding routine in polymorphic shellcode. The weakness of this second generation of systems is that they are not very effective against polymorphic shellcode, in which only the decoder appears as valid instructions in the network traffic, and the rest of the attack code is encoded. To address this problem, systems are designed to target properties specific to the decoding routines of polymorphic shellcode. The most commonly used property is the presence of GetPC code, which allows a position-independent shellcode to find out its own location, as well as the address of the payload to be decoded. In both [14,25], the presence of GetPC code (e.g. call, fnstenv) is used both as a precondition for further analysis, and an indicator of the beginning of the decoding routine. With this location of the GetPC code, [25] confirms that the identified code is indeed a decoder if it is self modifying and involves a loop which contains indirect write instructions with target addresses that are different in each iteration. On the other hand, after identifying the GetPC code, [14] characterizes the decoder by a significant number (6 or more) of reads from different locations within the analyzed traffic itself. A machine learning based approach is used in [13], where a neural network is employed to determine if a sequence of instructions is a decoder, based on the frequency at which different types of instructions appears in that sequence. Even though [13] shows that neural network trained with decoder from one polymorphic engine can identify decoder routines from another polymorphic engine, we believe retraining is necessary if there is a drastic change in the decoding algorithm. Finally, as an extension of [14], [15] argued that some non-self-contained polymorphic shellcode does not have any GetPC code or reads to bytes within the traffic itself. [15] thus proposed two new properties for identifying polymorphic shellcode: writing to a significant number of different memory locations, and then executing those locations that has been written to.

2.2 Evading Signature-Based and Statistics-Based Detectors

Polymorphic shellcode, which is the focus of many systems described in the previous section, is originally designed to evade signature-based defenses. The idea is simple, to avoid being matched by signatures generated based on previous instances of the same attack, the attacker will make every attack instance appears differently. This goal is usually achieved by having the attack code encoded by some very simple "keyed-encryption" algorithm, and has the code for each attack instance encoded under a different key. In order to allow correct execution of the attack code, we need to attach a decoder to each attack instance, provide it with the correct key and execute it to decode the real payload. This way, only the decoding routine will remain constant throughout all attack instances. To avoid the decoder from being targeted by signature-matching, various "polymorphic"[1] engines have been developed to make the routine slightly different in every attack instance. Common techniques for achieving this goal include instruction substitution [6,10] and insertion of junk instructions [10].

Even though the encoding of the actual payload, together with the metamorphism applied on the decoder will successfully evade a signature-based detector, the resulting attack instances may still have very different properties from normal traffic, and thus can be detected by some kind of anomaly detection. In [8], a technique is proposed to encode the payload so that it will have the same byte frequency distribution as the observed normal traffic, and evade anomaly detection systems based on byte frequency (e.g. PAYL [23]). The idea is extended in [7] so that encoded payload (using either xor-based or byte-substitution-based encoding) which satisfies any normal traffic profile (expressed in a finite-state machine) can be found.

The difficulties of detecting the decoder of a polymorphic attack with either a signature-based or statistical-based approach are also demonstrated in [20], but in an unconventional way. Instead of showing concrete ways to defeat the studied defenses, [20] only presents an "existential proof", showing that n-byte sequences that exhibit decoder-like behavior are distributed over a very large

[1] Which are actually "metamorphic" engines.

span of all possible n-byte sequences, and uses this as an evidence to suggest the actual decoder population may have a similar span, and thus it will be very difficult to characterize all of them with signatures or statistical model. What is of interest are the properties [20] used to define decoder behavior: self-writing (containing instructions that write to nearby locations) and self-modification (containing instructions that write to nearby locations using values read from another nearby location). This further illustrates the general perception of what decoders should look like, and can be very useful when we design our attack to evade systems that detect instruction sequences which appear to be decoding routines.

2.3 Other Related Attacks

In general, attacks for evading data-non-executable defenses can achieve the same goal as ours; they carry out the attack without executing (or placing) any code within the network traffic, thus there will be nothing for network-level emulation/analysis systems to detect. However, these attacks are usually much more difficult to construct than those that use highly obfuscated/polymorphed shellcode. For example, [18] makes extensive use of the ret-to-libc technique, and allows the attacker to "execute" arbitrary code by chaining up "gadgets", each being code fragment within libc which contains instructions for achieving some primitive binary-level operations (e.g. data movement), followed by a return instruction that will pass the control to the next gadget. However, there seems no easy way to automatically locate all the gadgets needed for some set of primitive operations, and these gadgets can only be invoked by using hard-coded addresses, which may harm the portability of the resulting attack, and can provide a lot of materials for signature matching. As for the non-control-data attack in [2], the logic of the attacked program is altered through the manipulation of its critical data, and such attacks cannot be designed without intimate knowledge of the internals of the victim program, as well as the whereabouts of its critical data. Once again, it is unclear to us whether [2] can be effective against signature-based or statistics-based defenses. Furthermore, standard techniques (like [6,8,10]) for evading detection, that mostly focus on code morphing/encoding, are not applicable to attacks against data-non-executable, since they don't involve any code at all. Finally, an attack of similar flavor but different objective to ours is [17], where a technique for evading signature-based detection systems is presented. The idea in [17] is similar to ours in the sense that evasion is achieved through breaking up the attack into many small pieces, and inserting some useless pieces in between (though the attacks generated by [17] still exploit the target vulnerability only once).

To conclude our discussion of related work, we note that techniques for evading the three types of detection systems can be easily combined; while the technique in [7] only works for certain types of encryption/decryption routine, both the technique we are going to present and the metamorphism employed in [6,10] can work on any kind of decoders. Thus, [7] will determine the decryption routine we can use, and provide an encrypted payload that can blend in with normal traffic, our swarm attack will modify the routine to remove any behavior expected of a decoder (or any non-polymorphic malicious code), and the metamorphism will be applied to the modified routine so that it appears differently in every attack instance.

3 Swarm Attack against Network-level Emulation/Analysis

As we have mentioned in the introduction, the idea of swarm attack against network-level emulation/analysis systems is to modify a control hijacking attack so that the decoder in its polymorphic shellcode will not appear in any attack traffic. We achieve this goal by creating the decoder inside the attacked process' address space using multiple instances of the attack, with each attack instance writing a small part of the decoder at the designated location. When we have finished building the decoder, we will send in one last attack instance which serves two purposes; first of all, it will hijack the control of the attacked process to start executing the decoder, and secondly, it will carry the encoded actual payload.

Note that the decoder under this swarm attack will have to be modified to locate the actual payload (which may not be found using the same method as in the original exploit where both the decoder and the encoded payload appear in the same attack traffic). However, this is not a serious difficulty; we can construct our last attack instance by modifying the original self-contained exploit so that the encoded payload is placed at where the decoder will have appeared in the original case. As such, the decoder can locate the payload based on how we direct the hijacked control to the right location in our original attack. If a hardcoded address is used in the original exploit, the decoder in the swarm attack will locate the payload using this same hardcoded value. If the original attack used a register spring, the address of the payload will be found in the register involved (remember that the last attack instance is constructed from the original attack by replacing the decoder with the encoded payload; if the register points to the beginning of the decoder in the old attack, it will point to the encoded payload in the new one).

Now let's consider the design of the attack instances responsible for building the decoder. If the vulnerability exploited allows writing arbitrary value to arbitrary address (e.g. a format string vulnerability), our task is trivial: we only have to build exploits to write the right value to the right place. Also, in this case, we can avoid putting any executable code into traffic generated, and it would be quite impossible for a detector based on network-level emulation/analysis to identify this attack. However, care must be taken to have some of the attack instances write to slightly overlapping addresses; otherwise, the attack instances responsible for building up the decoder may become easy target for signature matching. For example, suppose the vulnerability allows us to overwrite 4 bytes at a time, and the first four bytes of the decoder we are building are $b_1b_2b_3b_4$; if we build the decoder by always writing to non-overlapping bytes, we will always have an attack instance that contains the bytes $b_1b_2b_3b_4$. To avoid this problem, we can have one attack instance writing $b_1 r_1 r_2 r_3$ to address i, and the next instance writing $b_2 r_4 r_5 r_6$ to address i+1, so on so forth. Since we know $r_1 r_2 r_3$ will be overwritten by the second attack instance, we can put random values there. Of course, the byte b_1 will still appear in the first attack instance of every swarm attack that employs the same decoder, but this property that involves only one byte will not be very useful to the defender.

If the exploited vulnerability only allows direct control hijacking (e.g. stack based buffer overflow), the design of the attack instances which build up the decoder is much more interesting. In this case, we will need to put some executable code into each attack instance, and have each instance hijack the control to execute its attached code and write the correct value to the right address. As opposed to the previous case, the attack traffic will now contain some executable code. In order to evade detection by network-level emulation/analysis systems, we need to craft the code visible to these systems carefully. Nonetheless, we note that the task to be performed by this code snippet is very simple, and should not involve much behavior that is typically considered "decoder-like" (e.g. no GetPC or self-modification, minimal read/write). Thus the design should be quite easy. We have also taken care to have a design that is easily polymorphed, and does not have long sequence of bytes that remains constant over different attack instances, or always appears in an instance responsible for writing a particular part of the decoder. This precludes using the bytes we want to write as immediate operands or reading it directly from the attack traffic; i.e. we have to somehow "generate" what we are writing as a product of executing some instructions, and we used the xor operation for this purpose. We note that this design also allows us to use decoder that contains bytes forbidden for successful exploitation of the vulnerability (e.g. the presence of byte 0x00 is not allowed in many exploits). Similar constraints apply to the target of the write operations, and the same approach can be used for "generating" it in our attack. The code we have designed for building the decoder is given in Fig. 1.

As we can see on the left of Fig 1, we assume the initial value of ebp is under our control, which is true for almost all stack-based buffer overflows. Also, as shown in the right part of Fig. 1, by using some very simple metamorphism (replacing registers, using slightly different instructions and randomizing exxOffset, exxMask, ebpMask and ebpOffset) , we can achieve such degree of polymorphism that no two instances of the code we have for building the decoder will share any common byte sequence that is more than one byte long. Further polymorphism/metamorphism is possible by re-ordering some of the instructions, or inserting junk instructions. Finally, note that the last instruction in our code snippet will put the execution into a dead-loop. This is only necessary when we cannot crash the attacked thread without killing the entire process. In case we are attacking a serve-type process that handles thread failure gracefully, we can simply put some junk bytes after the instructions that write the value to the right location. This way, the code snippet will look even more innocuous to network-level emulation/analysis systems, since they all assume the attacker will not crash the target.

```
sub ebp, ebpMask                \x81\xed ebpMask (6 bytes)
mov ecx, [esp+ecxOffset]        \x8b\x4c\x24 ecxOffset (4 bytes)
xor ecx, ecxMask                \x81\xf1 ecxMask (5 bytes)
mov [ebp+ebpOffset], ecx        \x89\x4d ebpOffset (3 bytes)
jmp -2                          \xeb\xfe (2 bytes)

add ebp, ebpMask                \x81\xc5 ebpMask (6 bytes)
mov ebx, [esp+ebxOffset]        \x8b\x5c\x24 ecxOffset (4 bytes)
xor ebx, ebxMask                \x81\xf3 ecxMask (5 bytes)
mov [ebp+ebpOffset], ebx        \x89\x5d ebpOffset (3 bytes)
jnz -2                          \x75\xfe (2 bytes)
```

Fig. 1. Two possible versions of the attack code for building the decoder. All ebpMask, exxOffset, exxMask and ebpOffset are variable. The binary representation of the code are given on the right, with bytes that remain the same despite the use of different registers/operations highlighted. Note that condition used in the "jnz -2" is set by the xor. Since we know the result of that xor operation, we can choose the right kind of conditional branch, and there are many different condition codes that we can use in this branch instruction.

We have tested our swarm attack by modifying an exploit against a stack based buffer overflow in the Savant web server [11]. In our experiments, we used as our decoder a simple 27-byte routine which xor each DWORD of the encoded payload with a fixed key, and this requires 7 attack instances to build up the decoder, and one last instance to execute it. As for the real payload, we used a 198 byte shellcode that starts notepad.exe. More complicated shellcode are easily accommodated, we only choose this one for its very visible result (which makes it easy to determine that the attack is successful). As of the location of the decoder, we choose to build it at the end of Savant's data area. This makes our attack quite portable across machines running different versions of Windows, as long as Savant is loaded at the same place. However, since the address of this data area starts with 0x00, we cannot use a hardcoded address in the last attack instance to jump to the decoder. Instead, we execute a small (2-instruction, 8-byte long) code snippet in this last attack instance to "generate" the address of the decoder in some register (the same way we "generate" the target address for the write in the instances responsible for building the decoder) and jump to this address using a register indirect control transfer. By transferring the hijacked control to the decoder using a small, easily poly/meta-morphed payload in the last attack instance, we can also avoid the hardcoded address for the decoder from appearing in every swarm attack and being used as a signature. We believe this "trampoline" payload in the last attack instance is necessary if we cannot have too much variation in the location where we place the decoder. Finally, we report that all our experiments successfully lead to the execution of the decoded payload and launch notepad as expected.

4 Practical Concerns

In this section, we will address some possible difficulties that may arise during the implementation of the swarm attack. Our main focus is, given an exploit that allows us to execute arbitrary code on the attacked machine, what are the extra problems that we will have to face in order to build a swarm attack based on this exploit?

4.1 Multiple Exploitations

The biggest constraint in implementing a swarm attack is that we need a vulnerability that can be exploited multiple times, with the effect of each exploitation being persistent and visible to all later exploitations until the actual payload execution starts. The above constraint is automatically satisfied if the target is a multi-threaded program that will continue to function (i.e. accept further traffic/input) while under attack, and we note that many network servers have this nice property. We believe even the simplest of such servers will be multi-threaded (especially true under the Windows environment), and it is very likely that the port concerned will be freed to accept further traffic once the processing of the incoming request starts. In case we are attacking a single-threaded program (or one with only one thread performing the vulnerable processing), swarm attacks are still possible if:

1. the vulnerable program processes multiple inputs that may cause the control hijacking
2. we can have a way to continue the normal processing in the attacked process after accomplishing the current step in the decoder construction process.

Since we believe the first of the above conditions will be satisfied by many programs (and there is very little we can do otherwise), we will focus on ways to restore normal processing of the target program after each attack instance in our swarm attack. Though it first appears very complicated, we find this task quite achievable for the most common types of exploits.

If the targeted vulnerability is a stack buffer overflow, techniques similar to the "error-virtualization" in [19] can be applied to "return to normal execution" after an attack instance has accomplished its goal. The idea is to prevent the attacked process from crashing by rolling the execution forward to the point where some function x higher up in the "call tree" than the vulnerable function returns a value that signifies failure/error, with the caller of function x equipped to handle the error. This technique should be feasible in many cases because the attackers usually have very accurate knowledge of the size of the few activation records on the top of the stack when the injected code starts execution, and thus can properly adjust the stack for the return. Also note that such "recovery" from the attack can be achieved with very few instructions; it only involves an addition to esp, a single write (if we need to fake a return value) and a return. Furthermore, since the return address used does not come with the attack traffic, most network-level emulation/analysis systems will ignore the return (e.g. [3]), considering it to have too high a chance of crashing the process to be any part of a robust attack. The only system that may find this recovery suspicious is [24], which specifically looks for "push-call" patterns. However, two such patterns are needed to trigger an alert in [24], and we will only have one in our attacks. Another very favorable scenario appears when the control hijacking occurs within code that handles exceptions; in this case, the attacker simply executes an invalid instruction, and the attacked program will return to normal execution. Unlikely as it may sound, there is indeed one real life example of this favorable situation: the ANI vulnerability in Windows XP/Vista [16].

Format string vulnerabilities are also very suitable for a swarm attack: their exploitations generally do not contaminate any of the target program's data structures "by accident"; once the vulnerable function has finished processing the malicious format string, it will return properly. Thus the target program can usually carry on with its normal execution after every attack instance in a swarm attack built on top of a format string vulnerability (e.g. we have confirmed that it is possible to exploit the format string vulnerability in the wu-ftpd server [22] multiple times, through a single connection to the server). Finally, we admit that the feasibility of a swarm attack is more questionable in the case of a heap buffer overflow; in many cases, the corruption in the heap will crash the attacked program shortly. However, the technique in [9] may be improving the situation.

4.2 Where to Put the Decoder?

Another difficulty that we may face when implementing a swarm attack is that we need to find an area in the attacked process' address space that: (1) will be reliably writable in every instance of the program, even if it's running on different OSs, using different versions of libraries, and (2) will remain untouched until the decoding of the real payload is completed.

In most scenarios, the first condition can be easily satisfied given the original exploit for control hijacking. In particular, if the original exploit used a register spring to direct the hijacked control, we can easily derive an address within the data area of the module which holds the exploited register spring instruction (this is true even when some address space layout randomization is applied, if only the base of a module is randomized). In this case, we will argue that having to find a writable location to place the decoder does not make the swarm attack any more difficult to implement than the original. However, if the original exploit used a hardcoded address (which is less common nowadays), a different approach is needed. One solution is to use another hardcoded address (as we did in our experiments on Savant). Given our success in finding register springs that remain at the same address throughout various versions of OS/library, finding hardcoded addresses that are writable across different target machines should be very feasible. Another possibility is to see if any register is pointing to some global data area at the time of control hijacking.

As of storing the decoder in an area that will not be modified until we've finished building it, we note that since memory protection is applied at the granularity of a page, the last parts of many writable regions are never used (they do not correspond to any variable/data structure in the underlying program). Thus, the last part of all writable regions should be very good starting point in our search for places to hold the decoder, and we can always test the target program to "estimate" if it is safe to store the decoder at one of these candidate areas. Some other possible locations for persistent storage of injected code have been proposed in Sect. 3.1 of [12].

5 How Swarm Attacks Evade?

5.1 Sled-Detection Systems

To see how our swarm attack evades the sled-detection systems described in Sect. 2.1, we note that our attack against Savant used a register spring, and contains no sled. Thus the only executable code that a sled-detection system can find is that for building the decoder or transferring the control to the decoder.

Recall that [21] considers incoming traffic malicious if it contains a valid sequence of 30 instructions or more. For the swarm attack against Savant, the attack instances for building up the decoder consists of 6 instructions, with one to jump over junk bytes (not shown in Fig. 1 since it is specific to the attack against Savant), 4 for actually writing part of the decoder to the right place, and one that puts the execution in dead loop. As for the last attack instance that transfers control to the decoder we've built, it consists of one instruction to jump over junk bytes, one for setting up the target of the jump and one for the jump itself. Finally, note that [21] counts jumps targeting address outside the attack traffic as two instructions; any other jump instructions will be counted as one, and a jump targeting instructions that appear earlier in the instruction sequence will mark the end of that sequence. Obviously, [21] will not be able to discover any valid sequence that contains more than 6 instructions in all our attack instances, and thus will pass them all as benign. Similarly, [1] tries to locate the longest byte sequence in traffic such that any suffix of the sequence is a valid chain of instructions, and consider the traffic as malicious if the longest of such sequences found is 230 byte or longer. The longest sequence that [1] can find in our attack will be of 22 byte long (with 2 bytes for jumping over junk, and 20 bytes as shown in Fig. 1). Thus, the swarm attack will evade [1] also.

5.2 "Meaningful Code" Detection

As for systems that try to detect code that appears to serve some "meaningful" purpose in an attack, recall that [24] looks for push-call sequences and instructions that do not cause any data flow anomaly. All attack instances in our swarm attack contain no push-call sequence (there will be one if we try to continue with normal execution after the attack using the method described in Sect. 4.1), and contains at most 6 "useful" instructions. Since the number of useful instructions needed for [24] to sound the alarm is 10, [24] will not be able to detect our attack. As for [3], the detector only considers control transfers at the end of every chain of basic blocks it identifies with static analysis of the incoming traffic. As such, only the jump that forms a dead loop in the earlier attack instances and the jump to the decoder in the last instance will be used by [3] to determine whether the traffic is malicious. Since an empty loop is considered benign, and register indirect jumps are only malicious to [3] if they target instructions within the analyzed traffic, our attack will certainly evade [3].

5.3 Decoder Detection

As we've mentioned in Sect. 2.1, almost all network-level emulation/analysis systems designed to specifically detect decoders in polymorphic shellcode will only consider incoming traffic malicious if it contains some GetPC code. Since our attack does not contain any such code, it will evade all detection systems that use GetPC as a precondition for further analysis. For the sake of argument, even if some GetPC code is added to our attack instances, they still won't be sufficiently "decoder like" to be detected. For example, [25] requires a loop containing indirect writes for traffic to be classified as malicious, but the only loop in our attack instances is empty. As for [14], more than 6 reads to different locations within the analyzed traffic have to be found before it will be flagged as an attack, while our attack instances perform at most one read operation.

The successful evasion of [13] by our swarm attack is less certain. When presented traffic from our swarm attack, we believe [13] will successfully identify the code involved (either for building the decoder or for executing the decoder). Whether [13] can detect our attack will then be determined by its model for a shellcode decoder (i.e. the frequency at which different types of instructions appear in a decoder), and in the worst case, if [13] is trained to recognize code in our attack instances, it is quite likely that our attack will be detected. However, we can always polymorph our attack to introduce noise for the classification in [13] (i.e. introduce various types of instructions). Furthermore, it is questionable if [13] can maintain a low false positive if it's trained to recognize the small "decoder" in our attack (we will elaborate on this point in Sect. 6).

Finally, [15] used a negative heuristic that if the code recovered from the traffic contains fewer than 8 write instructions, it will be considered benign. Since all our attack instances contain only one write operation, they will all successfully bypass [15].

6 Can Network-Level Emulation/Analysis Detects Swarm Attacks?

In this section, we will try to answer the following question:

> Can network-level emulation/analysis systems be improved to detect the swarm attack we've proposed?

The answer to this question depends on our ability to characterize the kind of write operations that allows one to build the decoder, as well as the amount of false positives that will result from our best characterization of such "useful writes". We focus our discussion on characterizing the write operations used for building the decoder because it is the most essential feature of the visible payload in a swarm attack. The way of generating both the value to be written and the target address can be easily changed to evade detection, and as we will argue below, we maybe able to design swarm attacks in which these values are not "generated" by any instructions. As of the payload for the control transfer to the decoder in the last attack instance, we note that it may not be necessary in some swarm attacks. If we can build the decoder at many different places, we can have the last instance of attack direct the hijacked control to a hardcoded address (without executing any code), and still evade signature-based detection targeting that hardcoded value; due to the large number of choices we have for this address, any signature targeting a particular address will be useless. Even if it turns out that the executable payload in the last instance is unavoidable, we believe the difficulties in the characterization of this small payload, as well as the false positives resulting from detecting it, will be similar to that of the "useful write" discussed below.

Let's start our discussion by considering our attack against the Savant server once again. In the attack code in Fig. 1, the write operations involved have some very specific properties; in particular, both the value written and the address to write are direct products of previous instructions. However, we can easily avoid the dynamic generation of the former by using immediate values instead. This is especially true if we can afford to construct the decoder with more attack instances. For example, if we can double the number of attack instances used in building the decoder, we can specify the values to be written as immediate operands, and still leave no constant byte sequence for signature-based detection. This is because we can "write" each 4-byte of the decoder using two attack instances, the first writes an immediate value to the target location directly, and the second performs some arithmetic/bitwise operation between another immediate value and the previously written value, such that the result of the operation will be the right value for the decoder. This way, we can avoid having the values written in useful writes from being defined in previous instructions. To push the idea even further, if we can afford to build the decoder one byte at a time, and if we have control over an extra register when the control hijacking occurs, we can simply put the value to write in the register we control. As such, the value written will appear entirely undefined to the network-level emulation/analysis. When coupled with the overlapping-writes technique in Sect. 3, there will only be one byte that's constant across all attack instances responsible for building a particular part of the decoder.

If it turns out that there are so many locations in the attacked process' address space where the decoder can be safely built, we can avoid generating the target of the write operations used in building the decoder also; instead, we can include the immediate value of the write target in our attacks and still be able to evade any signature-matching by building the decoder at different place in different attacks. In fact, from our experience with attacking Savant, at least the least significant two bytes of the address where we place our decoder can show a high level of entropy, leaving only the most significant two bytes useful for signature matching (if we leave the write target unencoded in our attack instances). Thus, it is possible to design our attack such that the "useful writes" we use for building the decoder will appear to have both the value and the address written undefined to the network-level emulation/analysis.

It is also quite unlikely that we can keep the false positives of the network-level emulation/analysis low while we try to detect attack code as simple as those in the swarm attack. We based this pessimistic prediction on two pieces of data from [15]:

1. when tested against artificial binary data, the system in [15] found that 0.01% of the data writes to 8 unique addresses, and contain one control transfer to one of those written locations.
2. almost 1% of random binary/printable data will contain code that writes to a certain address and then an instruction that jumps to it.

We note that if we phrase a "useful write" in a swarm attack as "defining certain register and then use it as the target of a register indirect write", the behavior involved in the second item will be quite similar to a useful write in terms of the level of sophistication: both involve two related operations. Thus, it is not unreasonable to use the figure given to predict the level of false positives resulted from detecting traffic that contains one "useful write" operation.

7 Conclusions

In this paper, we have studied an attack strategy where the attacker deliberately makes his attack less efficient; instead of achieving their goal through one instance of control hijacking, they hijack the control of the target process multiple times, and achieve some minimal objective in each hijacking. Surprisingly, such swarm attack is very effective in evading detection systems that are based on network-level emulation/analysis, which is the least challenged approach for network intrusion detection. The swarm attack evades these systems by exposing a very small, simple piece of code in each attack instance, and slowly building up a decoder somewhere in the attacked process' memory using this minimal payload. Once the decoder is complete, one last instance of attack will be launched to carry the encoded payload and hijack the control to execute the decoder. We argue that since the exposed code in the swarm attack can be made so short and simple, it would be virtually impossible to detect such attack without incurring a high false positive. We have also noted that the need to hijack the control of the attacked process multiple times may constraint the type of vulnerabilities that can be exploited in a swarm attack, but we believe vulnerabilities in network servers are generally suitable for us, and there are techniques to exploit vulnerabilities that are less favorable to our swarm attacks.

As we have mentioned in the introduction, many security systems are built based on the assumption that the attackers will gain nothing by being inefficient and make their attacks "unnecessarily complicated". Thus, the swarm attack can have significant impact to other systems too. For example, a similar "multi-threaded" attack as the one we have presented can open up new avenue for mimicry attacks against system-call based IDS: in this new attack, not only can the attacker insert null calls into the sequence of system calls observed by the IDS, he can also issue system calls from another thread that's at a more favorable state. Our preliminary analysis also shows that some form of swarm attack can have significant impact on intrusion prevention systems which analyze information collected from the attacked host. In particular, with separation of the attack traffic that overwrites the targeted control structure and that hijacks the control using those contaminated structures, together with careful control of the time delay between the two parts of the attack, an attacker can make IPSs like Vigilante [5] vulnerable to allergy attacks [4], or force them into generating signatures/execution filters that are useless in stopping attacks. In our future work, we plan to further experiment with the swarm attacks against these two types of systems, and study their real impact.

References

1. Akritidis, P., Markatos, E.P., Polychronakis, M., Ananostakis, K.: Stride: Polymorphic sled detection through instruction sequence analysis. In: Proceedings of the 20th IFIP International Information Security Conference (IFIP/SEC 2005), Chiba, Japan (May 2005)
2. Chen, S., Xu, J., Sezer, E.C., Gauriar, P., Iyer, R.K.: Non-control data attacks are realistic threats. In: Proceedings of the 14th conference on USENIX Security Symposium (USENIX Security 2005), Madison (July 2005)
3. Chinchani, R., Van Den Berg, E.: A fast static analysis approach to detect exploit code inside network flows. In: Valdes, A., Zamboni, D. (eds.) RAID 2005. LNCS, vol. 3858, pp. 284–308. Springer, Heidelberg (2006)
4. Chung, S.P., Mok, A.K.: Allergy Attack Against Automatic Signature Generation. In: Zamboni, D., Krügel, C. (eds.) RAID 2006. LNCS, vol. 4219, pp. 61–80. Springer, Heidelberg (2006)
5. Costa, M., Crowcroft, J., Castro, M., Rowstron, A., Zhou, L., Zhang, L., Barham, P.: Vigilante: End-to-end containment of internet worms. In: Proceedings of 20th ACM Symposium on Operating Systems Principles, Brighton (October 2005)
6. Detristan, T., Ulenspiegel, T., Malcom, Y., von Underduk, M.S.: Polymorphic shellcode engine using spectrum analysis. In: Phrack, vol. 11 (2003)
7. Fogla, P., Lee, W.: Evading network anomaly detection systems: Formal reasoning and practical techniques. In: Proceedings of the 13th Conference on Computer and Communication Security (CCS 2006), Virginia (October 2006)
8. Fogla, P., Sharif, M., Perdisci, R., Kolesnikov, O., Lee, W.: Polymorphic blending attacks. In: Proceedings of 15th USENIX Security Symposium Abstract (USENIX Security 2006), Vancouver (July 2006)
9. jp. Advanced Doug lea's malloc exploits, `http://doc.bughunter.net/buffer-overflow/advanced-malloc-exploits.html`
10. K2. ADMmutate documentation (2003), `http://www.ktwo.ca/ADMmutate-0.8.4.tar.gz`

11. mati@see security.com. Savant 3.1 Web Server Buffer Overflow Tutorial, http://www.securinfos.info/english/security-whitepapers-hacking-tutorials/Savant-BO-tutorial.pdf
12. Parampalli, C., Sekar, R., Johnson, R.: A practical mimicry attack against powerful system-call monitors. In: Proceedings of the ACM Symposium on Information, Computer and Communications Security (ASIACCS 2008), Tokyo (March 2008)
13. Payer, U., Teufl, P., Lamberger, M.: Hybrid engine for polymorphic shellcode detection. In: Julisch, K., Krügel, C. (eds.) DIMVA 2005. LNCS, vol. 3548, pp. 19–31. Springer, Heidelberg (2005)
14. Polychronakis, M., Anagnostakis, K.G., Markatos, E.P.: Network-level polymorphic shellcode detection using emulation. In: Büschkes, R., Laskov, P. (eds.) DIMVA 2006. LNCS, vol. 4064, pp. 54–73. Springer, Heidelberg (2006)
15. Markatos, E.P., Anagnostakis, K.G., Polychronakis, M.: Emulation-Based Detection of Non-self-contained Polymorphic Shellcode. In: Kruegel, C., Lippmann, R., Clark, A. (eds.) RAID 2007. LNCS, vol. 4637, pp. 87–106. Springer, Heidelberg (2007)
16. Determina Security Research. Windows Animated Cursor Stack Overflow Vulnerability, http://www.determina.com/security.research/vulnerabilities/ani-header.html
17. Rubin, S., Jha, S., Miller, B.: Automatic generation and analysis of nids attacks. In: Proceedings of the Annual Computer Security Applications Conference 2004 (ACSAC 2004), California (December 2004)
18. Shacham, H.: The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In: Proceedings of the 14th Conference on Computer and Communication Security (CCS 2007), Virginia (October 2007)
19. Sidiroglou, S., Locasto, M.E., Boyd, S.W., Keromytis, A.D.: Building a reactive immune system for software services. In: Proceedings of the USENIX Annual Technical Conference 2005, California (April 2005)
20. Song, Y., Locasto, M.E., Stavrou, A., Keromytis, A.D., Stolfo, S.J.: On the infeasibility of modeling polymorphic shellcode. In: Proceedings of the 13th Conference on Computer and Communication Security (CCS 2007), Virginia (October 2007)
21. Toth, T., Kruegel, C.: Accurate buffer overflow detection via abstract payload execution. In: Wespi, A., Vigna, G., Deri, L. (eds.) RAID 2002. LNCS, vol. 2516. Springer, Heidelberg (2002)
22. US-CERT. Vulnerability Note VU#29823: Format string input validation error in wu-ftpd site_exec() function, http://www.kb.cert.org/vuls/id/29823
23. Wang, K., Cretu, G., Stolfo, S.J.: Anomalous payload-based worm detection and signature generation. In: Valdes, A., Zamboni, D. (eds.) RAID 2005. LNCS, vol. 3858, pp. 227–246. Springer, Heidelberg (2006)
24. Wang, X., Pan, C.C., Liu, P., Zhu, S.: Sigfree: A signature-free buffer overflow attack blocker. In: Proceedings of 15th USENIX Security Symposium Abstract (USENIX Security 2006), Vancouver (July 2006)
25. Zhang, Q., Reeves, D.S., Ning, P., Iyer, S.P.: Analyzing network traffic to detect self-decryption exploit code. In: Proceedings of the 2nd ACM Symposium on InformAtion, Computer and Communications Security (ASIACCS 2007), Singapore (March 2007)

Leveraging User Interactions for In-Depth Testing of Web Applications

Sean McAllister[1], Engin Kirda[2], and Christopher Kruegel[3]

[1] Secure Systems Lab, Technical University Vienna, Austria
sean@seclab.tuwien.ac.at
[2] Institute Eurecom, France
kirda@eurecom.fr
[3] University of California, Santa Barbara
chris@cs.ucsb.edu

Abstract. Over the last years, the complexity of web applications has grown significantly, challenging desktop programs in terms of functionality and design. Along with the rising popularity of web applications, the number of exploitable bugs has also increased significantly. Web application flaws, such as cross-site scripting or SQL injection bugs, now account for more than two thirds of the reported security vulnerabilities.

Black-box testing techniques are a common approach to improve software quality and detect bugs before deployment. There exist a number of vulnerability scanners, or fuzzers, that expose web applications to a barrage of malformed inputs in the hope to identify input validation errors. Unfortunately, these scanners often fail to test a substantial fraction of a web application's logic, especially when this logic is invoked from pages that can only be reached after filling out complex forms that aggressively check the correctness of the provided values.

In this paper, we present an automated testing tool that can find reflected and stored cross-site scripting (XSS) vulnerabilities in web applications. The core of our system is a black-box vulnerability scanner. This scanner is enhanced by techniques that allow one to generate more comprehensive test cases and explore a larger fraction of the application. Our experiments demonstrate that our approach is able to test more thoroughly these programs and identify more bugs than a number of open-source and commercial web vulnerability scanners.

1 Introduction

The first web applications were collections of static files, linked to each other by means of HTML references. Over time, dynamic features were added, and web applications started to accept user input, changing the presentation and content of the pages accordingly. This dynamic behavior was traditionally implemented by CGI scripts. Nowadays, more often then not, complete web sites are created dynamically. To this end, the site's content is stored in a database. Requests are processed by the web application to fetch the appropriate database entries and present them to the user. Along with the complexity of the web sites, the use cases have also become more involved. While in the beginning user interaction was typically limited to simple request-response pairs, web applications today often require a multitude of intermediate steps to achieve the desired results.

When developing software, an increase in complexity typically leads to a growing number of bugs. Of course, web applications are no exception. Moreover, web applications can be quickly deployed to be accessible to a large number of users on the Internet, and the available development frameworks make it easy to produce (partially correct) code that works only in most cases. As a result, web application vulnerabilities have sharply increased. For example, in the last two years, the three top positions in the annual Common Vulnerabilities and Exposures (CVE) list published by Mitre [17] were taken by web application vulnerabilities.

To identify and correct bugs and security vulnerabilities, developers have a variety of testing tools at their disposal. These programs can be broadly categorized as based on black-box approaches or white-box approaches. White-box testing tools, such as those presented in [2, 15, 27, 32], use static analysis to examine the source code of an application. They aim at detecting code fragments that are patterns of instances of known vulnerability classes. Since these systems do not execute the application, they achieve a large code coverage, and, in theory, can analyze all possible execution paths. A drawback of white-box testing tools is that each tool typically supports only very few (or a single) programming language. A second limitation is the often significant number of false positives. Since static code analysis faces undecidable problems, approximations are necessary. Especially for large software applications, these approximations can quickly lead to warnings about software bugs that do not exist.

Black-box testing tools [11] typically run the application and monitor its execution. By providing a variety of specially-crafted, malformed input values, the goal is to find cases in which the application misbehaves or crashes. A significant advantage of black-box testing is that there are no false positives. All problems that are reported are due to real bugs. Also, since the testing tool provides only input to the application, no knowledge about implementation-specific details (e.g., the used programming language) is required. This allows one to use the same tool for a large number of different applications. The drawback of black-box testing tools is their limited code coverage. The reason is that certain program paths are exercised only when specific input is provided.

Black-box testing is a popular choice when analyzing web applications for security errors. This is confirmed by the large number of open-source and commercial black-box tools that are available [1, 16, 19, 29]. These tools, also called web vulnerability scanners or fuzzers, typically check for the presence of well-known vulnerabilities, such as cross-site scripting (XSS) or SQL injection flaws. To check for security bugs, vulnerability scanners are equipped with a large database of test values that are crafted to trigger XSS or SQL injection bugs. These values are typically passed to an application by injecting them into the application's HTML form elements or into URL parameters.

Web vulnerability scanners, sharing the well-known limitation of black-box tools, can only test those parts of a web site (and its underlying web application) that they can reach. To explore the different parts of a web site, these scanners frequently rely on built-in web spiders (or crawlers) that follow links, starting from a few web pages that act as seeds. Unfortunately, given the increasing complexity of today's applications, this is often insufficient to reach "deeper" into the web site. Web applications often implement a complex workflow that requires a user to correctly fill out a series of forms. When the scanner cannot enter meaningful values into these forms, it will not reach certain parts of the site. Therefore, these parts are not tested, limiting the effectiveness of black-box testing for web applications.

In this paper, we present techniques that improve the effectiveness of web vulnerability scanners. To this end, our scanner leverages input from real users as a starting point for its testing activity. More precisely, starting from recorded, actual user input, we generate test cases that can be replayed. By following a user's session, fuzzing at each step, we are able to increase the code coverage by exploring pages that are not reachable for other tools. Moreover, our techniques allow a scanner to interact with the web application in a more meaningful fashion. This often leads to test runs where the web application creates a large number of persistent objects (such as database entries). Creating objects is important to check for bugs that manifest when malicious input is stored in a database, such as in the case of stored cross-site scripting (XSS) vulnerabilities. Finally, when the vulnerability scanner can exercise some control over the program under test, it can extract important feedback from the application that helps in further improving the scanner's effectiveness.

We have implemented our techniques in a vulnerability scanner that can analyze applications that are based on the Django web development framework [8]. Our experimental results demonstrate that our tool achieves larger coverage and detects more vulnerabilities than existing open-source and commercial fuzzers.

2 Web Application Testing and Limitations

One way to quickly and efficiently identify flaws in web applications is the use of vulnerability scanners. These scanners test the application by providing malformed inputs that are crafted so that they trigger certain classes of vulnerabilities. Typically, the scanners cover popular vulnerability classes such as cross-site scripting (XSS) or SQL injection bugs. These vulnerabilities are due to input validation errors. That is, the web application receives an input value that is used at a security-critical point in the program without (sufficient) prior validation. In case of an XSS vulnerability [10], malicious input can reach a point where it is sent back to the web client. At the client side, the malicious input is interpreted as JavaScript code that is executed in the context of the trusted web application. This allows an attacker to steal sensitive information such as cookies. In case of a SQL injection flaw, malicious input can reach a database query and modify the intended semantics of this query. This allows an attacker to obtain sensitive information from the database or to bypass authentication checks.

By providing malicious, or malformed, input to the web application under test, a vulnerability scanner can check for the presence of bugs. Typically, this is done by analyzing the response that the web application returns. For example, a scanner could send a string to the program that contains malicious JavaScript code. Then, it checks the output of the application for the presence of this string. When the malicious JavaScript is present in the output, the scanner has found a case in which the application does not properly validate input before sending it back to clients. This is reported as an XSS vulnerability.

To send input to web applications, scanners only have a few possible injection points. According to [26], the possible points of attack are the URL, the cookie, and the POST data contained in a request. These points are often derived from form elements that are present on the web pages. That is, web vulnerability scanners analyze web pages to find injection points. Then, these injection points are fuzzed by sending a large number of requests that contain malformed inputs.

Limitations. Automated scanners have a significant disadvantage compared to human testers in the way they can interact with the application. Typically, a user has certain goals in mind when interacting with a site. On an e-commerce site, for example, these goals could include buying an item or providing a rating for the most-recently-purchased goods. The goals, and the necessary operations to achieve these goals, are known to a human tester. Unfortunately, the scanner does not have any knowledge about use cases; all it can attempt to do is to collect information about the available injection points and attack them. More precisely, the typical workflow of a vulnerability scanners consists of the following steps:

– First, a web spider crawls the site to find valid injection points. Commonly, these entry points are determined by collecting the links on a page, the action attributes of forms, and the source attributes of other tags. Advanced spiders can also parse JavaScript to search for URLs. Some even execute JavaScript to trigger requests to the server.
– The second phase is the audit phase. During this step, the scanner fuzzes the previously discovered entry points. It also analyzes the application's output to determine whether a vulnerability was triggered.
– Finally, many scanners will start another crawling step to find stored XSS vulnerabilities. In case of a stored XSS vulnerability, the malicious input is not immediately returned to the client but stored in the database and later included in another request. Therefore, it is not sufficient to only analyze the application's immediate response to a malformed input. Instead, the spider makes a second pass to check for pages that contain input injected during the second phase.

The common workflow outlined above yields good results for simple sites that do not require a large amount of user interaction. Unfortunately, it often fails when confronted with more complex sites. The reason is that vulnerability scanners are equipped with simple rules to fill out forms. These rules, however, are not suited well to advance “deeper” into an application when the program enforces constraints on the input values that it expects. To illustrate the problem, we briefly discuss an example of how a fuzzer might fail on a simple use case.

The example involves a blogging site that allows visitors to leave comments to each entry. To leave a comment, the user has to fill out a form that holds the content of the desired comment. Once this form is submitted, the web application responds with a page that shows a preview of the comment, allowing the user to make changes before submitting the posting. When the user decides to make changes and presses the corresponding button, the application returns to the form where the text can be edited. When the user is satisfied with her comment, she can post the text by selecting the appropriate button on the preview page.

The problem in this case is that the submit button (which actually posts the message to the blog) is activated on the preview page only when the web application recognizes the submitted data as a valid comment. This requires that both the name of the author and the text field of the comment are filled in. Furthermore, it is required that a number of hidden fields on the page remain unchanged. When the submit button is successfully pressed, a comment is created in the application’s database, linked to the article, and subsequently shown in the comments section of the blog entry.

For a vulnerability scanner, posting a comment to a blog entry is an entry point that should be checked for the presence of vulnerabilities. Unfortunately, all of the tools evaluated in our experiments (details in Section 5.2) failed to post a comment. That is, even a relatively simple task, which requires a scanner to fill out two form elements on a page and to press two buttons in the correct order, proved to be too difficult for an automated scanner. Clearly, the situation becomes worse when facing more complex use cases.

During our evaluation of existing vulnerability scanners, we found that, commonly, the failure to detect a vulnerability is not due to the limited capabilities of the scanner to inject malformed input or to determine whether a response indicates a vulnerability, but rather due to the inability to generate enough valid requests to reach the vulnerable entry points. Of course, the exact reasons for failing to reach entry points vary, depending on the application that is being tested and the implementation of the scanner.

3 Increasing Test Coverage

To address the limitations of existing tools, we propose several techniques that allow a vulnerability scanner to detect more entry points. These entry points can then be tested, or fuzzed, using existing databases of malformed input values. The first technique, described in Section 3.1, introduces a way to leverage inputs that are recorded by observing actual user interaction. This allows the scanner to follow an actual use case, achieving more depth when testing. The second technique, presented in Section 3.2, discusses a way to abstract from observed user inputs, leveraging the steps of the use case to achieve more breadth. The third technique, described in Section 3.3, makes the second technique more robust in cases where the broad exploration interferes with the correct replay of a use case.

3.1 Increasing Testing Depth

One way to improve the coverage, and thus, the effectiveness of scanners, is to leverage actual user input. That is, we first collect a small set of inputs that were provided by users that interacted with the application. These interactions correspond to certain use cases, or workflows, in which a user carries out a sequence of steps to reach a particular goal. Depending on the application, this could be a scenario where the user purchases an item in an on-line store or a scenario where the user composes and sends an email using a web-based mail program. Based on the recorded test cases, the vulnerability scanner can replay the collected input values to successfully proceed a number of steps into the application logic. The reason is that the provided input has a higher probability to pass server-side validation routines. Of course, there is, by no means, a guarantee that recorded input satisfies the constrains imposed by an application at the time the values are replayed. While replaying a previously recorded use case, the scanner can fuzz the input values that are provided to the application.

Collecting input. There are different locations where client-supplied input data can be collected. One possibility is to deploy a proxy between a web client and the web server, logging the requests that are sent to the web application. Another way is to record the incoming requests at the server side, by means of web server log files or application level logging. For simplicity, we record requests directly at the server, logging the names and values of all input parameters.

It is possible to record the input that is produced during regular, functional testing of applications. Typically, developers need to create test cases that are intended to exercise the complete functionality of the application. When such test cases are available, they can be immediately leveraged by the vulnerability scanner. Another alternative is to deploy the collection component on a production server and let real-world users of the web application generate test cases. In any case, the goal is to collect a number of inputs that are likely correct from the application's point of view, and thus, allow the scanner to reach additional parts of the application that might not be easily reachable by simply crawling the site and filling out forms with essentially random values. This approach might raise some concerns with regards to the nature of the captured data. The penetration tester must be aware of the fact that user input is being captured and stored in clear text. This is acceptable for most sites but not for some (because, for example, the unencrypted storage of sensitive information such as passwords and credit card numbers might be unacceptable). In these cases, it is advisable to perform all input capturing and tests in a controlled testbed.

Replaying input. Each use case consists of a number of steps that are carried out to reach a certain goal. For each step, we have recorded the requests (i.e., the input values) that were submitted. Based on these input values, the vulnerability scanner can replay a previously collected use case. To this end, the vulnerability scanner replays a recorded use case, one step at a time. After each step, a fuzzer component is invoked. This fuzzer uses the request issued in the previous step to test the application. More precisely, it uses a database of malformed values to replace the valid inputs within the request sent in the previous step. In other words, after sending a request as part of a replayed use case, we attempt to fuzz this request. Then, the previously recorded input values stored for the current step are used to advance to the next step. This process of fuzzing a request and subsequently advancing one step along the use case is repeated until the test case is exhausted. Alternatively, the process stops when the fuzzer replays the recorded input to advance to the next page, but this page is different from the one expected. This situation can occur when a previously recorded input is no longer valid.

When replaying input, the vulnerability scanner does not simply re-submit a previously recorded request. Instead, it scans the page for elements that require user input. Then, it uses the previously recorded request to provide input values for those elements only. This is important when an application uses cookies or hidden form fields that are associated with a particular session. Changing these values would cause the application to treat the request as invalid. Thus, for such elements, the scanner uses the current values instead of the “old” ones that were previously collected. The rules used to determine the values of each form field aim to mimic the actions of a benign user. That is, hidden fields are not changed, as well as read-only widgets (such as submit button values or disabled elements). Of course security vulnerabilities can also be triggered by malicious input data within these hidden fields, but this is of no concern at this stage because the idea is to generate benign and valid input and then apply the attack logic to these values. Later on, during the attacking stage, the fuzzer will take care that all parameters will be tested.

Guided fuzzing. We call the process of using previously collected traces to step through an application *guided fuzzing.* Guided fuzzing improves the coverage of a vulnerability scanner because it allows the tool to reach entry points that were previously hidden behind forms that expect specific input values. That is, we can increase the depth that a scanner can reach into the application.

3.2 Increasing Testing Breadth

With guided fuzzing, after each step that is replayed, the fuzzer only tests the single request that was sent for that step. That is, for each step, only a single entry point is analyzed. A straightforward extension to guided fuzzing is to not only test the single entry point, but to use the current step as a starting point for fuzzing the complete site that is reachable from this point. That is, the fuzzer can use the current page as its starting point, attempting to find additional entry points into the application. Each entry point that is found in this way is then tested by sending malformed input values. In this fashion, we do not only increase the depth of the test cases, but also their breadth. For example, when a certain test case allows the scanner to bypass a form that performs aggressive input checking, it can then explore the complete application space that was previously hidden behind that form. We call this approach *extended, guided fuzzing.*

Extended, guided fuzzing has the potential to increase the number of entry points that a scanner can test. However, alternating between a comprehensive fuzzing phase and advancing one step along a recorded use case can also lead to problems. To see this, consider the following example. Assume an e-commerce application that uses a shopping cart to hold the items that a customer intends to buy. The vulnerability scanner has already executed a number of steps that added an item to the cart. At this point, the scanner encounters a page that shows the cart's inventory. This page contains several links; one link leads to the checkout view, the other one is used to delete items from the cart. Executing the fuzzer on this page can result in a situation where the shopping cart remains empty because all items are deleted. This could cause the following steps of the use case to fail, for example, because the application no longer provides access to the checkout page. A similar situation can arise when administrative pages are part of a use case. Here, running a fuzzer on a page that allows the administrator to delete all database entries could be very problematic.

In general terms, the problem with extended, guided fuzzing is that the fuzzing activity could interfere in undesirable ways with the use case that is replayed. In particular, this occurs when the input sent by the fuzzer changes the state of the application such that the remaining steps of a use case can no longer be executed. This problem is difficult to address when we assume that the scanner has no knowledge and control of the inner workings of the application under test. In the following Section 3.3, we consider the case in which our test system can interact more tightly with the analyzed program. In this case, we are able to prevent the undesirable side effects (or interference) from the fuzzing phases.

3.3 Stateful Fuzzing

The techniques presented in the previous sections work independently of the application under test. That is, our system builds black-box test cases based on previously recorded user input, and it uses these tests to check the application for vulnerabilities. In this subsection, we consider the case where the scanner has some control over the application under test.

One solution to the problem of undesirable side effects of the fuzzing step when replaying recorded use cases is to *take a snapshot* of the state of the application after each step that is replayed. Then, the fuzzer is allowed to run. This might result in significant changes to the application's state. However, after each fuzzing step, the application is *restored* to the previously taken snapshot. At this point, the replay component will find the application in the expected state and can advance one step. After that, the process is repeated - that is, a snapshot is taken and the fuzzer is invoked. We call this process *stateful fuzzing.*

In principle, the concrete mechanisms to take a snapshot of an application's state depend on the implementation of this application. Unfortunately, this could be different for each web application. As a result, we would have to customize our test system to each program, making it difficult to test a large number of different applications. Clearly, this is very undesirable. Fortunately, the situation is different for web applications. Over the last years, the model-view-controller (MVC) scheme has emerged as the most popular software design pattern for applications on the web. The goal of the MVC scheme is to separate three layers that are present in almost all web applications. These are the data layer, the presentation layer, and the application logic layer. The data layer represents the data storage that handles persistent objects. Typically, this layer is implemented by a backend database and an object (relational) manager. The application logic layer uses the objects provided by the data layer to implement the functionality of the application. It uses the presentation layer to format the results that are returned to clients. The presentation layer is frequently implemented by an HTML template engine. Moreover, as part of the application logic layer, there is a component that maps requests from clients to the corresponding functions or classes within the program.

Based on the commonalities between web applications that follow an MVC approach, it is possible (for most such applications) to identify general interfaces that can be instrumented to implement a snapshot mechanism. To be able to capture the state of the application and subsequently restore it, we are interested in the objects that are created, updated, or deleted by the object manager in response to requests. Whenever an object is modified or deleted, a copy of this object is serialized and saved. This way, we can, for example, undelete an object that has been previously deleted, but that is required when a use case is replayed. In a similar fashion, it is also possible to undo updates to an object and delete objects that were created by the fuzzer.

The information about the modification of objects can be extracted at the interface between the application and the data layer (often, at the database level). At this level, we insert a component that can serialize modified objects and later restore the snapshot of the application that was previously saved. Clearly, there are limitations to this technique. One problem is that the state of an application might not depend solely on the state of the persistent objects and its attributes. Nevertheless, this technique has the potential to increase the effectiveness of the scanner for a large set of programs that follow a MVC approach. This is also confirmed by our experimental results presented in Section 5.

Application feedback. Given that stateful fuzzing already requires the instrumentation of the program under test, we should consider what additional information might be useful to further improve the vulnerability scanning process.

One piece of feedback from the application that we consider useful is the *mapping of URLs to functions.* This mapping can be typically extracted by analyzing or instrumenting the controller component, which acts as a dispatcher from incoming requests to the appropriate handler functions. Using the mappings between URLs and the program functions, we can increase the effectiveness of the extended, guided fuzzing process. To this end, we attempt to find a set of forms (or URLs) that all invoke the same function within the application. When we have previously seen user input for one of these forms, we can reuse the same information on other forms as well (when no user input was recorded for these forms). The rationale is that information that was provided to a certain function through one particular form could also be valid when submitted as part of a related form. By reusing information for forms that the fuzzer encounters, it is possible to reach additional entry points.

When collecting user input (as discussed in Section 3.1), we record all input values that a user provides on each page. More precisely, for each URL that is requested, we store all the name-value pairs that a user submits with this request. In case the scanner can obtain application feedback, we also store the name of the program function that is invoked by the request. In other words, we record the name of the function that the requested URL maps to. When the fuzzer later encounters an unknown action URL of a form (i.e., the URL where the form data is submitted to), we query the application to determine which function this URL maps to. Then, we search our collected information to see whether the same function was called previously by another URL. If this is the case, we examine the name-value pairs associated with this other URL. For each of those names, we attempt to find a form element on the current page that has a similar name. When a similar name is found, the corresponding, stored value is supplied. As mentioned previously, the assumption is that valid data that was passed to a program function through one form might also be valid when used for a different form, in another context. This can help in correctly filling out unknown forms, possibly leading to unexplored entry points and vulnerabilities.

As an example, consider a forum application where each discussion thread has a reply field at the bottom of the page. The action URLs that are used for submitting a reply could be different for each thread. However, the underlying function that is eventually called to save the reply and link it to the appropriate thread remains the same. Thus, when we have encountered one case where a user submitted a reply, we would recognize other reply fields for different threads as being similar. The reason is that even though the action URLs associated with the reply forms are different, they all map to the same program function. Moreover, the name of the form fields are (very likely) the same. As a result, the fuzzer can reuse the input value(s) recorded in the first case on other pages.

4 Implementation Details

We developed a vulnerability scanner that implements the techniques outlined above. As discussed in the last section, some of the techniques require that a web application is instrumented **(i)** to capture and restore objects manipulated by the application, and **(ii)** to extract the mappings between URLs and functions. Therefore, we were looking for a web development framework that supports the model-view-controller (MVC) scheme. Among the candidates were most popular web development frameworks, such as Ruby on Rails [7], Java Servlets [28], or Django [8], which is based upon Python. Since we are familiar with Python, we selected the Django framework. That is, we extended the Django framework such that it provides the necessary functionality for the vulnerability scanner. Our choice implies that we can currently only test web applications that are built using Django. Note, however, that the previously introduced concepts are general and can be ported to other development frameworks (i.e., with some additional engineering effort, we could use our techniques to test applications based upon other frameworks).

Capturing web requests. The first task was to extend Django such that it can record the inputs that are sent when going through a use case. This makes it necessary to log all incoming requests together with the corresponding parameters. In Django, all incoming requests pass through two middleware classes before reaching the actual application code. One of these classes is a URL dispatcher class that determines the function that should be invoked. At this point, we can log the complete request information. Also, the URL dispatcher class provides easy access to the mapping between URLs and the functions that are invoked.

Replaying use cases. Once a use case, which consists of a series of requests, has been collected, it can be used for replaying. To this end, we have developed a small test case replay component based on twill [30], a testing tool for web applications. This component analyzes a page and attempts to find the form elements that need to be filled out, based on the previously submitted request data.

Capturing object manipulations. Our implementation uses the Django middleware classes to attach event listeners to incoming requests. These event listeners wait for signals that are raised every time an object is created, updated, or deleted. The signals are handled synchronously, meaning that the execution of the code that sent the signal is postponed until the signal handler has finished. We exploit this fact to create copies of objects before they are saved to the backend storage, allowing us to later restore any object to a previous state.

Fuzzer component. An important component of the vulnerability scanner is the fuzzer. The task of the fuzzer component is to expose each entry point that it finds to a set of malformed inputs that can expose XSS vulnerabilities. Typically, it also features a web spider that uses a certain page as a starting point to reach other parts of the application, checking each page that is encountered.

Because the focus of this work is not on the fuzzer component but on techniques that can help to make this fuzzer more effective, we decided to use an existing web application testing tool. The choice was made for the “Web Application Attack and Audit Framework,” or shorter, w3af [31], mainly because the framework itself is easy to extend and actively maintained.

5 Evaluation

For our experiments, we installed three publicly available, real-world web applications based on Django (SVN Version 6668):

– The first application was a blogging application, called Django-basic-blog [9]. We did not install any user accounts. Initially, the blog was filled with three articles. Comments were enabled for each article, and no other links were present on the page. That is, the comments were the only interactive component of the site.
– The second application was a forum software, called Django-Forum [23]. To provide all fuzzers with a chance to explore more of the application, every access was performed as coming from a privileged user account. Thus, each scanner was making requests as a user that could create new threads and post replies. Initially, a simple forum structure was created that consisted of three forums.
– The third application was a web shop, the Satchmo online shop 0.6 [24]. This site was larger than the previous two applications, and, therefore, more challenging to test. The online shop was populated with the test data included in the package, and one user account was created.

We selected these three programs because they represent common archetypes of applications on the Internet. For our experiments, we used Apache 2.2.4 (with pre-forked worker threads) and mod_python 3.3.1. Note that before a new scanner was tested on a site, the application was restored to its initial state.

5.1 Test Methodology

We tested each of the three aforementioned web applications with three existing web vulnerability scanners, as well as with our own tool. The scanners that we used were Burp Spider 1.21 [5], w3af spider [31], and Acunetix Web Vulnerability Scanner 5.1 (Free Edition) [1]. Each scanner is implemented as a web spider that can follow links on web pages. All scanners also have support for filling out forms and, with the exception of the Burp Suite Spider, a fuzzer component to check for XSS vulnerabilities. For each page that is found to contain an XSS vulnerability, a warning is issued. In addition to the three vulnerability scanners and our tool, we also included a very simple web spider into the tests. This self-written spider follows all links on a page. It repeats this process recursively for all pages that are found, until all available URLs are exhausted. This web spider serves as the lower bound on the number of pages that should be found and analyzed by each vulnerability scanner.

We used the default configuration for all tools. One exception was that we enabled the form filling option for the Burp Spider. Moreover, for the Acunetix scanner, we activated the "extensive scan feature," which optimizes the scan for mod_python applications and checks for stored XSS.

When testing our own tool, we first recorded a simple use case for each of the three applications. The use cases included posting a comment for the blog, creating a new thread and a post on the forum site, and purchasing an item in the online store. Then, we executed our system in one of three modes. First, guided fuzzing was used. In the second run, we used extended, guided fuzzing (together with application feedback). Finally, we scanned the program using stateful fuzzing.

There are different ways to assess the effectiveness or coverage of a web vulnerability scanner. One metric is clearly the number of vulnerabilities that are reported. Unfortunately, this number could be misleading because a single program bug might manifest itself on many pages. For example, a scanner might find a bug in a form that is reused on several pages. In this case, there is only a single vulnerability, although the number of warnings could be significantly larger. Thus, the number of unique bugs, or vulnerable *injection points*, is more representative than the number of warnings.

Another way to assess coverage is to count the number of *locations* that a scanner visits. A location represents a unique, distinct page (or, more precisely, a distinct URL). Of course, visiting more locations potentially allows a scanner to test more of the application's functionality. Assume that, for a certain application, Scanner A is able to explore significantly more locations than Scanner B. However, because Scanner A misses one location with a vulnerability that Scanner B visits, it reports fewer vulnerable injection points. In this case, we might still conclude that Scanner A is better, because it achieves a larger coverage. Unfortunately, this number can also be misleading, because different locations could result from different URLs that represent the same, underlying page (e.g., the different pages on a forum, or different threads on a blog).

Finally, for the detection of vulnerabilities that require the scanner to store malicious input into the database (such as stored XSS vulnerabilities), it is more important to create many different database objects than to visit many locations. Thus, we also consider the number and diversity of different (database) objects that each scanner creates while testing an application.

Even though we only tested for XSS vulnerabilities, many other attacks can be performed against web applications. XSS is a very common and well understood vulnerability and, therefore, we selected this type of attack for our testing. However, the techniques presented in this paper apply to other injection attacks as well (for example, SQL injection and directory traversal attacks).

5.2 Experimental Results

In this section, we present and discuss the results that the different scanners achieve when analyzing the three test applications. For each application, we present the number of locations that the scanner has visited, the number of reported vulnerabilities, the number of injection points (unique bugs) that these reports map to, and the number of relevant database objects that were created.

Blogging application. Table 1 shows the results for the simple blog application. Compared to the simple spider, one can see that all other tools have reached more locations. This is because all spiders (except the simple one) requested the root of each identified directory. When available, these root directories can provide additional links to pages that might not be reachable from the initial page. As expected, it can be seen that extended, guided fuzzing reaches more locations than guided fuzzing alone, since it attempts to explore the application in breadth. Moreover, there is no difference between the results for the extended,

Table 1. Scanner effectiveness for blog application

	Locations	POST/GET Requests	Comments	XSS Warnings		Injection Points	
				Reflected	Stored	Reflected	Stored
Spider	4	4	-	-	-	-	-
Burp Spider	8	25	0	-	-	-	-
w3af	9	133	0	0	0	0	0
Acunetix	9	22	0	0	0	0	0
Use Case	4	4	1	-	-	-	-
Guided Fuzzing	4	64	12	0	1	0	1
Extended Fuzz.	6	189	12	0	1	0	1
Stateful Fuzz.	6	189	12	0	1	0	1

guided fuzzing and the stateful fuzzing approach. The reason is that, for this application, invoking the fuzzer does not interfere with the correct replay of the use case.

None of the three existing scanners was able to create a valid comment on the blogging system. This was because the posting process is not straightforward: Once a comment is submitted, the blog displays a form with a preview button. This allows a user to either change the content of the comment or to post it. The problem is that the submit button (to actually post the message) is not part of the page until the server-side validation recognizes the submitted data as a valid comment. To this end, both comment fields (name and comment) need to be present. Here, the advantage of guided fuzzing is clear. Because our system relies on a previously recorded test case, the fuzzer can correctly fill out the form and post a comment. This is beneficial, because it is possible to include malicious JavaScript into a comment and expose the stored XSS vulnerability that is missed by the other scanners. Concerning the number of injection points, which are higher for some tested scanners, it has to be noted that this is caused by the way in which some scanners attempt to find new attack points. When discovering a new URL, these scanners also issue requests for all subdirectories of the injection point. Depending on the application, this might lead to the discovery of new pages (injection points), redirects, or page-not-found errors. As our fuzzer focuses on following use cases, we did not implement this heuristics for our scanner (of course, it could be easily added).

Forum application. For the forum application, the scanners were able to generate some content, both in the form of new discussion threads and replies. Table 2 shows that while Burp Spider [5] and w3af [31] were able to create new discussion threads, only the Acunetix scanner managed to post replies as well. w3af correctly identified the form's action URL to post a reply, but failed to generate valid input data that would have resulted in the reply being stored in the database. However, since the vulnerability is caused by a bug in the routine that validates the thread title, posting replies is not necessary to identify the flaw in this program.

Table 2. Scanner effectiveness for the forum application

	Locations	POST/GET	Threads	Replies	XSS Warnings		Inject. Points	
		Requests	Created	Created	Reflect	Stored	Reflect	Stored
Spider	8	8	-	-	-	-	-	-
Burp Spider	8	32	0	0	-	-	-	-
w3af	14	201	29	0	0	3	0	1
Acunetix	263	2,003	687	1,486	63	63	0	1
Use Case	6	7	1	2	-	-	-	-
Guided Fuzzing	16	48	12	22	0	1	0	1
Extended Fuzz.	85	555	36	184	0	3	0	1
Stateful Fuzz.	85	555	36	184	0	3	0	1

Both the number of executed requests and the number of reported vulnerabilities differ significantly between the vulnerability scanners tested. It can be seen that the Acunetix scanner has a large database of malformed inputs, which manifests both in the number of requests sent and the number of vulnerabilities reported. For each of the three forum threads, which contain a link to the unique, vulnerable entry point, Acunetix sent 21 fuzzed requests. Moreover, the Acunetix scanner reports each detected vulnerability twice. That is, each XSS vulnerability is reported once as reflected and once as stored XSS. As a result, the scanner generated 126 warnings for a single bug. w3af, in comparison, keeps an internal knowledge base of vulnerabilities that it discovers. Therefore, it reports each vulnerability only once (and the occurrence of a stored attack replaces a previously found, reflected vulnerability).

The results show that all our techniques were able to find the vulnerability that is present in the forum application. Similar to the Acunetix scanner (but unlike w3af), they were able to create new threads and post replies. Again, the extended, guided fuzzing was able to visit more locations than the guided fuzzing alone (it can be seen that the extended fuzzing checked all three forum threads that were present initially, while the guided fuzzing only analyzed the single forum thread that was part of the recorded use case). Moreover, the fuzzing phase was not interfering with the replay of the use cases. Therefore, the stateful fuzzing approach did not yield any additional benefits.

Online shopping application. The experimental results for the online shopping application are presented in Tables 3 and 4. Table 3 presents the scanner effectiveness based on the number of locations that are visited and the number of vulnerabilities that are detected, while Table 4 compares the number of database objects that were created by both the Acunetix scanner and our approaches. Note that the Acunetix scanner offers a feature that allows the tool to make use of login credentials and to block the logout links. For this experiment, we made two test runs with the Acunetix scanner: The first run (#1) as anonymous user and the second test run (#2) by enabling this feature.

Both w3af and Acunetix identified a reflected XSS vulnerability in the login form. However, neither of the two scanners was able to reach deep into the

Table 3. Scanner effectiveness for the online shopping application

	Locations	POST/GET Requests	XSS Warnings		Injection Points	
			Reflected	Stored	Reflected	Stored
Spider	18	18	-	-	-	-
Burp Spider	22	52	-	-	-	-
w3af	21	829	1	0	1	0
Acunetix #1	22	1,405	16	0	1	0
Acunetix #2	25	2,564	8	0	1	0
Use Case	22	36	-	-	-	-
Guided Fuzzing	22	366	1	8	1	8
Extended Fuzz.	25	1,432	1	0	1	0
Stateful Fuzz.	32	2,078	1	8	1	8

Table 4. Object creation statistics for the online shopping application

Object Class	Acunetix #1	Acunetix #2	Use Case	Guided Fuzzing	Extended Fuzzing	Stateful Fuzzing
OrderItem	-	-	1	1	-	2
AddressBook	-	-	2	2	-	7
PhoneNumber	-	-	1	3	-	5
Contact	1	-	1	1	1	2
CreditCardDetail	-	-	1	1	-	2
OrderStatus	-	-	1	1	-	1
OrderPayment	-	-	1	1	-	2
Order	-	-	1	1	-	2
Cart	2	1	1	1	3	3
CartItem	2	1	1	1	5	5
Comment	-	-	1	21	11	96
User	1	-	1	1	1	1

application. As a result, both tools failed to reach and correctly fill out the form that allows to change the contact information of a user. This form contained eight stored XSS vulnerabilities, since none of the entered input was checked by the application for malicious values. However, the server checked the phone number and email address for their validity and would reject the complete form whenever one of the two values was incorrect.

In contrast to the existing tools, guided fuzzing was able to analyze a large part of the application, including the login form and the user data form. Thus, this approach reported a total of nine vulnerable entry points. In this experiment, we can also observe the advantages of stateful fuzzing. With extended, guided fuzzing, the fuzzing step interferes with the proper replay of the use case (because the fuzzer logs itself out and deletes all items from the shopping cart). The stateful fuzzer, on the other hand, allows to explore a broad range of entry points, and, using the snapshot mechanism, keeps the ability to replay the test case. The number of database objects created by the different approaches (as shown in Table 4) also confirms the ability of our techniques to create a large variety of different, valid objects, a result of analyzing large portions of the application.

Discussion. All vulnerabilities that we found in our experiments were previously unknown, and we reported them to the developers of the web applications. Our results show that our fuzzing techniques consistently find more (or, at least, the same amount) of bugs than other open-source and commercial scanners. Moreover, it can be seen that the different approaches carry out meaningful interactions with the web applications, visiting many locations and creating a large variety of database objects. Finally, the different techniques exhibit different strengths. For example, stateful fuzzing becomes useful especially when the tested application is more complex and sensitive to the fuzzing steps.

6 Related Work

Concepts such as vulnerability testing, test case generation, and fuzzing are well-known concepts in software engineering and vulnerability analysis [3, 4, 11]. When analyzing web applications for vulnerabilities, black-box fuzzing tools [1, 5, 31] are most popular. However, as shown by our experiments, they suffer from the problem of test coverage. Especially for applications that require complex interactions or expect specific input values to proceed, black-box tools often fail to fill out forms properly. As a result, they can scan only a small portion of the application. This is also true for SecuBat [16], a web vulnerability scanner that we developed previously. SecuBat can detect reflected XSS and SQL injection vulnerabilities. However, it cannot fill out forms and, thus, was not included in our experiments.

In addition to web-specific scanners, there exist a large body of more general vulnerability detection and security assessment tools. Most of these tools (e.g., Nikto [19], Nessus [29]) rely on a repository of known vulnerabilities that are tested. Our tool, in contrast, aims to discover unknown vulnerabilities in the application under analysis. Besides application-level vulnerability scanners, there are also tools that work at the network level, e.g., nmap [14]. These tools can determine the availability of hosts and accessible services. However, they are not concerned with higher-level vulnerability analysis. Other well-known web vulnerability detection and mitigation approaches in literature are Scott and Sharp's application-level firewall [25] and Huang et al.'s [13] vulnerability detection tool that automatically executes SQL injection attacks. Moreover, there are a large number of static source code analysis tools [15, 27, 32] that aim to identify vulnerabilities.

A field that is closely related to our work is automated test case generation. The methods used to generate test cases can be generally summarized as random, specification-based [20, 22], and model-based [21] approaches. Fuzzing falls into the category of random test case generation. By introducing use cases and guided fuzzing, we improve the effectiveness of random tests by providing some inputs that are likely valid and thus, allow the scanner to reach "deeper" into the application.

A well-known application testing tool, called WinRunner, allows a human tester to record user actions (e.g., input, mouse clicks, etc.) and then to replay these actions while testing. This could be seen similar to guided fuzzing, where inputs are recorded based on observing real user interaction. However, the testing with WinRunner is not fully-automated. The developer needs to write scripts and create check points to compare the expected and actual outcomes from the test runs. By adding automated, random fuzzing to a guided execution approach, we combine the advantages provided by a tool such as WinRunner with black-box fuzzers. Moreover, we provide techniques to generalize from a recorded use case.

Finally, a number of approaches [6, 12, 18] were presented in the past that aim to explore the alternative execution paths of an application to increase the analysis and test coverage of dynamic techniques. The work we present in this paper is analogous in the sense that the techniques aim to identify more code to test. The difference is the way in which the different approaches are realized, as well as their corresponding properties. When exploring multiple execution paths, the system has to track constraints over inputs, which are solved at branching points to determine alternative paths. Our system, instead, leverages known, valid input to directly reach a large part of an application. Then, a black-box fuzzer is started to find vulnerabilities. This provides better scalability, allowing us to quickly examine large parts of the application and expose it to black-box tests.

7 Conclusions

In this paper, we presented a web application testing tool to detect reflected and stored cross-site scripting (XSS) vulnerabilities in web applications. The core of our system is a black-box vulnerability scanner. Unfortunately, black-box testing tools often fail to test a substantial fraction of a web application's logic, especially when this logic is invoked from pages that can only be reached after filling out complex forms that aggressively check the correctness of the provided values. To allow our scanner to reach "deeper" into the application, we introduce a number of techniques to create more comprehensive test cases. One technique, called guided fuzzing, leverages previously recorded user input to fill out forms with values that are likely valid. This technique can be further extended by using each step in the replay process as a starting point for the fuzzer to explore a program more comprehensively. When feedback from the application is available, we can reuse the recorded user input for different forms during this process. Finally, we introduce stateful fuzzing as a way to mitigate potentially undesirable side-effects of the fuzzing step that could interfere with the replay of use cases during extended, guided fuzzing. We have implemented our use-case-driven testing techniques and analyzed three real-world web applications. Our experimental results demonstrate that our approach is able to identify more bugs than several open-source and commercial web vulnerability scanners.

Acknowledgments

This work has been supported by the Austrian Science Foundation (FWF) under grant P-18764, the FIT-IT project SECoverer (Detection of Application Logic Errors in Web Applications), and the Secure Business Austria Competence Center.

References

[1] Acunetix. Acunetix Web Vulnerability Scanner (2008), http://www.acunetix.com/
[2] Balzarotti, D., Cova, M., Felmetsger, V., Jovanov, N., Kirda, E., Kruegel, C., Vigna, G.: Saner: Composing Static and Dynamic Analysis to Validate Sanitization in Web Applications. In: IEEE Security and Privacy Symposium (2008)
[3] Beizer, B.: Software System Testing and Quality Assurance. Van Nostrand Reinhold (1984)
[4] Beizer, B.: Software Testing Techniques. Van Nostrand Reinhold (1990)
[5] Spider, B.: Web Application Security (2008), http://portswigger.net/spider/
[6] Cadar, C., Ganesh, V., Pawlowski, P., Dill, D., Engler, D.: EXE: Automatically Generating Inputs of Death. In: ACM Conference on Computer and Communication Security (2006)
[7] Hannson, D.: Ruby on Rails (2008), http://www.rubyonrails.org/
[8] Django. The Web Framework for Professionals with Deadlines (2008), http://www.djangoproject.com/
[9] Basic Django Blog Application, http://code.google.com/p/django-basic-blog/
[10] Endler, D.: The Evolution of Cross Site Scripting Attacks. Technical report, iDEFENSE Labs (2002)
[11] Ghezzi, C., Jazayeri, M., Mandrioli, D.: Fundamentals of Software Engineering. Prentice-Hall International, Englewood Cliffs (1994)
[12] Godefroid, P., Klarlund, N., Sen, K.: DART. In: Programming Language Design and Implementation (PLDI) (2005)
[13] Huang, Y., Huang, S., Lin, T.: Web Application Security Assessment by Fault Injection and Behavior Monitoring. In: 12th World Wide Web Conference (2003)
[14] Insecure.org. NMap Network Scanner (2008), http://www.insecure.org/nmap/
[15] Jovanovic, N., Kruegel, C., Kirda, E.: Pixy: A Static Analysis Tool for Detecting Web Application Vulnerabilities (Short Paper). In: IEEE Symposium on Security and Privacy (2006)
[16] Kals, S., Kirda, E., Kruegel, C., Jovanovic, N.: SecuBat: A Web Vulnerability Scanner. In: World Wide Web Conference (2006)
[17] Mitre. Common Vulnerabilities and Exposures, http://cve.mitre.org/
[18] Moser, A., Kruegel, C., Kirda, E.: Exploring Multiple Execution Paths for Malware Analysis. In: IEEE Symposium on Security and Privacy (2007)
[19] Nikto. Web Server Scanner (2008), http://www.cirt.net/code/nikto.shtml
[20] Offutt, J., Abdurazik, A.: Generating Tests from UML Specifications. In: Second International Conference on the Unified Modeling Language (1999)
[21] Offutt, J., Abdurazik, A.: Using UML Collaboration Diagrams for Static Checking and Test Generation. In: Evans, A., Kent, S., Selic, B. (eds.) UML 2000. LNCS, vol. 1939, pp. 383–395. Springer, Heidelberg (2000)

[22] Offutt, J., Liu, S., Abdurazik, A., Ammann, P.: Generating Test Data from State-based Specifications. In: Journal of Software Testing, Verification and Reliability (2003)
[23] Poulton, R.: Django Forum Component, http://code.google.com/p/django-forum/
[24] Satchmo, http://www.satchmoproject.com/
[25] Scott, D., Sharp, R.: Abstracting Application-level Web Security. In: 11th World Wide Web Conference (2002)
[26] WhiteHat Security. Web Application Security 101 (2005), http://www.whitehatsec.com/articles/webappsec101.pdf
[27] Su, Z., Wassermann, G.: The Essence of Command Injection Attacks in Web Applications. In: Symposium on Principles of Programming Languages (2006)
[28] Sun. Java Servlets (2008), http://java.sun.com/products/servlet/
[29] Tenable Network Security. Nessus Open Source Vulnerability Scanner Project (2008), http://www.nessus.org/
[30] Twill. Twill: A Simple Scripting Language for Web Browsing (2008), http://twill.idyll.org/
[31] Web Application Attack and Audit Framework, http://w3af.sourceforge.net/
[32] Xie, Y., Aiken, A.: Static Detection of Security Vulnerabilities in Scripting Languages. In: 15th USENIX Security Symposium (2006)

Model-Based Covert Timing Channels: Automated Modeling and Evasion

Steven Gianvecchio[1], Haining Wang[1], Duminda Wijesekera[2], and Sushil Jajodia[2]

[1] Department of Computer Science
College of William and Mary, Williamsburg, VA 23187, USA
{srgian,hnw}@cs.wm.edu
[2] Center for Secure Information Systems
George Mason University, Fairfax, VA 22030, USA
{dwijesek,jajodia}@gmu.edu

Abstract. The exploration of advanced covert timing channel design is important to understand and defend against covert timing channels. In this paper, we introduce a new class of covert timing channels, called model-based covert timing channels, which exploit the statistical properties of legitimate network traffic to evade detection in an effective manner. We design and implement an automated framework for building model-based covert timing channels. Our framework consists of four main components: filter, analyzer, encoder, and transmitter. The filter characterizes the features of legitimate network traffic, and the analyzer fits the observed traffic behavior to a model. Then, the encoder and transmitter use the model to generate covert traffic and blend with legitimate network traffic. The framework is lightweight, and the overhead induced by model fitting is negligible. To validate the effectiveness of the proposed framework, we conduct a series of experiments in LAN and WAN environments. The experimental results show that model-based covert timing channels provide a significant increase in detection resistance with only a minor loss in capacity.

Keywords: covert timing channels, traffic modeling, evasion.

1 Introduction

A covert channel is a "communication channel that can be exploited by a process to transfer information in a manner that violates a system's security policy" [1]. There are two types of covert channels: covert storage channels and covert timing channels. A covert storage channel manipulates the contents of a storage location (e.g., disk, memory, packet headers, etc.) to transfer information. A covert timing channel manipulates the timing or ordering of events (e.g., disk accesses, memory accesses, packet arrivals, etc.) to transfer information. The focus of this paper is on covert timing channels.

The potential damage of a covert timing channel is measured in terms of its capacity. The capacity of covert timing channels has been increasing with the development of high-performance computers and high-speed networks. While covert timing channels studied in the 1970s could transfer only a few bits per second [2], covert timing channels in modern computers can transfer several megabits per second [3]. To defend against covert timing channels, researchers have proposed various methods to detect and disrupt them. The disruption of covert timing channels manipulates traffic to slow or stop covert timing channels [4,5,6,7,8]. The detection of covert timing channels mainly uses statistical tests to differentiate covert traffic from legitimate traffic [9,10,11,12,13]. Such detection methods are somewhat successful, because most existing covert timing channels cause large deviations in the timing behavior from that of normal traffic, making them relatively easy to detect.

In this paper, we introduce model-based covert timing channels, which endeavor to evade detection by modeling and mimicking the statistical properties of legitimate traffic. We design and develop a framework for building model-based covert timing channels, in which hidden information is carried through pseudo-random values generated from a distribution function. We use the inverse distribution function and cumulative distribution function for encoding and decoding. The framework includes four components, filter, analyzer, encoder, and transmitter. The filter profiles the legitimate traffic, and the analyzer fits the legitimate traffic behavior to a model. Then, based on the model, the encoder chooses the appropriate distribution functions from statistical tools and traffic generation libraries to create covert timing channels. The distribution functions and their parameters are determined by automated model fitting. The process of model fitting proves very efficient and the induced overhead is minor. Lastly, the transmitter generates covert traffic and blends with legitimate traffic.

The two primary design goals of covert timing channels are high capacity and detection resistance. To evaluate the effectiveness of the proposed framework, we perform a series of LAN and WAN experiments to measure the capacity and detection resistance of our model-based covert timing channel. We estimate the capacity with a model and then validate the model with real experiments. Our experimental results show that the capacity is close to that of an optimal covert timing channel that transmits in a similar condition. In previous research, it is shown that the shape [9, 10] and regularity [11, 12] of network traffic are important properties in the detection of covert timing channels. We evaluate the detection resistance of the proposed framework using shape and regularity tests. The experimental results show that both tests fail to differentiate the model-based covert traffic from legitimate traffic. Overall, our model-based covert timing channel achieves strong detection resistance and high capacity.

There is an arms race between covert timing channel design and detection. To maintain the lead, researchers need to continue to improve detection methods and investigate new attacks. The goal of our work is to increase the understanding of more advanced covert timing channel design. We anticipate that our demonstration of model-based covert timing channels will ultimately lead to the development of more advanced detection methods.

The remainder of the paper is structured as follows. Section 2 surveys related work. Section 3 provides background information on covert timing channels and describes two base cases in their design. Section 4 details the design and implementation of the proposed framework. Section 5 validates the effectiveness of the model-based covert timing channel through live experiments over the Internet. Finally, we conclude the paper and discuss future directions in Section 6.

2 Related Work

To defend against covert timing channels, researchers have proposed different solutions to detect and disrupt covert traffic. The disruption of covert timing channels adds random delays to traffic, which reduces the capacity of covert timing channels but reduces the network performance as well. The detection of covert timing channels is mainly accomplished using statistical tests to differentiate covert traffic from legitimate traffic. While the focus of earlier work is on the disruption of covert timing channels [4,5,6,7,8], more recent research has begun to investigate the design and detection of covert timing channels [9,10,11,12,14].

Kang et al. [5] designed a device, known as "The Pump," which reduces the capacity of covert timing channels by disrupting the timing of communication. This device increases the number of errors by randomly manipulating the timing values. The basic version of "The Pump" is designed to address covert timing channels within systems. A network version was later designed and developed [6,7]. Giles et al. [8] studied the disruption of covert timing channels from a game theoretic perspective. The authors takes the point of view of both the jammer and the covert timing channel, and discusses the strategies for both optimal jammers and optimal input processes. Fisk et al. [4] investigated the concept of Active Wardens in relation to covert channels. The authors introduced the quantity of Minimal Requisite Fidelity (MRF), which is the minimum fidelity needed to support the communication channel, and proposed a system to identify and eliminate unneeded fidelity in traffic that could be used for covert channels.

Cabuk et al. [11] designed and implemented a simple covert timing channel and showed that the regularity of the covert timing channel can be used in its detection. To disrupt the regularity, the authors tried two approaches. The first is to change the timing intervals, which is still successfully detected. The second is to introduce noise in the form of legitimate traffic. However, the covert timing channel is still sometimes detected, even with 50% of the inter-packet delays being legitimate traffic. This covert timing channel has similar regularity test scores to Fixed-average Packet Rate (FPR) and OPtimal Capacity (OPC) (described in Section 3) but transmits information more slowly.

Berk et al. [9,10] proposed a scheme for detecting binary and multi-symbol covert timing channels. The detection method measures the distance between the mean and modes, with a large distance indicating a potential covert timing channel. The detection test assumes a normal distribution for the inter-packet delays and, as a result, is not applicable to the covert timing channels we discussed. The authors used the Arimoto-Blahut algorithm [15,16] in the binary case without considering the cost. In contrast, we use the Arimoto-Blahut algorithm in the multi-symbol case but with a cost constraint, to formulate the optimal input distribution for FPR.

Shah et al. [12] developed a keyboard device, called *JitterBug*, to create a loosely-coupled covert channel capable of leaking information typed on a keyboard over the network. Such a covert timing channel takes advantage of small delays in key-presses to affect the inter-packet delays of a networked application. As a result, the keyboard slowly leaks information to an observer outside of the network. The authors showed that the initial scheme leaves a regular pattern in the inter-packet delays, which can be removed by rotating the position of the window. The JitterBug transmits information much more slowly than our model-based covert timing channel, but does so under tighter constraints on the transmission mechanism.

Borders et al. [17] developed a system, called *Web Tap*, to detect covert tunnels in web traffic based on header fields, inter-request delays, request sizes, capacity usage, request regularity, and request time. Such a system is successful in detecting several spyware and backdoor programs. However, the technique used by our model-based covert timing channel to mimic the inter-request delays and request regularity of traffic, could be used by spyware and backdoor programs to evade the Web Tap.

While some recent research has taken steps to better hide covert timing channels [11,12], these works focus on removing regularity rather than making the covert timing channel look like legitimate traffic. Moreover, removing regularity is the last step in the covert channel design process, instead of a consideration up front. In contrast, our framework is designed from the ground up to provide high detection resistance. As a result, the proposed model-based covert timing channel is able to provide much stronger detection resistance than most practical implementations of covert timing channel presented in the literature.

There are recent works on using timing channels to watermark traffic [18,19] and on detecting such timing-based watermarks [20]. Wang et al. [18] developed a robust watermarking scheme for tracing encrypted attack traffic through stepping stones. The scheme, through the use of redundancy, can resist arbitrarily large timing perturbations, if there are a sufficient number of packets to watermark. Peng et al. [20] investigated how to detect such watermarks, as well as methods for removing or duplicating the watermarks. Yu et al. [19] developed a sophisticated technique for hiding watermarks by disguising them as pseudo-noise. There are some interesting differences between timing-based watermarking and traditional covert timing channels, such as the fact that the defender, not the attacker, uses the timing channel in the watermarking schemes.

3 Background

In this section, we describe basic communication concepts and relate them to covert timing channels. Then, based on these concepts, we formulate two base cases in covert timing channel design. The basic problem of communication, producing a message at one point and reproducing that message at another point, is the same for both overt and covert channels, although covert channels must consider the additional problem of hiding communication.

3.1 Basic Communication Concepts

The capacity of a communication channel is the maximum rate that it can reliably transmit information. The capacity of a covert timing channel is measured in bits per time unit [21]. The capacity in bits per time unit C_t is defined as:

$$C_t = \max_X \frac{I(X;Y)}{E(X)},$$

where X is the transmitted inter-packet delays or input distribution, Y is the received inter-packet delays or output distribution, $I(X;Y)$ is the mutual information between X and Y, and $E(X)$ is the expected time of X.

The mutual information measures how much information is carried across the channel from X to Y. The mutual information $I(X;Y)$ is defined as:

$$I(X;Y) = \begin{cases} \sum\limits_X \sum\limits_Y P(y \mid x)P(x)\log\frac{P(y|x)P(x)}{P(x)P(y)}, \text{(discrete)} \\ \int\limits_X \int\limits_Y P(y \mid x)P(x)\log\frac{P(y|x)P(x)}{P(x)P(y)}\, dx\, dy, \text{(continuous)} \end{cases}$$

The noise, represented by the conditional probability in the above definitions, is defined as:

$$P(y \mid x) = f_{noise}(y,x),$$

where f_{noise} is the noise probability density function, x is the transmitted inter-packet delays, and y is the received inter-packet delays.

The noise distribution f_{noise} is the probability that the transmitted inter-packet delay x results in the received inter-packet delay y. The specific noise distribution for inter-packet delays is detailed in Section 5.2.

3.2 Base Cases in Design

The two main goals of covert timing channel design are high capacity and detection resistance. There are few examples of practical implementations of covert timing channels in the literature, so we begin to explore the design space in terms of both capacity and detection resistance. The focus of our model-based covert timing channel is to achieve high detection resistance. In the following section, we formulate two base cases in covert channel design as comparison to the model-based covert timing channel.

The first case, optimal capacity, transmits as much information as possible, sending hundreds or more packets per second. Such a design might not be able to achieve covert communication, but is useful as a theoretical upper bound. The second case, fixed average packet rate, sends packets at a specific fixed average packet rate, encoding as much information per packet as possible. The fixed average packet rate is mainly determined by the packet rate of legitimate traffic.

Optimal Capacity Channel. The first design, OPtimal Capacity (OPC), uses the discrete input distribution that transmits information as fast as possible. The optimal capacity is dependent on the optimal distance between two symbols. The first symbol is (approximately) zero and the second symbol is non-zero, so the use of more symbols (i.e., four or eight) will introduce more non-zero symbols and decrease the symbol rate. The use of smaller distances between the two symbols increases the symbol rate and the error rate. The optimal distance is the point at which the increase in error rate balances the increase in symbol rate.

The code operates based on two functions. The encode function is defined as:

$$F_{encode}(s) = d_s = \begin{cases} 0, & s = 0 \\ d, & s = 1 \end{cases}$$

where s is a symbol, d_s is an inter-packet delay with a hidden symbol s, and d is the optimal distance between the two symbols. The decode function is defined as:

$$F_{decode}(d_s) = s = \begin{cases} 0, & d_s < \frac{1}{2}d \\ 1, & \frac{1}{2}d \leq d_s \end{cases}$$

where d_s is an inter-packet delay with a hidden symbol s.

Channel Capacity: The channel capacity of OPC is dependent on the optimal input distribution and noise. The input distribution is defined as:

$$P(x) = \begin{cases} p, & x = d \\ 1 - p, & x = 0 \\ 0, & \text{otherwise} \end{cases}$$

where p is the probability of the symbol $s = 1$, and $1 - p$ is the probability of the symbol $s = 0$.

Therefore, the capacity of OPC is the maximum of the mutual information with respect to the parameters d and p of the input distribution over the expected time $d \cdot p$:

$$C_t = \max_{d,p} \frac{1}{d \cdot p} \sum_X \sum_Y P(y \mid x)P(x) \log \frac{P(y \mid x)P(x)}{P(x)P(y)}.$$

Fixed-Average Packet Rate Channel. The second design, Fixed-average Packet Rate (FPR), uses the input distribution that encodes as much information per packet as possible with a constraint on the average cost of symbols. The cost is measured in terms of the time required for symbol transmission. Therefore, the optimal input distribution is subject to the constraint on the average packet rate, i.e., the cost of symbol transmission.

The optimal input distribution for FPR is computed with the Arimoto-Blahut algorithm generalized for cost constraints [16]. The Arimoto-Blahut algorithm computes the optimal input distribution for capacity in bits per channel usage. The capacity in bits per channel usage C_u is defined as:

$$C_u = \max_X I(X;Y).$$

In general, C_u and C_t do not have the same input distribution X. However, if the input distribution is constrained so that $E(X) = c$ (where c is a constant), then the optimal input distribution X is optimal for both C_u and C_t, and $C_u = C_t \cdot c$. Thus, FPR transmits as much information per packet (channel usage) and per second (time unit) as possible with a fixed average packet rate. We use the Arimoto-Blahut algorithm to compute the optimal input distribution for FPR. The capacity results for FPR, based on the Arimoto-Blahut algorithm, are detailed in Section 5.

4 The Framework

The covert timing channel framework, as shown in Figure 1, is a pipeline that filters and analyzes legitimate traffic then encodes and transmits covert traffic. As the output of the pipeline, the covert traffic mimics the observed legitimate traffic, making it easy to evade detection. The components of the framework include filter, analyzer, encoder, and transmitter, which are detailed in the following paragraphs.

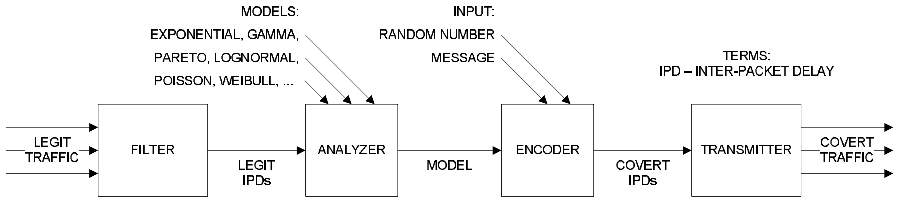


Fig. 1. Framework for building model-based covert timing channels

The filter monitors the background traffic and singles out the specific type of traffic to be mimicked. The more specific application traffic the filter can identify and profile, the better model we can have for generating covert traffic. For example, FTP is an application protocol based on TCP, but generating a series of inter-packet delays based on a model of all TCP traffic would be a poor model for describing FTP behaviors. Once the specified traffic is filtered, the traffic is further classified into individual flows based on source and destination IP addresses. The filter then calculates the inter-packet delay between subsequent pair of packets from each flow, and forwards the results to the analyzer.

The analyzer fits the inter-packet delays in sets of 100 packets with the Exponential, Gamma, Pareto, Lognormal, Poisson, and Weibull distributions. The

Table 1. The scores for different models for a sample of HTTP inter-packet delays

model	parameters	root mean squared error
Weibull	0.0794, 0.2627	0.0032
Gamma	0.1167, 100.8180	0.0063
Lognormal	-4.3589, 3.5359	0.0063
Pareto	3.6751, 0.0018	0.0150
Poisson	11.7611	0.0226
Exponential	11.7611	0.0294

fitting process uses maximum likelihood estimation (MLE) to determine the parameters for each model. The model with the smallest root mean squared error (RMSE), which measures the difference between the model and the estimated distribution, is chosen as the traffic model. The model selection is automated. Other than the set of models provided to the analyzer, there is no human input. The models are scored based on root mean squared errors, as shown in Table 1. The model with the lowest root mean squared error is the closest to the data being modeled. Since most types of network traffic are non-stationary [22], the analyzer supports piecewise modeling of non-stationary processes by adjusting the parameters of the model after each set of 100 covert inter-packet delays. The analyzer refits the current model with new sets of 100 packets to adjust the parameters. The analyzer can take advantage of a larger selection of models to more accurately model different types of application traffic. For example, if we know that the targeted traffic is well-modeled as an Erlang distribution, we will add this distribution to the set of models. For each of the current models, the computational overhead is less than 0.1 milliseconds and the storage overhead for the executable is less than 500 bytes, so the induced resource consumption for supporting additional models is not an issue.

The filter and analyzer can be run either offline or online. In the offline mode, the selection of the model and parameters is based on traffic samples. The offline mode consumes less resources, but the model might not represent the current network traffic behavior well. In the online mode, the selection of the model and parameters is based on live traffic. The online mode consumes more resources and requires that the model and parameters be transmitted to the decoder with the support of a startup protocol, but the model better represents the current network traffic behavior. The startup protocol is a model determined in advance, and is used to transmit the online model (1 byte) and parameters (4-8 bytes) to the decoder.

The encoder generates random covert inter-packet delays that mimic legitimate inter-packet delays. The input to the encoder includes the model, the message, and a sequence of random numbers. Its output is a sequence of covert random inter-packet delays. The message to be sent is separated into symbols. The symbols map to different random timing values based on a random code that distributes symbols based on the model.

Using a sequence of random numbers r_1, r_2, ..., r_n., we transform the discrete symbols into continuous ones. The continuization function is

$$F_{continuize}(s) = (\frac{s}{\mid S \mid} + r) \bmod 1 = r_s,$$

where S is the set of possible symbols, s is a symbol and r is a Uniform(0,1) random variable. The corresponding discretization function is:

$$F_{discretize}(r_s) = \mid S \mid \cdot ((r_s - r) \bmod 1) = s,$$

where r_s is a Uniform(0,1) random variable with a hidden symbol s.

The encoder and decoder start with the same seed and generate the same sequence of random numbers, r_1, r_2, ..., r_n. To maintain synchronization, the encoder and decoder associate the sequence of symbols with TCP sequence numbers, i.e., s_1 with the first TCP sequence number, s_2 with the second TCP sequence number, and so on. [1] Therefore, both the encoder and decoder have the same values of r through the sequence of symbols. The inverse distribution function F^{-1}_{model} takes a Uniform(0,1) random number as input and generates a random variable from the selected model as output. The sequence of transformed random numbers r_{s1}, r_{s2}, ..., r_{sn} is used with the inverse distribution function to create random covert inter-packet delays d_{s1}, d_{s2}, ..., d_{sn}. The encode function is:

$$F_{encode} = F^{-1}_{model}(r_s) = d_s,$$

where F^{-1}_{model} is the inverse distribution function of the selected model. The decode function is:

$$F_{decode} = F_{model}(d_s) = r_s,$$

where F_{model} is the cumulative distribution function of the selected model, and d_s is a random covert inter-packet delay with a hidden symbol s.

The transmitter sends out packets to produce the random covert inter-packet delays d_{s1}, d_{s2}, ..., d_{sn}. The receiver then decodes and discretizes them to recover the original symbols s_1, s_2, ..., s_n.

4.1 Model-Based Channel Capacity

The model-based channel capacity is also dependent on the input distribution and noise. The input distribution is defined as:

$$P(x) = f_{model}(x)$$

where f_{model} is the probability density function of the selected model.

Therefore, the capacity of the model-based channel is the mutual information over the expected time $E(X)$:

$$C_t = \frac{1}{E(X)} \int_X \int_Y P(y \mid x) P(x) \log \frac{P(y \mid x) P(x)}{P(x) P(y)}.$$

[1] With this mechanism, repacketization can cause synchronization problems, so other mechanisms such as "bit stuffing" [12] could be useful for synchronization.

4.2 Implementation Details

We implement the proposed framework using `C` and `MATLAB` in Unix/Linux environments. The components run as user-space processes, while access to `tcpdump` is required. The filter is written in C and runs `tcpdump` with a user-specified filtering expression to read the stream of packets. The filter processes the traffic stream and computes the inter-packet delays based on the packet timestamps. The analyzer is written in `MATLAB` and utilizes the fitting functions from the statistics toolbox for maximum likelihood estimation.

The encoder is written in `C`, and uses random number generation and random variable models from the Park-Leemis [23] simulation C libraries. The transmitter is also written in `C`, with some inline assembly, and uses the Socket API. The timing mechanism used is the Pentium CPU Time-Stamp Counter, which is accessed by calling the `RDTSC` (Read Time-Stamp Counter) instruction. The `RDTSC` instruction has excellent resolution and low overhead, but must be calibrated to be used as a general purpose timing mechanism. The `usleep` and `nanosleep` functions force a context switch, which delays the packet transmission with small inter-packet delays, so these functions are not used.

5 Experimental Evaluation

In this section, we evaluate the effectiveness of a model-based covert timing channel built from our framework. The OPC and FPR covert timing channels, discussed in Section 3, are used as points of comparison. In particular, we examine the capacity and detection resistance of each covert timing channel.

5.1 Experimental Setup

The defensive perimeter of a network, composed of firewalls and intrusion detection systems, is responsible for protecting the network. Typically, only a few specific application protocols, such as HTTP and SMTP, are commonly allowed to pass through the defensive perimeter. We utilize outgoing HTTP inter-packet delays as the medium to build model-based covert timing channels, due to the wide acceptance of HTTP traffic for crossing the network perimeter. We refer to the model-based HTTP covert timing channel as MB-HTTP.

Testing Scenarios. There are three different testing scenarios in our experimental evaluation. The first scenario is in a LAN environment, a medium-size campus network with subnets for administration, departments, and residences. The LAN connection is between two machines, located in different subnets. The connection passes through several switches, the routers inside the campus network, and a firewall device that protects each subnet.

The other two scenarios are in WAN environments. The first WAN connection is between two machines, both are on the east coast of the United States but in different states. One is on a residential cable network and the other is on

Table 2. The network conditions of each test scenario

	LAN	WAN E-E	WAN E-W
distance	0.3 miles	525 miles	2660 miles
RTT	1.766ms	59.647ms	87.236ms
IPDV	2.5822e-05	2.4124e-03	2.1771e-04
hops	3	18	13
IPDV - inter-packet delay variation			

a medium-size campus network. The second WAN connection is between two machines on the opposite coasts of the United States, one on the east coast and the other on the west coast. Both machines are on campus networks.

The network conditions for different experiment scenarios are summarized in Table 2. The two-way round-trip time (RTT) is measured using the ping command. We compute the one-way inter-packet delay variation based on the delays between packets leaving the source and arriving at the destination. The inter-packet delay variations of the three connections span three orders of magnitude, from 1×10^{-3} to 1×10^{-5}. The LAN connection has the lowest inter-packet delay variation and the two WAN connections have higher inter-packet delay variation, as expected. The WAN E-E connection is shorter and has smaller RTT time than the WAN E-W connection. However, WAN E-E has higher inter-packet delay variation than WAN E-W, due to more traversed hops. This implies that the inter-packet delays variation is more sensitive to the number of hops than the physical distance and RTT between two machines.

Building MB-HTTP. We install the components of the framework on the testing machines. The filter distinguishes the outgoing HTTP traffic from background traffic. The analyzer observes 10 million HTTP inter-packet delays, then fits the HTTP inter-packet delays to the models, as described in Section 4. The fitting functions use maximum likelihood estimation (MLE) to determine the parameters for each model. The model with the best root mean squared error (RMSE), a measure of the difference between the model and the distribution being estimated, is chosen as the traffic model.

For the HTTP inter-packet delays, the analyzer selects the Weibull distribution based on the root mean squared error. Note that HTTP inter-packet delays have been shown to be well approximated by a Weibull distribution [22]. The Weibull probability distribution function is:

$$f(x, \lambda, k) = \frac{k}{\lambda}(\frac{x}{\lambda})^{(k-1)} e^{-(\frac{x}{\lambda})^k}.$$

The parameters, which vary for each set of 100 packets, have a mean scale parameter λ of 0.0371 and a mean shape parameter k of 0.3010. With these parameters, the mean inter-packet delay is 0.3385, approximately 3 packets per second.

Table 3. The mean packets per second and mean inter-packet delay for OPC

channel	LAN		WAN E-E		WAN E-W	
	PPS	IPD	PPS	IPD	PPS	IPD
OPC	12,777.98	7.87e-05	137.48	7.31e-03	1,515.56	6.63e-04
PPS - mean packets per second, IPD - mean inter-packet delay						

Formulating OPC and FPR. The average packet rate for FPR is fixed at $\frac{1}{0.3385} = 2.954$ packets per second, based on the average packet rate of HTTP traffic. We use the Arimoto-Blahut algorithm to compute the optimal input distribution, with the average packet rate of 2.954 as the cost constraint. The optimal input distribution balances high cost symbols with low probabilities and low cost symbols with high probabilities, such that the average cost constraint is satisfied. The constraint can be satisfied for infinitely large symbols with infinitely small probabilities, and hence, the optimal input distribution decays exponentially to infinity. The results of the Arimoto-Blahut algorithm, as the number of intervals increases, reduce to an Exponential distribution with an inverse scale parameter of $\lambda = 2.954$. The Exponential probability distribution function is:

$$f(x, \lambda) = \lambda e^{-\lambda x}.$$

We compute the optimal distance between packets for OPC based on the noise distribution. The optimal distance between packets and the average packet rate for OPC is shown in Table 3. For connections with higher inter-packet delay variation, OPC increases the time elapse between packets to make the inter-packet delays easier to distinguish, and, as a result, lowers the average number of packets per second.

5.2 Capacity

The definition of capacity allows us to estimate the capacity of each covert timing channel based on the network conditions of each connection. In previous research [24], the inter-packet delay differences have been shown to be well-modeled by a Laplace distribution. The probability density function of the Laplace distribution is:

$$f(x, \mu, b) = \frac{1}{2b} e^{-\frac{|x-\mu|}{b}}.$$

The setting of the scale parameter b is based on the inter-packet delay variation for each connection. The variation of the Laplace distribution is $\sigma^2 = 2b^2$. Therefore, we set b to:

$$b = \sqrt{\frac{1}{2}\sigma^2},$$

where σ^2 is the inter-packet delay variation for each connection.

Table 4. The theoretical capacity of each covert timing channel

channel	LAN		WAN E-E		WAN E-W	
	CPP	CPS	CPP	CPS	CPP	CPS
MB-HTTP	9.39	27.76	4.12	12.19	6.84	20.21
FPR	12.63	37.32	6.15	18.17	9.59	28.35
OPC	0.50	6395.39	0.50	68.80	0.50	758.54
CPP - capacity per packet, CPS - capacity per second						

The results, in terms of capacity per packet and capacity per second, are shown in Table 4. While OPC has the highest capacity, it is the least efficient in terms of capacity per packet. Furthermore, with the large number of packets per second, it can be easily detected by most intrusion detection systems.

The capacity of MB-HTTP is 67% to 74% of that of FPR, with larger differences for connections with high inter-packet delay variation than for those with low inter-packet delay variation. The Weibull distribution has a larger proportion of very small values than the Exponential distribution. As a result, MB-HTTP uses more small values than FPR and benefits more from lower inter-packet delay variation.

The theoretical capacity is somewhat optimistic. The model only considers the noise introduced after packets leave the transmitter. With the real covert timing channels, noise is introduced before packets leave the transmitter. The transmitter is sometimes not able to transmit at the appropriate times, due to slow processing, context switches, etc. Thus, the actual distance between packets can increase or decrease from the intended distance as the packets are transmitted.

Empirical Capacity. To evaluate the channel capacity in practice, we run covert timing channels on each connection. The channels are configured to transmit 16,000 random bits of information. For FPR and MB-HTTP, the number of bits encoded per packet is set to 16 (i.e., $2^{16} = 65,536$ different values), while OPC transmits a single bit per packet.

During these tests, we measure the bit error rate of each covert timing channel from the most significant bit to the least significant bit of each packet. The most significant bit represents a large part of the inter-packet delay, where the least significant bit represents a small part of the inter-packet delay. While flipping the most significant bit causes a difference in seconds or tenths of seconds, changing the least significant bit means a difference only in milliseconds or microseconds. In other words, the higher the number of bits encoded per packet, the smaller the precision of the lowest order bits. Interestingly, encoding at 16 bits per packet and decoding at 8 bits per packet produces the most significant 8 bits of the 16 bit code.

To determine the transmission rate with error correction, we measure the empirical capacity of each bit as a binary symmetric channel. The binary symmetric channel is a special case where the channel has two symbols of equal probability.

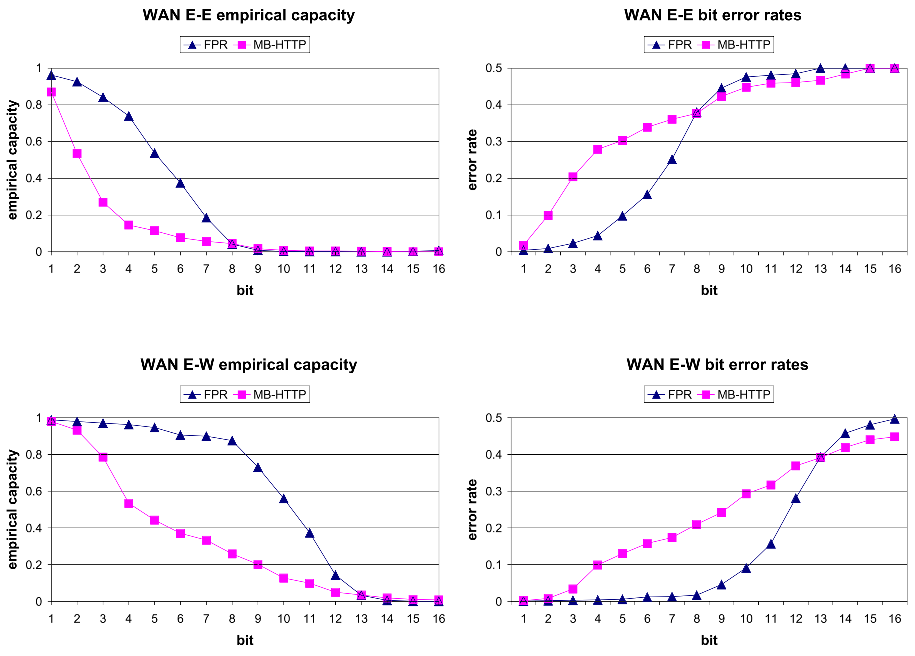


Fig. 2. The empirical capacity and bit error rates for WAN E-E and WAN E-W

The capacity of a binary symmetric channel is:

$$C = I(X;Y) = 1 - (p \log p + q \log q),$$

where p is the probability of a correct bit and $q = 1 - p$ is the probability of an incorrect bit.

The empirical capacity and bit error rate for each bit, from the most significant to the least significant, are shown in Figure 2. The empirical capacity per bit degrades as the bit error rates increase. The total capacity of the channel is the summation of the capacity for each bit. For MB-HTTP, the bit error rate increases somewhat linearly. For FPR, the bit error rate accelerates gradually, eventually overtaking the bit error rates of MB-HTTP, though at this point the capacity per bit is insignificant.

The empirical capacity of each covert timing channel is shown in Table 5. The empirical capacity of MB-HTTP is still about 46% to 61% of that of FPR, somewhat lower than the case in the theoretical model. This is because a larger proportion of MB-HTTP traffic has small inter-packet delays than that of FPR, and small inter-packet delays are more sensitive to noise caused by transmission delays (i.e., slow processing, context switches, etc.) than large inter-packet delays, which is not represented in the theoretical model.

Table 5. The empirical capacity of each covert timing channel

	LAN		WAN E-E		WAN E-W	
channel	ECPP	ECPS	ECPP	ECPS	ECPP	ECPS
MB-HTTP	6.74	19.93	2.15	6.35	5.18	15.31
FPR	10.95	32.35	4.63	13.67	9.37	27.69
OPC	0.85	10,899.62	0.66	91.28	0.98	1,512.53
ECPP - empirical capacity per packet, ECPS - empirical capacity per second						

5.3 Detection Resistance

The detection resistance, as described in Section 3, is estimated based on the shape and regularity tests. To examine the shape of the distribution, we use the Kolmogorov-Smirnov test [25], which is a non-parametric goodness-of-fit test. To examine the regularity of the traffic, we use the regularity test [11], which studies the variance of the traffic pattern. In this section, we detail these two tests and show the detection resistance of MB-HTTP against both tests.

Shape Tests. The two-sample Kolmogorov-Smirnov test determines whether or not two samples come from the same distribution. The Kolmogorov-Smirnov test is distribution free, meaning the test is not dependent on a specific distribution. Thus, it is applicable to a variety of types of traffic with different distributions. The Kolmogorov-Smirnov test statistic measures the maximum distance between two empirical distribution functions.

$$KSTEST = \max \mid S_1(x) - S_2(x) \mid,$$

where S_1 and S_2 are the empirical distribution functions of the two samples.

In our experiments, we test a large set of legitimate inter-packet delays against a sample of either covert or legitimate inter-packet delays. The large set is a training set of 10,000,000 HTTP inter-packet delays. The training set is used to represent the normal behavior of the HTTP protocol.

The test score by comparing the two sets is used to determine if the sample is covert or legitimate. A small score indicates that the behavior is close to normal. However, if the test score is large, i.e., the sample does not fit the normal behavior of the protocol, it indicates a potential covert timing channel.

Table 6. The mean and standard deviation of the Kolmogorov-Smirnov test scores

	LEGIT-HTTP		MB-HTTP		FPR		OPC	
sample size	mean	stdev	mean	stdev	mean	stdev	mean	stdev
100x 2,000	.193	.110	.196	.093	.925	.002	.999	.000
100x 10,000	.141	.103	.157	.087	.925	.001	.999	.000
100x 50,000	.096	.088	.122	.073	.924	.000	.999	.000
100x 250,000	.069	.066	.096	.036	.924	.000	.999	.000

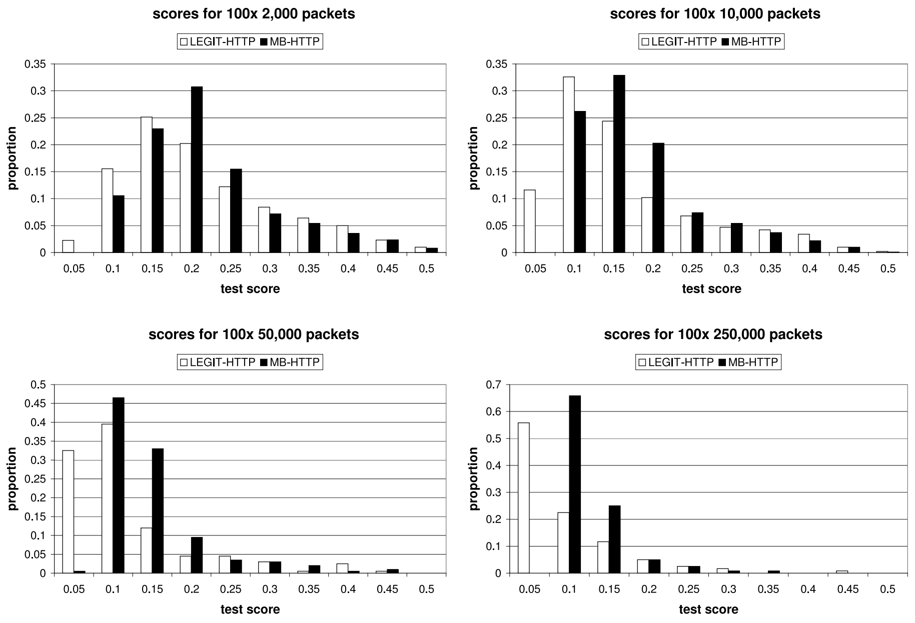


Fig. 3. The distribution of Kolmogorov-Smirnov test scores

The Kolmogorov-Smirnov test is run 100 times for each of 2,000, 10,000, 50,000, and 250,000 packet samples of legitimate and covert traffic from each covert timing channel. The mean and standard deviation of the test scores are shown in Table 6. For FPR and OPC, the mean scores are over 0.90 and the standard deviations are extremely low, indicating that the test can reliably differentiate both covert timing channels from normal HTTP traffic. By contrast, the mean scores for MB-HTTP samples are very close to those of legitimate samples. The mean scores are for 100 tests, which in total include as many as 25 million (250,000 x 100) inter-packet delays. The distribution of individual test scores is illustrated in Figure 3.

The detection resistance based on the Kolmogorov-Smirnov test is shown in Table 7. The targeted false positive rate is 0.01. To achieve this false positive rate, the cutoff scores—the scores that decide whether samples are legitimate or

Table 7. The false positive and true positive rates for the Kolmogorov-Smirnov test

		LEGIT-HTTP	MB-HTTP	FPR	OPC
sample size	cutoff	false pos.	true pos.	true pos.	true pos.
100x 2,000	$KSTEST \geq .66$	.01	.01	1.00	1.00
100x 10,000	$KSTEST \geq .65$	.01	.01	1.00	1.00
100x 50,000	$KSTEST \geq .41$	.01	.01	1.00	1.00
100x 250,000	$KSTEST \geq .21$	.01	.02	1.00	1.00

covert—are set at the 99th percentile of legitimate sample scores. The true positive rates, based on the cutoff scores, are then shown for each covert timing channel. Since the true positive rates in all 100 tests are 1.00, the Kolmogorov-Smirnov test detects FPR and OPC easily. However, the true positive rates for MB-HTTP are approximately the same as the false positive rates. The Kolmogorov-Smirnov test cannot differentiate between MB-HTTP and legitimate samples. Such a result can be explained based on the distribution of individual test scores, which is shown in Figure 3. While the mean scores of MB-HTTP traffic in Table 6 are slightly higher than those of LEGIT-HTTP, the distributions of individual scores overlap so that the false positive rate and true positive rate are approximately equal.

Regularity Tests. The regularity test [11] determines whether the variance of the inter-packet delays is relatively constant or not. This test is based on the observation that for most types of network traffic, the variance of the inter-packet delays changes over time. With covert timing channels, the code used to transmit data is a regular process and, as a result, the variance of the inter-packet delays remains relatively constant over time.

In our experiments, we test the regularity of a sample of either covert or legitimate inter-packet delays. The sample is separated into sets of w inter-packet delays. Then, for each set, the standard deviation of the set σ_i is computed. The regularity is the standard deviation of the pairwise differences between each σ_i and σ_j for all sets $i < j$.

$$regularity = STDEV(\frac{\mid \sigma_i - \sigma_j \mid}{\sigma_i}, i < j, \forall i, j)$$

The regularity test is run 100 times for 2,000 packet samples of legitimate and covert samples from each covert timing channel. The window sizes of $w = 100$ and $w = 250$ are used. The mean regularity scores are shown in Table 8. If the regularity is small, the sample is highly regular, indicating a potential covert timing channel.

Table 8. The mean of the regularity test scores

sample size	LEGIT-HTTP	MB-HTTP	FPR	OPC
100x 2,000 w=100	43.80	38.21	0.34	0.00
100x 2,000 w=250	23.74	22.87	0.26	0.00

The mean regularity scores for OPC are 0.0 for both tests, indicating regular behavior. There are two values, each with 0.5 probability. Therefore, the standard deviation within sets is small $\sigma = 0.5d = 3.317e - 4$, and there is no detectable change in the standard deviation between sets. The mean regularity score for FPR is small as well, showing that the test is able to detect the regular behavior. While the standard deviation of FPR, which is based on the Exponential distribution, is $\sigma = \lambda = 0.3385$, the code is a regular process, so the variance of the inter-packet delays remains relatively constant.

The mean regularity scores for MB-HTTP are close to those of legitimate samples. This is because the parameters are recalibrated after each set of 100 packets, as described in Section 4. The parameters of the distribution determine the mean and standard deviation, so adjusting the parameters changes the variance after each set of 100 inter-packet delays. As a result, like legitimate traffic, the variance of the inter-packet delays appears irregular.

Table 9. The false positive and true positive rates for the regularity test

		LEGIT-HTTP	MB-HTTP	FPR	OPC
sample size	cutoff	false pos.	true pos.	true pos.	true pos.
100x 2,000 w=100	$reg. \leq 6.90$	.01	.00	1.00	1.00
100x 2,000 w=250	$reg. \leq 5.20$	.01	.00	1.00	1.00

The detection resistance based on the regularity test is shown in Table 9. The targeted false positive rate is 0.01. The cutoff scores are set at the 1st percentile of legitimate sample scores, in order to achieve this false positive rate. The true positive rates, based on the cutoff scores, are then shown for each covert timing channels. The regularity test is able to detect FPR and OPC in all 100 tests. The resulting true positive rates for MB-HTTP are approximately the same as the false positive rate. Basically, the test is no better than random guessing at detecting MB-HTTP.

6 Conclusion

We introduced model-based covert timing channels, which mimic the observed behavior of legitimate network traffic to evade detection. We presented a framework for building such model-based covert timing channels. The framework consists of four components: filter, analyzer, encoder, and transmitter. The filter characterizes the specific features of legitimate traffic that are of interest. The analyzer fits the traffic to several models and selects the model with the best fit. The encoder generates random covert inter-packet delays that, based on the model, mimic the legitimate traffic. The transmitter then manipulates the timing of packets to create the model-based covert timing channel.

Using channel capacity and detection resistance as major metrics, we evaluated the proposed framework in both LAN and WAN environments. Our capacity results suggest that model-based covert timing channels work efficiently even in the coast-to-coast scenario. Our detection resistance results show that, for both shape and regularity tests, covert traffic is sufficiently similar to legitimate traffic that current detection methods cannot differentiate them. In contrast, the Kolmogorov-Smirnov and regularity tests easily detect FPR and OPC.

Our future work will further explore the detection of model-based covert timing channels. There are other non-parametric goodness-of-fit tests, such as the Anderson-Darling and Cramer-Von Mises tests [25], that are less general than the Kolmogorov-Smirnov test but might be more effective in measuring certain types of traffic. We will also further consider the regularity test at different levels of granularity. We believe that a scheme capable of detecting model-based covert timing channels will be effective in detecting other types of covert timing channels as well.

Acknowledgments

We would like to thank Cheng Jin and Lachlan Andrew at CalTech for assisting us in the coast-to-coast experiments. We also thank the anonymous reviewers for their insightful comments. This work was partially supported by NSF grants CNS-0627340 and CNS-0627493.

References

1. Department of Defense, U.S.: Trusted computer system evaluation criteria (1985)
2. Lampson, B.W.: A note on the confinement problem. Communications of the ACM 16(10) (October 1973)
3. Wang, Z., Lee, R.: Covert and side channels due to processor architecture. In: Jesshope, C., Egan, C. (eds.) ACSAC 2006. LNCS, vol. 4186, Springer, Heidelberg (2006)
4. Fisk, G., Fisk, M., Papadopoulos, C., Neil, J.: Eliminating steganography in internet traffic with active wardens. In: Proc. of the 2002 International Workshop on Information Hiding (October 2002)
5. Kang, M.H., Moskowitz, I.S.: A pump for rapid, reliable, secure communication. In: Proc. of ACM CCS 1993 (November 1993)
6. Kang, M.H., Moskowitz, I.S., Lee, D.C.: A network version of the pump. In: Proc. of the 1995 IEEE Symposium on Security and Privacy (May 1995)
7. Kang, M.H., Moskowitz, I.S., Chincheck, S.: The pump: A decade of covert fun. In: Srikanthan, T., Xue, J., Chang, C.-H. (eds.) ACSAC 2005. LNCS, vol. 3740. Springer, Heidelberg (2005)
8. Giles, J., Hajek, B.: An information-theoretic and game-theoretic study of timing channels. IEEE Trans. on Information Theory 48(9) (September 2002)
9. Berk, V., Giani, A., Cybenko, G.: Covert channel detection using process query systems. In: Proc. of FLOCON 2005 (September 2005)
10. Berk, V., Giani, A., Cybenko, G.: Detection of covert channel encoding in network packet delays. Technical Report TR2005-536, Department of Computer Science, Dartmouth College, Hanover, NH., USA (August 2005)
11. Cabuk, S., Brodley, C., Shields, C.: IP covert timing channels: Design and detection. In: Proc. of ACM CCS (October 2004)
12. Shah, G., Molina, A., Blaze, M.: Keyboards and covert channels. In: Proc. of the 2006 USENIX Security Symposium (July–August, 2006)
13. Gianvecchio, S., Wang, H.: Detecting covert timing channels: An entropy-based approach. In: Proceedings of the 2007 ACM Conference on Computer and Communications Security (October 2007)
14. Luo, X., Chan, E.W.W., Chang, R.K.C.: Cloak: A ten-fold way for reliable covert communications. In: Biskup, J., López, J. (eds.) ESORICS 2007. LNCS, vol. 4734, pp. 283–298. Springer, Heidelberg (2007)

15. Arimoto, S.: An algorithm for computing the capacity of arbitrary discrete memoryless channels. IEEE Trans. on Information Theory 18(1) (January 1972)
16. Blahut, R.E.: Computation of channel capacity and rate-distortion functions. IEEE Trans. on Information Theory 18(4) (July 1972)
17. Borders, K., Prakash, A.: Web tap: Detecting covert web traffic. In: Proc. of ACM CCS 2004 (October 2004)
18. Wang, X., Reeves, D.S.: Robust correlation of encrypted attack traffic through stepping stones by manipulation of interpacket delays. In: Proc. of ACM CCS 2003 (October 2003)
19. Yu, W., Fu, X., Graham, S., Xuan, D., Zhao, W.: Dsss-based flow marking technique for invisible traceback. In: Proc. of the 2007 IEEE Symposium on Security and Privacy, Washington, DC, USA (May 2007)
20. Peng, P., Ning, P., Reeves, D.S.: On the secrecy of timing-based active watermarking trace-back techniques. In: Proc. of the 2006 IEEE Symposium on Security and Privacy (May 2006)
21. Moskowitz, I.S., Kang, M.H.: Covert channels - here to stay? In: Proc. of the 1994 Annual Conf. on Computer Assurance (June 1994)
22. Cao, J., Cleveland, W.S., Lin, D., Sun, D.X.: On the nonstationarity of internet traffic. In: Proc. of SIGMETRICS/Performance 2001 (June 2001)
23. Leemis, L., Park, S.K.: Discrete-Event Simulation: A First Course. Prentice-Hall, Upper Saddle River (2006)
24. Zheng, L., Zhang, L., Xu, D.: Characteristics of network delay and delay jitter and its effect on oice over IP (VoIP). In: Proc. of the 2001 IEEE International Conf. on Communications (June 2001)
25. Duda, R., Hart, P., Stork, D.: Pattern Classification. Wiley-Interscience, New York (2001)

Optimal Cost, Collaborative, and Distributed Response to Zero-Day Worms - A Control Theoretic Approach

Senthilkumar G. Cheetancheri[1,⋆], John-Mark Agosta[2], Karl N. Levitt[1], Felix Wu[1], and Jeff Rowe[1]

[1] Security Lab, Dept. of Computer Science, Univ. of California, One Shields Ave., Davis, CA - 95616, USA
{cheetanc,levitt,wu,rowe}@cs.ucdavis.edu

[2] Intel Research. 2200, Mission College Blvd., Santa Clara, CA - 95052, USA
{john.m.agosta}@intel.com

Abstract. Collaborative environments present a happy hunting ground for worms due to inherent trust present amongst the peers. We present a novel control-theoretic approach to respond to zero-day worms in a signature independent fashion in a collaborative environment. A federation of collaborating peers share information about anomalies to estimate the presence of a worm and each one of them independently chooses the most cost-optimal response from a given set of responses. This technique is designed to work when the presence of a worm is uncertain. It is unique in that the response is dynamic and self-regulating based on the current environment conditions. Distributed Sequential Hypothesis Testing is used to estimate the extent of worm infection in the environment. Response is formulated as a Dynamic Programming problem with imperfect state information. We present a solution and evaluate it in the presence of an Internet worm attack for various costs of infections and response.

A major contribution of this paper is analytically formalizing the problem of optimal and cost-effective response to worms. The second contribution is an adaptive response design that minimizes the variety of worms that can be successful. This drives the attacker towards kinds of worms that can be detected by other means; which in itself is a success. Counter-intutive results such as leaving oneself open to infections being the cheapest option in certain scenarios become apparent with our response model.

Keywords: Worms, Collaboration, Dynamic Programming, Control Theory.

1 Introduction

Computer worms are a serious problem. Particularly in a collaborative environment, where the perimeter is quite secure but there is some amount of trust and implicit security within the environment. Once a worm breaks the perimeter

⋆ Corresponding Author.

defense, it essentially has a free run within the collaborative environment. An enterprise environment is a typical example of a network with this ‘crunchy on the outside – chewy on the inside’ characteristic. In this paper, we try to leverage the collaboration to collectively defend against such worm attacks. Dealing with known worms is a solved problem – signatures to be used by Intrusion Prevention Systems(IPSs) are developed to prevent further infections, and patches are developed to fix vulnerabilities exploited by these worms. Dealing with unknown worms – worms that exploit zero-day vulnerabilities or vulnerabilities for which patches have either not been generated or not applied yet – is still a research question. Several ingenious proposals to detect them automatically exist. Many sophisticated counter measures such as automatic signature generation and distribution [17,13,16,20] and automatic patch generation to fix vulnerabilities [18] have also been developed.

Often times, even if automated, there is not much time to either generate or distribute signatures or patches. Other times, system administrators are skeptical about applying patches. During instances when response based on the above mentioned techniques are not feasible, the only option left is to either completely shut-down the vulnerable service or run it risking infection. It is usually preferred to shut-down the service briefly until a mitigating response is engineered manually.

However, making a decision becomes hard when one is not certain if there is really a worm, and if the service being offered is vulnerable to it. It is not desirable to shut-down a service only to realize later that such an action was unwarranted because there is no worm. However, suspending the service in an attempt to prevent infection is not considered bad. Intuitively, it is desired to suspend the service briefly until it is clear whether there is an attack or not. Balancing the consequences of providing the service risking infection against that of not providing the service is of the essence.

This paper captures this intuition and devises an algorithm using Dynamic Programming(DP) techniques to minimize the overall cost of response to worms. Cost is defined as some mathematical expression of an undesirable outcome.

These algorithms use information about anomalous events that are potentially due to a worm from other co-operating peers to choose optimal response actions for local application. Such response can be later rolled-back in response to changes to the environment such as a curtailed worm. Since peers decide to implement response independently, the response is completely decentralized.

We surprisingly found that in certain scenarios, leaving oneself open to infection by the worm might be the least expensive option. We also show that these algorithms do not need large amounts of information to make decisions. One of the key achievements here is that we use weak Intrusion Detection Systems(IDSs) as sensors that have high false positive rates. By corroborating alerts raised by them with other collaborating sensors, we are able to minimize the false positives and achieve better fidelity in detecting worms.

2 Dynamic Programming

This section provides a brief introduction to the theory behind Dynamic Programming [6]. DP as applied to the current problem balances the low costs presently associated with operating a system against the undesirability of high future costs. The basic model of such a system is dynamic and discrete with an associated cost that is additive over time. The evolution of such a system can be described as:

$$x_{k+1} = f_k(x_k, u_k, w_k), \quad k = 0, 1, \ldots, N-1 \ , \tag{1}$$

where k indexes discrete time, x_k is the state of the system and summarizes past information that is relevant for future optimization, u_k is the control or decision variable to be selected at time k, w_k is a random parameter, also called disturbance or noise depending on the context, N is the horizon or the number of times control is applied and f_k is the mechanism by which the state is updated. The cost incurred at time k is denoted by $g_k(x_k, u_k, w_k)$, which is a random function because it depends on w_k. The goal is to minimize the total *expected cost*

$$J_\pi(x_0) = \underset{w_k}{E}\left\{ g_N(x_N) + \sum_{k=0}^{N-1} g_k(x_k, u_k, w_k) \right\} \ .$$

This is achieved by finding a sequence of functions called the *policy* or *control law*, $\pi = \{\mu_0, \ldots, \mu_{N-1}\}$, where each $\mu_k(x_k) \rightarrow u_k$ when applied to the system takes it from state x_k to x_{k+1} and minimizes the total *expected cost*. In general, for a given π, we use $J_k(x_k)$ to denote the *cost-to-go* from state x_k at time k to the final state at time N.

Dynamic Programming Algorithm: The optimal total cost is given by $J_0(x_0)$ in the last step of the following algorithm, which proceeds backwards in time from period $N-1$ to period 0:

$$J_N(x_N) = g_N(x_N), \tag{2}$$

$$J_k(x_k) = \min_{u_k} \underset{w_k}{E}\Big\{ g_k(x_k, u_k, w_k) + J_{k+1}(x_{k+1}) \Big\}, \quad k = 0, 1, \ldots, N-1 \ . \tag{3}$$

2.1 Imperfect Information Problems

DP problems as described above have perfect information about the state of the system, x_k. Often, x_k cannot be determined accurately; only an estimate,

$$z_k = h_k(x_k, v_k) \ , \tag{4}$$

can be made, where h_k is a sensor that maps x_k and a random disturbance v_k, into an observation, z_k. Such problems are solved by reformulating them into a perfect state information problem by introducing an augmented state variable I_k, which is a vector of the past observations and controls applied.

$$\begin{aligned} I_{k+1} &= (I_k, z_{k+1}, u_k), \quad k \ = \ 0, 1, \ldots, N \ - \ 2\,, \\ I_0 &= z_0 \ . \end{aligned} \tag{5}$$

3 Response Formulation with Imperfect State Information

In this section we formulate the computer worm response problem as a DP problem with imperfect state information. We assume that there could be only one worm and that the worm is a random scanning worm. We also assume that there is a sensor, such as an IDS albeit not very accurate. This DP formulation tells us which control should be applied to minimize the costs incurred until the worm detection process is complete.

3.1 Problem Statement

System Evolution: Consider a machine that provides some service. This machine needs to be operated for N steps or N time units. This machine can be in one of two states, P or $\overline{P}$, corresponding to the machine being in proper(desired state) or improper(infected by a worm) state respectively. During the course of operating the machine, it goes from state P to $\overline{P}$ with a certain probability λ and remains in state P with a probability $\overline{\lambda} = (1-\lambda)$. If the machine enters state $\overline{P}$, it remains there with probability 1. The infectious force λ, is an unknown quantity and depends on how much of the Internet is infected with the worm, if at all a worm is present.

Sensor: The machine also has a *sensor*, which inspects the machine for worm infections. However, it cannot determine the exact state of the machine. Rather, it can only determine the state of a machine with a certain probability. There are two possible observations; denoted by G (good, probably not infected) and B(bad, probably worm infected). Alternatively, instead of infections, we can imagine that the *sensor* looks for infection attempts and anomalies. The outcome would then indicate that there is probably a worm on the Internet (B) or not (G) as opposed to whether the host machine is infected or not. It is this latter interpretation we adopt for the rest of this paper. For the time being, let us assume that the inspections happen proactively at random intervals and also when alerts are received from peers. We also assume that the *sensor*'s integrity is not affected by the worm.

Controller: The machine also includes a *controller* that can continue(C) or stop(S) operating the machine. The machine cannot change states by itself if it is stopped. Thus the *controller* can stop the machine to prevent a worm infection and start it when it deems it safe to operate the machine. There are certain costs involved with each of these actions under different conditions as described in the next paragraph. The controller takes each action so that the overall cost of operating the machine for N steps is minimized.

Costs: Continuing(C) to operate the machine when it is in state P costs nothing. It is the nominal. We incur a cost of τ_1 for each time step the machine is stopped(S) irrespective of whether it is infected or not, and a cost τ_2 for each step an infected machine is operated. One might argue that τ_1 and τ_2 should be

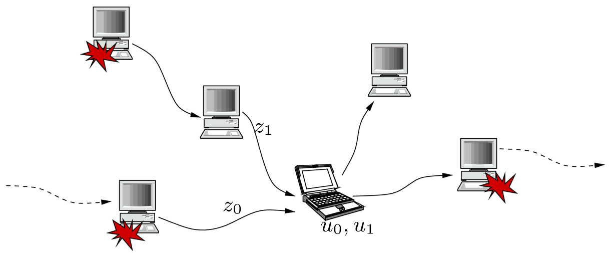


Fig. 1. Alert Sharing Protocol. The laptop is our machine of interest. It uses information, z_0 and z_1, from different chains to choose, actions, u_0 and u_1. It may or may not have seen an anomaly while the machines shown with a blast have seen an anomaly.

the same because an infected machine is as bad as a stopped machine. If that argument is true, the problem becomes trivial and it can be stated right away that the most cost effective strategy is to operate the machine uninterrupted until it is infected. On the contrary, we argue that operating an infected machine costs more as it can infect other machines also. Hence, $\tau_2 > \tau_1$.

Alert Sharing Protocol: Since a computer worm is a distributed phenomenon, inspection outcomes at one machine is a valid forecast of the outcome from a later inspection at another identical machine. (This is an assumption we make to develop the formulation and will be relaxed later on when we discuss a practical application.) Hence, a collection of such machines with identical properties seek to co-operate and share the inspection outcomes. Under this scheme, an inspection outcome at one machine is transmitted to another co-operating peer chosen randomly. The *controller* on the randomly chosen machine uses such received messages to select the optimal control to apply locally. This has the effect of a machine randomly polling several neighbors to know the state of the environment. This gives the uninfected machines an opportunity to take actions that prevent infection. Refer to Fig. 1. In addition to the inspection outcome, peers share information about the anomaly observed in what we call an *anomaly vector* – the structure, form and generation of which we leave undefined. Any two peers observing the same anomaly generate identical *anomaly vector*s.

Goal: Now, the problem is to determine the policy that minimizes the total expected cost of operating the machine for N time periods in an environment that could possibly be infected with a worm. DP problems are generally plagued with state space explosion with increasing number of stages to the horizon. However, since we solve the DP formulation of our problem offline, the value of N does not have any impact on the operational efficiency of the model. Moreover, DP problems can be solved approximately, or analytically for larger Ns significantly reducing the computational needs of the original formulation. The rest of this section develops the formulation for the current problem and provides a solution for $N = 3$. Computer generated results for larger Ns are presented and discussed in later sections.

3.2 Problem Formulation

The above description of the problem fits the general framework of Sect. 2.1, "Problems with imperfect state information." The state, control and observation variables take values as follows:

$$x_k \in \{P, \overline{P}\}, \quad u_k \in \{C, S\}, \quad z_k \in \{G, B\} \ .$$

The machine by itself does not transit from one state to another. Left to itself, it remains put. It is transferred from P to $\overline{P}$ only by a worm infection, a random process – an already infected victim chooses this machine randomly. The evolution of this system follows (1), and is shown in Fig. 2. The function f_k of (1) can be derived from Fig. 2 as follows:

$$\begin{aligned} &P(x_{k+1} = P \mid x_k = P, u_k = C) = \overline{\lambda}\,, \\ &P(x_{k+1} = \overline{P} \mid x_k = P, u_k = C) = \lambda\,, \\ &\vdots \\ &P(x_{k+1} = \overline{P} \mid x_k = \overline{P}, u_k = S) = 1 \ . \end{aligned} \tag{6}$$

The random disturbance, w_k is provided by λ and is rolled in x_k. λ is the infectious force, a function of the number of the machines infected on the Internet. Assuming the machine initially starts in state P, the probability distribution of x_0 is

$$P(x_0 = P) = \overline{\lambda}\,, \qquad P(x_0 = \overline{P}) = \lambda \ . \tag{7}$$

(This assumption is for exposition only. In practice, we do not have to know the initial state the machine starts in.) Recollect that the outcome of each inspection of the machine is an imperfect observation of the state of the system. Thus,

$$\begin{aligned} &P(z_k = G \mid x_k = \overline{P}) = \mathrm{fn}\,, \\ &P(z_k = B \mid x_k = \overline{P}) = (1 - \mathrm{fn})\,, \\ &P(z_k = G \mid x_k = P) = (1 - \mathrm{fp})\,, \\ &P(z_k = B \mid x_k = P) = \mathrm{fp} \ , \end{aligned} \tag{8}$$

where fp and fn are properties of the *sensor*s denoting the false positive and false negative (miss) rates.

Assuming the cost function remains the same regardless of time, the sub-script k can be dropped from g_k. We define the cost function as follows:

$$\begin{aligned} &g(P, C) = 0, \qquad g(\overline{P}, C) = \tau_2, \\ &g(P, S) = g(\overline{P}, S) = \tau_1, \\ &g(x_N) = 0. \end{aligned} \tag{9}$$

$g(x_N) = 0$ because u_N is chosen with accurate knowledge of the environment, (i.e) whether there is a worm or not. If there is a worm, $u_N = S$, else $u_N = C$.

Our problem now is to find functions $\mu_k(I_k)$ that minimize the total expected cost

$$\underset{x_k, z_k}{E} \left\{ g(x_N) + \sum_{k=0}^{N-1} g\big(x_k, \mu_k(I_k)\big) \right\} \ .$$

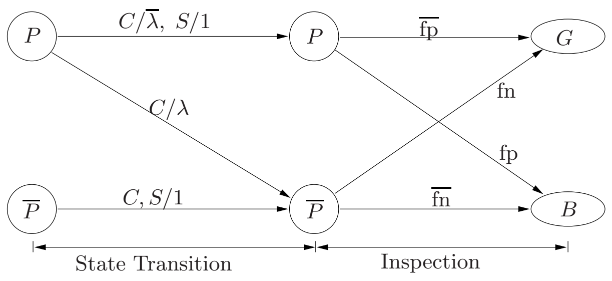


Fig. 2. The left half of the figure shows the state transition probabilities for each action. For example, the system goes from state P to P with a probability of $\overline{\lambda}$ when action C is applied. The right half of the figure shows the observation probabilities for each state. For example, when the system is in state P, the sensors output a G with a probability of $\overline{\text{fp}}$.

We now apply the DP algorithm to the augmented system (refer Sect. 2.1). It involves finding the minimum cost over the two possible actions, C and S, and has the form:

$$\begin{aligned} J_k(I_k) = \min_{\{C,S\}} \Bigg[& \Big(P(x_k = P \mid I_k, C) \cdot g(P, C) + P(x_k = \overline{P} \mid I_k, C) \cdot g(\overline{P}, C) \Big) \\ & \qquad + \mathop{E}_{z_{k+1}} \Big\{ J_{k+1}(I_k, C, z_{k+1}) \mid I_k, C \Big\} \ , \\ & \Big(P(x_k = P \mid I_k, S) \cdot g(P, S) + P(x_k = \overline{P} \mid I_k, S) \cdot g(\overline{P}, S) \Big) \\ & \qquad + \mathop{E}_{z_{k+1}} \Big\{ J_{k+1}(I_k, S, z_{k+1}) \mid I_k, S \Big\} \Bigg] \end{aligned} \quad (10)$$

where $k = 0, 1, \ldots N-1$ and the terminal condition is $J_N(I_N) = 0$. Applying the costs (9), and noticing that $P(x_k = P \mid I_k, S) + P(x_k = \overline{P} \mid I_k, S)$ is the sum of probabilities of all elements in a set of exhaustive events, which is 1, we get

$$\begin{aligned} J_k(I_k) = \min_{\{C,S\}} \Bigg[& \tau_2 \cdot P(x_k = \overline{P} \mid I_k, C) \ + \mathop{E}_{z_{k+1}} \Big\{ J_{k+1}(I_k, C, z_{k+1}) \mid I_k, C \Big\}, \\ & \tau_1 \ + \mathop{E}_{z_{k+1}} \Big\{ J_{k+1}(I_k, S, z_{k+1}) \mid I_k, S \Big\} \Bigg] \ . \end{aligned} \quad (11)$$

This is the required DP formulation of response to worms. Next, we demonstrate a solution derivation to this formulation for $N = 3$.

3.3 Solution

Here we show a solution assuming that we expect to know with certainty about the presence of a worm at the receipt of the third message, that is, $N = 3$. The same procedure can be followed for larger Ns.

With that assumption, control u_2 can be determined without ambiguity. If the third message says there is a worm, we set $u_2 = S$, else we set it to C. This also means that the cost to go at that stage is

$$J_2(I_2) = 0 \ . \qquad \text{(Terminal Condition)}$$

Penultimate Stage: In this stage we determine the cost $J_1(I_1)$. Applying the terminal condition to the DP formulation (11), we get

$$J_1(I_1) = \min\Big[\tau_2 \cdot P(x_1 = \overline{P} \mid I_1, C) \ , \ \tau_1 \Big] \ . \tag{12}$$

The probabilities $P(x_1 = \overline{P} | I_1, C)$ can be computed using Bayes' rule and (6–8), assuming the machine starts in state P. (See Sect. B for exposition.) The cost for each of the eight possible values of $I_1 = (z_0, z_1, u_0)$ under each possible control, $u_1 \in \{C, S\}$ is computed using (11). Then, the control with the smallest cost is chosen as the optimal one to apply for each z_1 observed. The *cost-to-go*, $J_1(I_1)$, thus calculated are used for the zeroth stage.

Stage 0*:* In this stage we determine the cost $J_0(I_0)$. We use (11) and values of $J_1(I_1)$ calculated during the previous stage to compute this cost. As before this cost is computed for each of the two possible values of $I_0 = (z_0) = \{G, B\}$, under each possible control, $u_1 = \{C, S\}$. Then, the control with the smallest cost is chosen as the optimal one to apply for the observed state of the machine. Thus we have,

$$\begin{aligned} J_0(I_0) = \min\Big[&\tau_2 \cdot P(x_0 = \overline{P} \mid I_0, C) \ + \underset{z_1}{E}\Big\{J_1(I_1) \mid I_0, C\Big\} \ , \\ &\tau_1 + \underset{z_1}{E}\Big\{J_1(I_1) \mid I_0, S\Big\}\Big] \ . \end{aligned} \tag{13}$$

The optimal cost for the entire operation is finally given by

$$J^* = P(G)J_0(G) + P(B)J_0(B) \ .$$

We implemented a program that can solve the above formulation for various values of λ, fp, and fn. A sample rule-set generated by that program is given in Table 1. Armed with this solution, we now show a practical application.

4 A Practical Application

4.1 Optimal Policy

Table 1 shows the optimal policies for a given set of operational parameters. The table is read bottom up. At start, assuming the machine is in state P, the optimal action is to continue, C. In the next time step, stage 0, if the observation is B, the optimal action is to stop, S. If $z_0 = B$ is followed by $z_1 = G$, the optimal action is to operate the machine, C. This is denoted by the second line in

Table 1. An optimal policy table

$\lambda = 0.50$, fp $= 0.20$, fn $= 0.10$, $\tau_1 = 1$, $\tau_2 = 2$			
	I_k	J_k	u_k
Stage 1	(G, G, S)	0.031	C
	(B, G, S)	0.720	C
	(G, B, S)	0.720	C
	(B, B, S)	1.000	S
	(G, G, C)	0.270	C
	(B, G, C)	1.000	S
	(G, B, C)	1.000	S
	(B, B, C)	1.000	S
Stage 0	(G)	0.922	C
	(B)	1.936	S
Start		1.480	C

stage 1. This shows that an undesirable response is rolled back when the environment is deemed not dangerous. In a practical application, such a table will be looked up for a given λ and observation to choose the optimal action. Note that the first, third, sixth and eighth states are unreachable because, for the given z_0, the control u_0 mentioned in the vector is never applied if the system operates in good faith.

4.2 Choosing λ

The value of λ varies with the extent of infection in the Internet. Given we are uncertain that there is a worm in the Internet, λ cannot be determined with any accuracy. Rather, only estimates can be made. Hence the distributed Sequential Hypothesis Testing developed earlier is used to estimate λ [9].

Given a sequence of observations $\boldsymbol{y} = \{y_0, y_1, \ldots, y_n\}$, made by a sequence of other participating nodes, and two contradicting hypotheses that there is a worm on the Internet(H_1) and not(H_0), the former is chosen when the likelihood ratio $L(\boldsymbol{y})$ of these hypotheses is greater than a certain threshold η [9]. This threshold η is determined by the performance conditions required of the algorithm. Assuming the observations are independent, $L(\boldsymbol{y})$ and η are defined as follows:

$$L(\boldsymbol{y}) = \prod_{i=1}^{n} \frac{P(y_i|H_1)}{P(y_i|H_0)}, \quad \eta = \frac{DD}{DF} , \tag{14}$$

where DD is the minimum desired detection rate and DF is the maximum tolerable false positive rate of the distributed Sequential Hypothesis Testing(dSHT) algorithm. We define each of the above probabilities as follows:

$$P(y_k = B \mid H_1) = [\lambda\,(1 - \text{fn}) + (1 - \lambda)\,\text{fp}] ,$$
$$P(y_k = G \mid H_1) = [(\lambda\,\text{fn}) + (1 - \lambda)(1 - \text{fp})] ,$$

$$\begin{aligned} P(y_k = B \mid H_0) &= \mathrm{fp}\,, \\ P(y_k = G \mid H_0) &= (1 - \mathrm{fp})\;. \end{aligned} \tag{15}$$

The first equation in the above set is the probability of observing a B given hypothesis H_1 is true. It is the sum of probability of getting infected (λ) times the probability of detection, and the probability of not getting infected$(1 - \lambda)$ times the probability of false positives. The others in (15) are defined similarly.

For a received sequence of observations, a node calculates $L(\boldsymbol{y})$ for several values of λ – say for ten different values in steps of 0.1 starting at 0.1. The lowest λ for which the $L(\boldsymbol{y})$ exceeds η is then taken as the current levels of infection and used in determining the optimal response. The reason for choosing discrete values of λ will be apparent shortly.

An observation at a node can be conveyed to another by transmitting the observation vector, $\boldsymbol{y} = \{y_0\}$. The recepient can add its own observation to this vector making it $\boldsymbol{y} = \{y_0, y_1\}$. Such a sequence accumulates information leading to larger vectors with each hop. Given $L(\boldsymbol{y})$ in (14) is essentially a digest of such vectors, no node has to transmit a whole vector. Instead, it suffices to transmit just one number, $L(\boldsymbol{y})$. A recepient can update the received $L(\boldsymbol{y})$ using (14), (15), and its own observations. It is indeed a conundrum to estimate λ using (15), which is a function of λ itself. This problem is solved as described in the previous paragraph – the lowest λ for which $L(\boldsymbol{y})$ exceeds η is taken as the current operating λ.

In operational practice, a policy in the form of a table is calculated offline for several candidate values of λ. Each row in these tables gives a u_k for a given I_k. For each new λ estimated, the corresponding table is consulted to choose u_k given I_k, where I_k is the node's own past observations and corresponding actions. Thus, each node only receives a likelihood ratio of the worm's presence from its peers and also has to remember only its own I_k. Limiting the number of such tables is the reason for choosing discrete λs in the preceeding paragraphs.

4.3 Larger *N*s

As N increases, the dimensions of I_k increases, which in turn increases the number of the calcuations involved exponentially. This problem can be overcome by reducing I_k to smaller dimensions containing only the *Sufficient Statistics* yet summarizing all essential contents of I_k as far as control is concerned. There are many different functions that can serve as *sufficient statitics*. The one we use here is the conditional probability distribution $P_{x_k|I_k}$ of the state x_k, given the information vector I_k [6]. Discrete-time stochastic systems can be described by the evolution

$$P_{x_{k+1}|I_{k+1}} = \Phi_k(P_{x_k|I_k}, u_k, z_{k+1})\,, \tag{16}$$

where Φ_k is a function that estimates the probalistic state of the system $P_{x_k|I_k}$ based on $P_{x_{k-1}|I_{k-1}}$, z_k and u_{k-1}, and can be determined from the data of the problem [5]. Figure 3 explains this concept. The actuator $\overline{\mu}_k$ then selects the optimal response based on $P_{x_k|I_k}$.

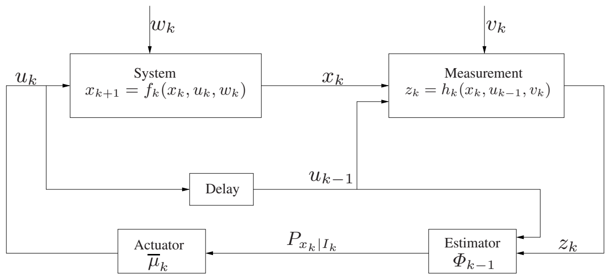


Fig. 3. The controller split into an *Estimator* and an *Actuator*. The *Estimator* Φ_{k-1} estimates the probabilistic state of the system $P_{x_k|I_k}$ while the *Actuator* $\overline{\mu}_k$ picks the appropriate control u_k.

This re-formulation makes it easy to apply the response model for larger Ns. We implement this model and evaluate it in a simulation. The evaluation and the results are discussed in the next section.

5 Evaluation

The sufficient statistics formulation discussed in the previous section was implemented and evaluated with a discrete event simulation. The simulation consisted of a world of 1000 participants with 10% of the machines being vulnerable. We set the number of stages to operate the machine, $N = 4$ to calculate the rule-sets. Note that $N = 4$ is used only to calculate the rule-sets but the machines can be operated for any number of steps. N is essentially the number of past observations and actions that each machine remembers. The local IDSes were set to have a false positive and false negative rates of 0.1. These characteristics of the local IDS is used to calculate the probability of infection, λ with a desired worm detection rate of 0.9 and failure rate of 0.1. In all the following experiments, we used a random scanning worm that scans for vulnerable machines once every unit-time.

5.1 Experiments

Parameters of Evaluation: A set of experiments was designed to understand the effect of various parameters on the effectiveness of the model in controlling the spread of the worm. The only free variable we have here is the ratio τ_2/τ_1. There is no one particular parameter that can measure or describe the effectiveness of the response model. Rather, the effectiveness is described by the number of vulnerable machines that are not infected and of those the number that provide service, i. e. in state C.

Algorithm: The algorithm for the discrete-event simulation is as follows. At each time cycle.

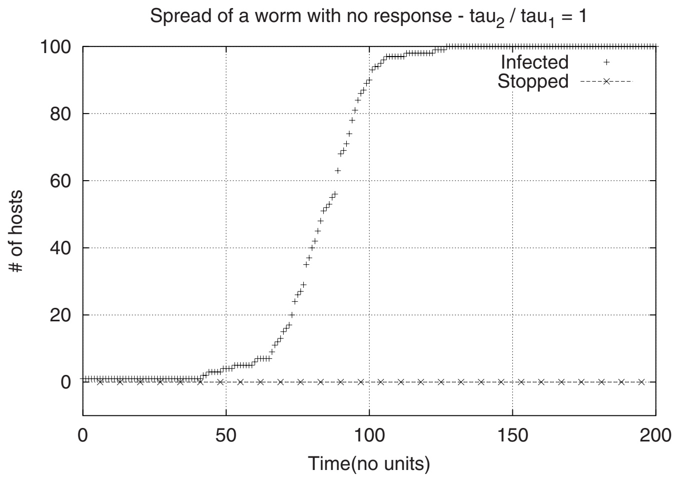


Fig. 4. No machines are stopped when the cost of being infected is the same as cost of shutting down the machine. fp = fn = 0.1, $DD = 0.9, DF = 0.1$

- all infected machines attempt one infection,
- all machines that had an alert to share, share the likelihood ratio that there is a worm on the Internet with another randomly chosen node,
- and all vulnerable machines that received an alert earlier take a response action based on the information received and the current local observations.

Results: In the first experiment, we want to make sure that we have a worm that behaves as normal random scanning worm and validate the response model for

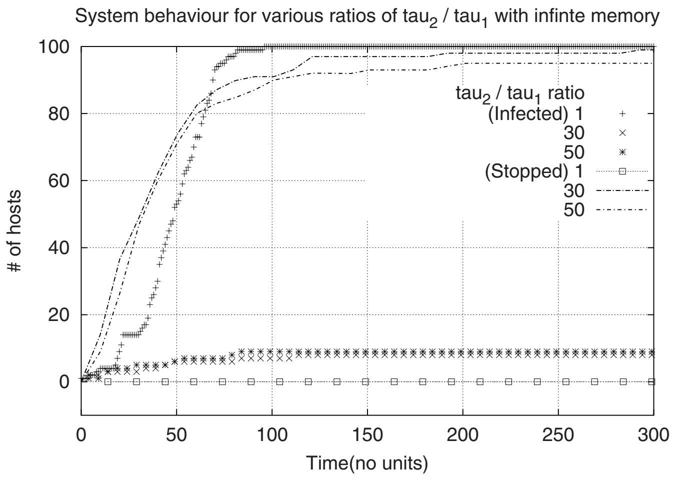


Fig. 5. When nodes are set to remember infection attempts forever, they never back-off their defensive posture. Once entered the S state, a machine stays there. fp = fn = $0.1, DD = 0.9, DF = 0.1$.

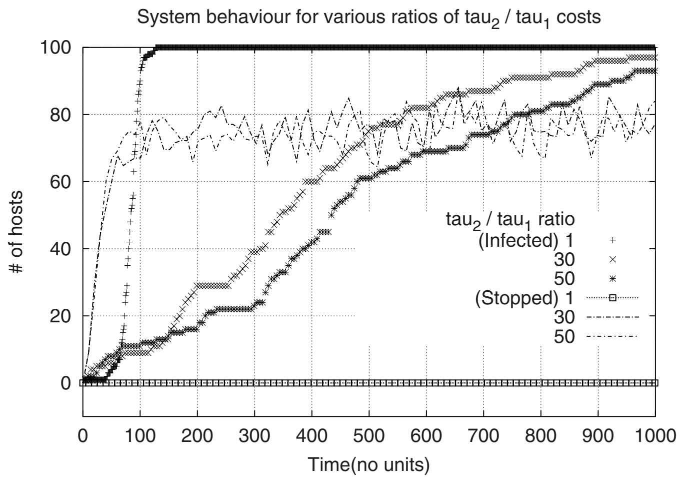


Fig. 6. Higher costs of being infected invoke stricter responses. fp = fn = 0.1, $DD = 0.9, DF = 0.1$.

the degenerate cases. We verify this by providing no response. This response can be achieved by setting the cost ratio to 1 – the cost of stopping the service is the same as getting infected. In this scenario, we expect the response model not to take any defensive measures against suspected infection attempts. As expected, we see in Fig. 4, that none of the machines are stopped (S state). The worm spreads as it would spread when there is no response in place. This validates our worm and also our response model.

As another sanity check we set the machines to remember infection attempts forever. Under this policy, once a machine enters the S state, it remains in that state forever. We see that in this case (Fig. 5) the number of machines infected are very low except when $\tau_2/\tau_1 = 1$.

In the next experiment, we try to understand the behavior of our response model in various situations. Since the only free variable is the ratio τ_2/τ_1, we repeat the previous experiment with various values for that ratio. The results for this set of experiments is shown in Fig. 6. This graph shows behavior of our response model in three different tests. There are two different curves for each test indicating the number of vulnerable machines being infected and the number of machines that are stopped. We can see that when the ratio is 1, the number of machines that are in S state is 0. As the ratio τ_2/τ_1 rises, the response becomes stricter. We see that the number of machines in the stopped(S) state is higher when the cost of being infected is higher. Also the worms spreads significantly slower than without any response in place or with a lower τ_2/τ_1 ratio.

5.2 Effects of Increasing *N*

The experiments shown earlier in this section were all conducted with $N = 4$. An interesting question to ask here, "What happens if we increase the value of

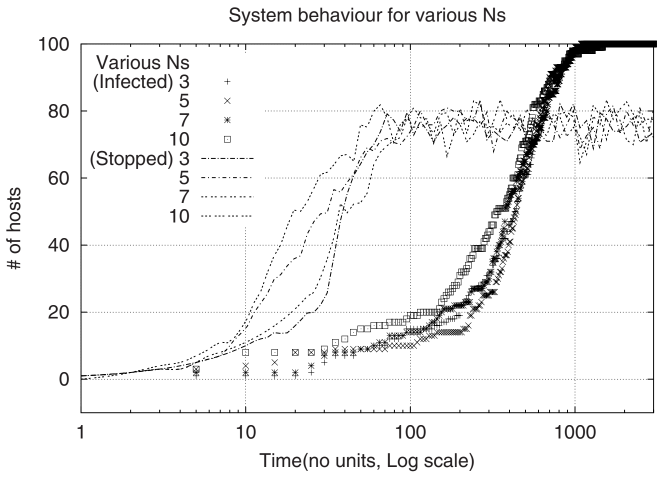


Fig. 7. Larger Ns do not contribute much to improve performance due to the small number of dimensions to the state, $x_k \in \{P, \overline{P}\}$. fp = fn = $0.1, DD = 0.9, DF = 0.1$

N?". Fig. 7 shows the performance of the system for various values of N while holding the ratio of τ_2/τ_1 constant at 30. The set of sigmoidal curves that increase monotonically trace the growth of the worm, while the other set of curves trace the number of nodes that are shut-down at any given instant. We notice that there is no appreciable slowing of the worm with increased values of N – all the worm growth curves are bunched up together. This is due to the small number of dimensions to the state, $x_k \in \{P, \overline{P}\}$. A larger observation space does not contribute much to improve the performance of the system.

6 Conclusion

This section concludes this paper by reflecting on the strengths and weaknesses of the approach discussed so far. Assumptions are identified. Arguments are made for the choice of certain design and evaluation decisions. Where appropriate, future directions are provided to address the limitations identified.

6.1 Limitations and Redress

There are several topics in this paper yet to be addressed. There are issues to be addressed from three different perspectives – one, problems that would arise during the practical adoption of this model; two, in the evaluation; and three, in the model itself.

Adoption Impediments: This is a collaborative response model. As with any collaborative effort, there is a host of issues such as privacy, confidentiality, non-repudiation, etc, that will need to be addressed during practical adoption. Thankfully, these are issues for which there are solutions available already through cryptography and IPSEC. In a co-operative or collaborative environment, we expect these issues to be either easily resolved or already addressed. Regardless, co-operation amongst various entities on the Internet such as amongst different corporate networks pose more legal, political, and economic problems than technical. In such cases where sharing anomaly information with networks outside of the corporation is not feasible, applying this response model within the corporate network itself can provide valuable protection.

Assigning realistic values to τ_1 and τ_2 is another major impediment to adoption. However, that is a separate problem requiring independent study. There are indeed prior work that attempt to assign costs to various responses that can be used [14,4]. Whereas, this paper focusses on optimizing those costs for overall operation of a system.

Evaluation Issues: Integral and faithful scaling down of the Internet is a difficult problem [22], which makes evaluating worm defenses more so [8]. At one extreme we have realistic evaluation possible only on the Internet, which is infeasible. At the other extreme, we have pure mathematical models. In between these two extremes, we have simulations such as the one used in this paper and emulation as used in one of our previous studies for worm detection [9].

With the availability of data about Internet traffic during worm outbreaks, it may be possible to evaluate the defense model on a network testbed such as Emulab [23] or DETER [3] by replaying the traffic for a scaled down version of the Internet. Such an experiment would need the available data to be carefully replayed with tools such as TCP Replay,TCP Opera [12], etc. This is a future task. Nevertheless, such emulation experiments can only scale up to a certain level and after that we would have to resort to mathematics or simulations to extrapolate the results to Internet scales.

We avoid emulation experiments by choice. Emulations can provide details about exploit behavior, traffic patterns, etc. As important as those issues are, they lie outside the scope of our present interest and are considered for later study. Focus for this paper is primarily on the cost optimization models. As mentioned in the previous paragraph, experiment population sizes are limited in emulations while simulations can support larger number of nodes. Given that stochastic processes are involved in our model, we need a large population to achieve fidelity in results. Consequently, simulations form a natural choice for our experiments.

An issue to be studied is the behavior of this model in the face of false alarms and isolated intrusions. For example, consider one and only participant raising an alarm for an isolated event and several other participants choosing the S control. We would like to know when these participants would apply the C control. Trivially, we can set a time-out for the defense to be turned-off. However, the time-out should be chosen carefully and probably be dynamic to guard against exposing oneself to slow-worm attacks.

In our experiments we have showed only one worm operating at a time. While this might seem like a limitation of the model, it is not. As mentioned in Sect. 3.1, there is an *anomaly vector* associated with each suspected worm incident. When multiple worms operate simultaneously, each will be associated with a different *anomaly vector*. In operational practice, we expect a different process to be associated with each *anomaly vector* so multiple worms can be handled independently and concurrently.

Limitations and Extensions to the Model: When there is a cost to sampling packets, this model can be extended to optimally stop the sampling process and declare either that there is a worm or that there is no worm – essentially a distributed worm detector. Interestingly, this extension would lead us to the distributed Sequential Hypothesis Testing that we discussed in our previous paper [9].

One of the assumptions in our model is that the worm is a random scanning worm. This model will not work against more intelligent worms such as hit-list or flash worms but will likely be moderately successful against sub-net scanning worms [19]. Evaluating and extending the model against such other kinds of worms is a future work.

Integrity of the sensors, and absence of wilful malfeasance are assumed in our model. After all, in the real world we do assume the safety and security of the firewalls and IDSes we use. Nevertheless, if a few of the sensors are compromised by the attackers, we expect the stochastic nature of our model to act as a cushion absorbing some of the ill-effects. This needs to be evaluated. If numerous sensors are affected, our assumption about collaboration is not valid any more and the results are undefined.

Actions such as C and S if applied frequently could lead to a very unstable system. We need to evaluate this factor in light of ambient anomaly levels in different environments. This is a problem with the model itself. However, this can be alleviated to some extent during adoption in various ways. For example, the set of response options, $\{C, S\}$, can be made larger by introducing several levels of reduced functionality. This will however increase the complexity of the DP formulation but can be tolerated as we solve the formulation offline.

When all participants behave identically each participant knows exactly how the others will behave. In such a scenario, each one can make a better decision about the optimal control to be applied taking into account the others' behavior. For example, if participant A determines that the optimal policy to be applied is S, it now knows that all other participants will also apply the same control. Then, there is no need for A to apply S. Instead A could apply C as there is no opportunity for a worm to spread when all others participants are stopped. The problem now enters the realm of *game theory*.

6.2 Strengths

One question that needs to be answered for any defensive technique is this: "If the attacker knows about the approach being used for defense, will s/he be able to write a new generation of worms that can overcome the defense?"

There are two different goals that an attacker with knowledge about our system can try to achieve. One, try to circumvent the defense and spread the worm. Two, trick the defense into over-reacting.

The second goal cannot be achieved because of the dynamic and self-regulating nature of our approach, which is based on the current environmental conditions as depicted in Fig. 3. The attacker may force our system to react to an initial stimulus that is not a true worm, but once the stimulus has reduced, the defence pulls back too. If the sensors are compromised, however, the results are undefined as mentioned in the previous section. However, compromising sensors are extraneous to the model and is not a tenable argument against the model.

To achieve the first goal, the worm needs to either spread very slowly such that information about anomalous incidents are forgotten by the participants, or attack pre-selected victims that may not be alerted by its peers. However, since the alerts are shared with randomly chosen peers while the worm is spreading, there can be no effective pre-selection that can overcome the defense. Whereas a slow spreading worm might be successful to a certain extent.

Nevertheless, we believe that a slow spreading worm can be identified by other means such as manual trouble-shooting prompted by the ill-effects of the worm; unless the worm installs a time-bomb that is set to trigger after the worm has spread to most vulnerable nodes. We also believe that such slow worms will be circumvented by routine maintenance patches – most worms we know so far have exploited only known, but unpatched, vunlerabilities.

Moreover, there is a heightened awarness about security issues amongst the information technology community than ever before. Laws related to data security are being tightened and enforced more vigorosly than in the past. Patch generation and deployment techniques have advanced tremendously recently. In such an environment, we expect that steps to patch or workaround known vulnerabilities will be taken with more urgency than ever before effectively thwarting extremely slow worms discussed in the preceeding paragraphs.

Thus, the worm has a very narrow window between spreading too slow and spreading too fast – the window where our response mechanism works to thwart the worm. In conclusion, to answer the question above, knowledge of our approach does not provide much value to the attacker or new generation of worms.

6.3 Summary

This paper presents a novel control-theoretic approach toward worm response. We showed how to formalize a response strategy as a Dynamic Programming problem and solve this formulation to yield a practically applicable response solution. This formalization has been one of the key contributions of this paper. We show how this model severely curtails the worm options available to attackers. Several interesting directions in which this work could be extended are identified.

Acknowledgments. We would like to thank Branislav Kveton of Intel Research and Jed Crandall of University of New Mexico for providing early critique on the work.

References

1. Anagnostakis, K.G., et al.: A cooperative immunization system for an untrusting internet. In: Proc. of IEEE ICON, October 2003, pp. 403–408 (2003)
2. Anagnostakis, K.G., Greenwald, M.B., Ioannidis, S., Keromytis, A.D.: Robust reactions to potential day-zero worms through cooperation and validation. In: Katsikas, S.K., López, J., Backes, M., Gritzalis, S., Preneel, B. (eds.) ISC 2006. LNCS, vol. 4176, pp. 427–442. Springer, Heidelberg (2006)
3. Bajcsy, R., et al.: Cyber defense technology networking and evaluation. Commun. of the ACM 47(3), 58–61 (2004)
4. Balepin, I., Maltsev, S., Rowe, J., Levitt, K.: Using specification-based intrusion detection for automated response. In: Vigna, G., Krügel, C., Jonsson, E. (eds.) RAID 2003. LNCS, vol. 2820, pp. 136–154. Springer, Heidelberg (2003)
5. Bertsekas, D.P., Shreve, S.E.: Stochastic Optimal Control: The Discrete Time Case. Academic Press, N.Y (1978)
6. Bertsekas, D.P.: Dynamic Programming and Optimal Control, 3rd edn., vol. 1. Athena Scientific (2005)
7. Cai, M., Hwang, K., Kwok, Y.-K., Song, S., Chen, Y.: Collaborative internet worm containment. IEEE Security and Privacy 4(3), 34–43 (2005)
8. Cheetancheri, S.G., et al.: Towards a framework for worm defense evaluation. In: Proc. of the IPCCC Malware Workshop on Swarm Intelligence, Phoenix (April 2006)
9. Cheetancheri, S.G., Agosta, J.M., Dash, D.H., Levitt, K.N., Rowe, J., Schooler, E.M.: A distributed host-based worm detection system. In: Proc. of SIGCOMM LSAD, pp. 107–113. ACM Press, New York (2006)
10. Costa, M., et al.: Vigilante: end-to-end containment of internet worms. In: Proc. of the SOSP, pp. 133–147. ACM Press, New York (2005)
11. Dash, D., Kveton, B., Agosta, J.M., Schooler, E., Chandrashekar, J., Bachrach, A., Newman, A.: When gossip is good: Distributed probabilistic inference for detection of slow network intrusions. In: Proc. of AAAI, AAAI Press, Menlo Park (2006)
12. Hong, S.-S., Felix Wu, S.: On Interactive Internet Traffic Replay. In: Valdes, A., Zamboni, D. (eds.) RAID 2005. LNCS, vol. 3858, pp. 247–264. Springer, Heidelberg (2006)
13. Kim, H.-A., Karp, B.: Autograph: Toward automated, distributed worm signature detection. In: Proc. of the USENIX Security Symposium (2004)
14. Lee, W., Fan, W., Miller, M., Stolfo, S.J., Zadok, E.: Towards cost-sensitive modeling for intrusion detection and response. J. of Computer Security 10(1,2) (2002)
15. Malan, D.J., Smith, M.D.: Host-based detection of worms through peer-to-peer cooperation. In: Proc. of the WORM, pp. 72–80. ACM Press, New York (2005)
16. Newsome, J., Karp, B., Song, D.: Polygraph: Automatically generating signatures for polymorphic worms. In: Proc. of the IEEE Symposium on Security and Privacy, pp. 226–241. IEEE, Los Alamitos (2005)
17. Singh, S., Estan, C., Varghese, G., Savage, S.: Automated worm fingerprinting. In: Proc. of OSDI, San Francisco, CA (December 2004)
18. Sidiroglou, S., Keromytis, A.D.: Countering network worms through automatic patch generation. IEEE Security and Privacy 3(6), 41–49 (2005)
19. Staniford, S., Paxson, V., Weaver, N.: How to 0wn the Internet in Your Spare Time. In: Proc. of the Summer USENIX Conf., Berkeley, August 2002. USENIX (2002)

20. Wang, K., Cretu, G., Stolfo, S.J.: Anomalous payload-based worm detection and signature generation. In: Proc. of RAID. ACM Press, New York (2005)
21. Wang, K., Stolfo, S.J.: Anomalous payload-based network intrusion detection. In: Proc. of RAID, September 2004. ACM Press, New York (2004)
22. Weaver, N., Hamadeh, I., Kesidis, G., Paxson, V.: Preliminary results using scale-down to explore worm dynamics. In: Proc. of WORM, pp. 65–72. ACM Press, New York (2004)
23. White, B., et al.: An integrated experimental environment for distributed systems and networks. In: OSDI, Boston, December 2002, pp. 255–270. USENIX (2002)
24. Zou, C.C., Gao, L., Gong, W., Towsley, D.: Monitoring and early warning for internet worms. In: Proc. of the CCS, pp. 190–199. ACM Press, New York (2003)

A DP Example

We provide a short, classical inventory control example to help readers unfamiliar with DP to formulate a DP problem. This is an example from Bertsekas [6].

Consider the problem of stocking store shelves for N days. The state of the system is denoted by the quantity (x_k) of a certain item available on the store shelves at the beginning of a day. Shelves are stocked(with u_k units) at day break while demand(w_k) for the item is stochastic during the day. Both w_k and u_k are non-negative. There is no change overnight. It is clear that this system evolves according to:

$$x_{k+1} = \max(0, x_k + u_k - w_k)\,.$$

While there is an upper bound of, say, 2 units on the stock that can be on the shelves, demand in excess of stocks is lost business. Say, the storage costs for a day is $(x_k + u_k - w_k)^2$ implying penalty for both lost business and for excess inventory at the end of the day. Assuming the purchase cost incurred by the store is 1 per unit stock, the operating cost per day is

$$g_k(x_k, u_k, w_k) = u_k + (x_k + u_k - w_k)^2\,.$$

The terminal cost at the end of N days is assumed to be 0. Say the planning horizon N is 3 days and the initial stock $x_0 = 0$. Say, the demand w_k has the same probability distribution for all three days and is given by

$$p(w_k = 0) = 0.1 \qquad p(w_k = 1) = 0.7 \qquad p(w_k = 2) = 0.2\,.$$

The problem now is to determine the *optimal policy* for reordering of stocks so as to minimize the total operational cost. Applying (3), the DP algorithm for this problem is

$$J_k(x_k) = \min_{\substack{0 \le u_k \le 2 - x_k \\ u_k = 0,1,2}} \mathop{E}_{w_k} \left\{ u_k + (x_k + u_k - w_k)^2 + J_{k+1}(x_{k+1}) \right\}\,, \tag{17}$$

where $k = 0, 1, 2$, and x_k, u_k, w_k can take the values of $0, 1, 2$ while the terminal condition $J_3(x_3) = 0\,.$

Now starting with $J_3(x_3) = 0$ and solving (17) backwards for $J_2(x_k)$, $J_1(x_k)$ and $J_0(x_k)$ for $k = 0, 1, 2$, we find that the *optimal policy* is to reorder one unit if the shelves are empty and nothing otherwise.

B Applying Bayes' Rule

The probabilities, $P(x_1 = \overline{P} \mid I_1, C)$ for (12) can be calculated using Bayes' rule and (6–8). We show the calculations for one of them here for exposition.

$$
\begin{aligned}
&P(x_1 = \overline{P} \mid G,\, G,\, S) \\
&\quad = \frac{P(x_1 = \overline{P},\, G,\, G,\, \mid S)}{P(G,\, G,\, \mid S)} \\
&\quad = \frac{\sum\limits_{i=\{P,\overline{P}\}} P(G|x_0 = i) \cdot P(x_0 = i) \cdot P(G|x_1 = \overline{P}) \cdot P(x_1 = \overline{P}|x_0 = i, u_0 = S)}{\sum\limits_{i=\{P,\overline{P}\}} \sum\limits_{j=\{P,\overline{P}\}} P(G|x_0 = i) \cdot P(x_0 = i) \cdot P(G|x_1 = j) \cdot P(x_1 = j|x_0 = i, u_0 = S)} \\
&\quad = \frac{(\overline{\mathrm{fp}} \cdot \overline{\lambda} \cdot \mathrm{fn} \cdot 0) + (\mathrm{fn} \cdot \lambda \cdot \mathrm{fn} \cdot 1)}{(\overline{\mathrm{fp}} \cdot \overline{\lambda} \cdot \overline{\mathrm{fp}} \cdot 1) + (\overline{\mathrm{fp}} \cdot \overline{\lambda} \cdot \mathrm{fn} \cdot 0) + (\mathrm{fn} \cdot \lambda \cdot \overline{\mathrm{fp}} \cdot 0) + (\mathrm{fn} \cdot \lambda \cdot \mathrm{fn} \cdot 1)}
\end{aligned}
$$

On the Limits of Payload-Oblivious Network Attack Detection

M. Patrick Collins[1] and Michael K. Reiter[2]

[1] RedJack
michael.collins@redjack.com[⋆]
[2] Department of Computer Science,
University of North Carolina at Chapel Hill
reiter@cs.unc.edu

Abstract. We introduce a methodology for evaluating network intrusion detection systems using an *observable attack space*, which is a parameterized representation of a type of attack that can be observed in a particular type of log data. Using the observable attack space for log data that does not include payload (*e.g.*, NetFlow data), we evaluate the effectiveness of five proposed detectors for bot harvesting and scanning attacks, in terms of their ability (even when used in conjunction) to deter the attacker from reaching his goals. We demonstrate the ranges of attack parameter values that would avoid detection, or rather that would require an inordinately high number of false alarms in order to detect them consistently.

Keywords: network intrusion detection, ROC curve, evaluation.

1 Introduction

We address the problem of evaluating network intrusion detection systems, specifically against scan and harvesting attacks. In the context of this work, a harvesting attack is a mass exploitation where an attacker initiates communications with multiple hosts in order to control and reconfigure them. This type of automated exploitation is commonly associated with worms, however, modern bot software often includes automated buffer-overflow and password exploitation attacks against local networks[1]. In contrast, in a scanning attack, the attacker's communication with multiple hosts is an attempt to determine what services they are running; *i.e.*, the intent is reconnaissance.

While harvesting attacks and scanning may represent different forms of attacker *intent* (*i.e.*, reconnaissance vs. host takeover), they can appear to be similar phenomena in traffic logs. More specifically, a single host, whether scanning

[⋆] This work was done while the author was affiliated with the CERT/NetSA group at the Software Engineering Institute, Carnegie Mellon University.

[1] A representative example of this class of bot is the Gaobot family, which uses a variety of propagation methods including network shares, buffer overflows and password lists. A full description is available at http://www.trendmicro.com/vinfo/virusencyclo/default5.asp?VName=WORM_AGOBOT.GEN.

or harvesting, will open communications to an unexpectedly large number of addresses within a limited timeframe. This behavior led to Northcutt's observation that in the absence of payload—either due to the form of log data, encryption or simply a high connection failure rate—methods for detecting these attacks tend to be threshold-based [19]. That is, they raise alarms after identifying some phenomenon that exceeds a threshold for normal behavior.

Historically, such IDS have been evaluated purely as alarms. Lippmann *et al.* [16] established the standard for IDS evaluation in their 1998 work on comparing IDS data. To compare intrusion detectors, they used ROC curves to compare false positive and false negative rates among detectors. Since then, the state of the practice for IDS evaluation and comparison has been to compare IDS' ROC curves [9].

The use of ROC curves for IDS evaluation has been criticized on several grounds. For our purposes, the most relevant is the *base rate fallacy* described by Axelsson [2]. Axelsson observes that a low relative false positive rate can result in a high number of actual false positives when a test is frequently exercised. For NIDS, where the test frequency may be thousands or tens of thousands of per day, a false positive rate as low as 1% may still result in hundreds of alarms.

In this paper, we introduce an alternative method of evaluating IDS that focuses on an IDS' capacity to frustrate an attacker's goals. In order to do so, we develop a model for evaluating IDS that captures the attacker's payoff over an *observable attack space*. The observable attack space represents a set of attacks an attacker can conduct as observed by a particular logging system. The role of logging in the observable attack space is critical; for example, NetFlow, the logging system used in this paper, does not record payload. As such, for this paper, we define an observable attack space that classifies attacks by the attacker's *aggressiveness* (the number of addresses to which they communicate in a sample period) and their *success* (the probability that a communication opened to an address actually contacts something).

To evaluate the payoff, we construct a *detection surface*, which is the probability of detection over the observable attack space, and then apply a *payoff function* to this detection surface. The payoff function is a function representing the rate at which an attacker achieves the strategic goal of that attack, which is either occupying hosts (in a harvesting attack) or scouting network composition (in a scanning attack).

We use the payoff function to evaluate the impact of various IDS on attacker strategy. We can model payoff as a function of the number of viable hosts in a network that an attacking bot communicates with — the more hosts a bot can contact without being detected, the higher his payoff. We show in this paper that several methods which are good at raising alarms primarily identify low-payoff attacks; with these detectors, an attacker can achieve a high payoff simply by limiting his behavior.

By combining detection surfaces with a payoff function, we are able to compare IDS with greater insight about their relative strengths and weaknesses. In particular, we are able to focus on the relationship between detection capacity and attacker payoff. Instead of asking what kind of false positive rate we get for a specific true positive rate, we are able to relate false positive rates to the attacker goals. By doing so, we are able to determine how high a false positive rate we must tolerate in order to prevent an attacker from, say, substantially compromising a network via a harvesting attack. Our work therefore extends the ROC framework into a model of the attacker's own goals. By doing so, we can reframe questions of IDS designs by evaluating their impact on attacker behavior, on the grounds that a rational attacker will attempt to maximize payoff.

Using this approach, we compare the efficacy of five different detection techniques: client degree (*i.e.*, number of addresses contacted); protocol graph size and protocol graph largest component size [6]; server address entropy [15]; and Threshold Random Walk [11]. We train these systems using traffic traces from a large (larger than /8) network. Using this data, we demonstrate the configurations of aggressiveness and success rate with which an attack will go undetected by any of these techniques. Furthermore, we show that when configured to be sufficiently sensitive to counter attackers' goals, these anomaly detection systems will result in more than ten false alarms per hour, even when alarms are limited to occur only once per 30-second interval.

To summarize, the contributions of this paper are the following. First, we introduce a new methodology for evaluating NIDS that do not utilize payload. Second, we apply this methodology to evaluate several attack detection methods previously proposed in the literature, using data from a very large network. And third, we demonstrate via this evaluation the limits that these techniques face in their ability to prevent attackers from reaching harvesting or scanning goals.

The remainder of this paper is structured as follows. §2 is a review of relevant work in IDS evaluation and anomaly detection. §3 describes the IDS that we evaluate in this paper, and how we configure them for analysis. §4 describes the observable attack space and detection surface. §5 describes the first of our two attack scenarios, in this case the acquisition of hosts by an attacker with a hit list. §6 describes the second scenario: reconnaissance by attackers scanning networks. §7 concludes this work.

2 Previous Work

Researchers have conducted comparative IDS evaluations in both the host-based and network-based domains. In the host-based domain, Tan and Maxion [25,17] developed an evaluation methodology for comparing the effectiveness of multiple host-based IDS. Of particular importance in their methodology is the role of the data that an IDS can actually analyze, an idea further extended in Killourhy *et al.*'s work on a defense-centric taxonomy [13]. The methods of Tan and Maxion and of Killourhy *et al.* informed our experimental methodology and the concept of an observable attack space. However, their approach is focused on host-based IDS and they consequently work with a richer dataset then we believe feasible for NIDS.

A general approach to evaluating IDS was proposed by Cárdenas *et al.* [4], who developed a general cost-based model for evaluating IDS based on the work of Gaffney and Ulvila [8] and Stolfo *et al.* [24]. However, these approaches all model cost from a defender-centric viewpoint — the defensive mechanism is assumed to have no impact on the attacker. In contrast, our models treat the attacker as economically rational, meaning that the attacker attempts to maximize payoff within the rules given by the model.

The general problem of NIDS evaluation was first systematically studied by Lippmann *et al.* [16]. Lippmann's comparison first used ROC curves to measure the comparative effectiveness of IDS. The ROC-based approach has been critiqued on multiple grounds [18,9,2]. Our evaluation model is derived from these critiques, specifically Axelsson's [2] observations on the base-rate fallacy. Our work uses a ROC-based approach (specifically, comparing Type I and Type II errors) as a starting point to convert the relative error rates into payoffs.

3 IDS Construction and Training

In the context of this work, an *IDS* is an anomaly detection system that compares the current state of a network against a model of that network's state developed from historical data. In this section, we describe our candidate IDS, and our method for training and configuring them. This section is divided as follows: §3.1 describes the raw data, §3.2 describes the types of IDS used, and §3.3 describes the detection thresholds used for our IDS.

3.1 Raw Data

Every IDS in this paper is trained using a common data source over a common period of time. The source data used in this paper consists of unsampled NetFlow records[2] generated by internal routers in a large (in excess of 16 million distinct IP address) network. For training and evaluation, we use SSH traffic.

NetFlow records approximate TCP sessions by grouping packets into *flows*, sequences of identically addressed packets that occur within a timeout of each other [5]. NetFlow records contain size and timing information, but no payload. For the purposes of this paper, we treat NetFlow records as tuples of the form (clntip, srvip, succ, stime).

The elements of this tuple are derived from the fields available in CISCO NetFlow. The clntip, srvip, succ and stime fields refer, respectively, to the client address, server address, whether a session was successful, and the start time for the session. Since SSH is TCP based, we rely on the port numbers recorded in the original flow record both for protocol identification and classifying the role a particular address played in the flow. Any flow which is sent to or from TCP port 22 is labeled an SSH flow, srvip is the address corresponding to that port

[2] CISCO Systems, "CISCO IOS NetFlow Datasheet", `http://www.cisco.com/en/US/products/ps6601/products_data_sheet0900aecd80173f71.html`, last fetched October 8th, 2007.

and clntip the other address[3]. stime, the start time, is derived directly from the corresponding value in the flow record, and is the time at which the recording router observed the flow's earliest packet.

The succ element is a binary-valued descriptor of whether the recorded flow describes a legitimate TCP session. succ is 0 when the flow describes a TCP communication that was not an actual session (*e.g.*, the target communicated with a nonexistent host), 1 when the flow describes a real exchange between a client and a server.

succ is an inferred property in the sense that it can be truly determined only by the receiving host — a sufficiently perverse attacker could generate one side of a session without the others' involvement. *In situ*, we can approximate succ using other flow properties, such as the number of packets in the flow or TCP flag combinations. In our work on IDS training [7], we approximate succ by setting it to 1 when a flow has 4 or more packets, on the grounds that a TCP session has at least 3 packets of overhead. Other methods for calculating succ include looking for indicators such as total payload, the presence of ACK flags, or aggregate measures such as Binkley and Singh's TCP work weight [3].

In our simulations we generate the succ values as part of the process of generating attack flows. During the simulations, attackers choose their targets from a hit list generated from the training data; the attack's success rate determines how many addresses come from this hit list, and how many addresses are chosen from a pool of dark addresses. For flows communicating with the hit list, succ $= 1$, and for flows communicating with the pool of dark addresses, succ $= 0$.

IDS properties are generated using 30 second (s) samples of traffic data. We refer to a distinct sample as a *log file*, Λ, consisting of all the flows $\lambda_1 \ldots \lambda_l$ whose stime values occur in the same 30s period. The use of 30s periods comes from our previous work on protocol graphs [6].

3.2 IDS State Variables

In the context of this paper, an IDS is a threshold-based alarm that triggers if a value derived from a log file Λ exceeds a threshold derived from a set of training data. Each IDS in this paper is based around a single *state variable* which, when evaluated against a log file produces a scalar *state value.* For this paper, we evaluate the state of a log file using five distinct state variables: g, c, h, d and r. Each state variable is used by one IDS; we will refer to each IDS by its state variable (*e.g.*, "g is an IDS").

$g(\Lambda)$ and $c(\Lambda)$ are, respectively, the total graph size and the largest component size of a *protocol graph* constructed from Λ. A protocol graph, described in our previous work on hit-list detection, is an undirected graph constructed from a log of traffic for a single protocol over a limited observation period [6]. In a protocol graph, the nodes represent hosts communicating using that protocol, and the links represent that a communication between these two hosts happened during

[3] We constrain the flows used in this paper to flows which used an ephemeral port between 1024 and 5000.

that time. In a protocol graph, the graph size is equivalent to the total number of hosts communicating using a particular protocol. The largest component size is the size of the largest connected component of the graph.

$h(\Lambda)$ is the entropy of server addresses in Λ. This metric is derived from work by Lakhina *et al.* [15] on mining traffic features. The entropy is defined as:

$$h(\Lambda) = -\sum_{i \in \mathsf{srvs}(\Lambda)} \left(\frac{|\{\lambda \in \Lambda | \lambda.\mathsf{srvip} = i\}|}{|\Lambda|} \right) \log_2 \left(\frac{|\{\lambda \in \Lambda | \lambda.\mathsf{srvip} = i\}|}{|\Lambda|} \right) \quad (1)$$

where $\mathsf{srvs}(\Lambda) = \bigcup_{\lambda \in \Lambda} \lambda.\mathsf{srvip}$ is the set of all server addresses observed in the log file. During a harvesting attack, an attacker will increase $|\mathsf{srvs}(\Lambda)|$, which reduces the probability of any one server being the target of a communication and therefore increases the entropy.

$d(\Lambda)$, the maximum degree of Λ, is the number of servers with which the busiest client in Λ communicated. $d(\Lambda)$ is arguably the simplest form of scan detection and consequently has been used by a variety of anomaly detection systems, notably GrIDS [23] and Bro [20].

$r(\Lambda)$ is the maximum *failed connection run* observed in Λ. A failed connection run is a sequence of flow records $\lambda_1 \ldots \lambda_n$ where each λ in the run has the same client address and $\lambda_i.\mathsf{succ} = 0$. This method is used by TRW scan detection [11] to determine if an address is actively scanning. We use the maximum failed connection run measure to indicate whether TRW would have detected at least one attack during the sample period.

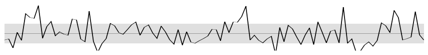
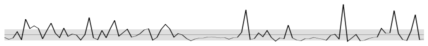
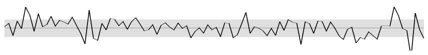
3.3 IDS Thresholds

In order to calculate detection thresholds for four of the IDS we consider (g, c, h and d), we first must train the IDS using log files of benign traffic from the monitored network. However, SSH traffic is prone to constant scanning [1] which, unless aggressively filtered, will result in artificially high thresholds.

To address the problem of constant clumsy scanning, we use a two-stage filtering method developed in previous work [7]. This approach is based on our previous observations that certain graph attributes of major protocols (graph size and largest component size) can be modeled using a Gaussian distribution when the traffic logs describing those attributes do not contain attacks [6]. Using these results, we use a stateless filter that eliminates records where $\mathsf{succ} = 0$. The resulting log files are then tested using the Shapiro-Wilk normality test [22] to identify those log files where the observed graph and largest component size are outside the expected range for a Gaussian distribution.

The initial training data consisted of 7,200 log files for the five business days between February 11–15, 2008. Source data was chosen exclusively from 1200GMT to 2359GMT for each day, a period corresponding to peak activity for the network. After filtering, the resulting set consisted of 5,619 log files from a source set of 7,200.

Table 1. Summary of Gaussian state variables in SSH training set

State variable x	Range	$\mu_X \pm \sigma_X$
g		299.27±42.49
c		35.13±21.32
h		6.88±0.35

Applying this filtering technique in order to isolate benign traffic yields a vector $\Lambda_1 \ldots \Lambda_m$ of log files, each representing benign traffic in a 30s interval. State values are calculated for each log file in this vector; we refer to the resulting vector of state values using the same subscript notation, *e.g.*, $r(\Lambda_i) = r_i$. We refer to the complete vector of values for a vector of log files by the corresponding capital letter (*e.g.*, $G = \{g(\Lambda_1) \ldots g(\Lambda_m)\}$).

We examined the H and D distributions in the filtered data to see if they could be modeled via a Gaussian distribution. (Our previous work already established that G and C are Gaussian for the monitored network [6].) Using the Shapiro-Wilk statistic (W) [22], we found that H had $W = 0.97$ and negligible p-value, and so we treated entropy as Gaussian. D had a Shapiro-Wilk statistic of $W = 0.77$ with negligible p-value, and consequently was not considered Gaussian.

Table 1 summarizes the Gaussian state variables, *i.e.*, g, c, and h. This table shows the summary data (left hand column), the mean and standard deviation (right side) and a sparkline for each data set. The sparkline is a time series plot of the activity over the training period. We plot the mean and shade an area one standard deviation from the mean in each sparkline.

For these three state variables, we can use (2) to calculate the detection threshold. For a given false positive rate, FPR, the corresponding threshold for a Gaussian IDS x is given by:

$$\theta_x = \mu_X + \sqrt{2}\mathrm{erf}^{-1}(\mathsf{FPR})\sigma_X \tag{2}$$

where erf is the error function [14], μ_X is the arithmetic mean of the vector of observations X, and σ_X is the standard deviation of the same vector.

The detection threshold for d is computed differently since, as shown above, d is not normally distributed over the sample space. We use d's maximum observed value over the benign log files as the detection threshold:

$$\theta_d = \max(D) \tag{3}$$

The detection threshold for r is prescribed by Jung *et al.* to be

$$\theta_r = \frac{\beta \ln \frac{\beta}{\alpha} + (1-\beta) \ln \frac{1-\beta}{1-\alpha}}{t_1 \ln \frac{t_1}{t_0} + (1-t_1) \ln \frac{1-t_1}{1-t_0}} \tag{4}$$

Here, α and β are user-configured thresholds for the maximum false positive rate (α) and the minimum true positive rate (β). For this work, we set $\beta = 1 - \alpha$, and set α to our acceptable FPR (see below). t_0 and t_1 are, respectively, the probabilities that a normal user will successfully communicate with a target, and the probability that a randomly scanning attacker will successfully communicate with a target. Per these definitions, t_0 and t_1 depend on a variety of factors including the density of targets in the network, the type of protocol involved, and dynamic addressing, some of which are difficult to accurately evaluate for the monitored network due to our limited view of it. However, Jung's simulation analysis of TRW [10] suggest that choices of t_0 and t_1 have relatively little impact on performance. As such, we generally adopt Jung's original values of $t_0 = 0.8$ and $t_1 = 0.2$ and will examine the impact of alternative θ_r values in §5.2.

Recall that based on our previous work on graph-based anomaly detection [6], we monitor traffic over 30s periods. This 30s period governs the effective response time of the entire intrusion detection and response mechanism — an IDS sends out at most one alert in a period, and defenders respond to changes at that time. If we constrain the *aggregate* false positives for *all* of the detectors to one false alarm per eight hours (*i.e.*, the duration of a typical network analyst's shift), this yields a combined rate of 0.1% for the five IDS together. We solve for individual false positive rates FPR using

$$0.001 = 1 - (1 - \mathsf{FPR})^5 \tag{5}$$

Plugging this value of FPR into (2) yields detection thresholds $\theta_g = 447$, $\theta_c = 110$, and $\theta_h = 8.105$, and setting $\alpha = \mathsf{FPR}$ in (4) yields $\theta_r = 6$. We also use the value $\theta_d = 150$, computed directly from (3). These are the thresholds we use in our evaluations in subsequent sections. Equation 5 treats each IDS as a statistically independent. While not true, this simplifies our exploratory analysis.

4 Observable Attack Spaces and Detection Probability

In §3, we developed and configured a combined IDS based around five different state variables: graph size g, largest component size c, server address entropy h, maximum client degree d and maximum failed connection run r. In doing so, we specifically configured these systems to yield a low false positive rate, resulting in one false positive per eight-hours as predicted by our training data. Now that we have developed this hybrid IDS, we can evaluate its efficacy for deterring attackers.

In order to do this, we develop a method for describing attacker utility which we call the *observable attack space* (OAS). An observable attack space describes the range of attacks that an attacker can conduct *as observed by a particular logging mechanism*. In this section, we develop an observable attack space common to our logging system (NetFlow) and our five candidate IDS. Using this approach, we model the aggregate *detection surface* of the OAS and use this to evaluate both our combined IDS and the constituent IDS individually.

This section is structured as follows. §4.1 describes OAS, IDS and the estimation of detection surfaces. §4.2 then compares the effectiveness of our five detection methods both in aggregate and as individual detection schemes.

4.1 OAS and Detection Surface

The type of log data that an IDS uses strongly impacts the types of attacks that an IDS can detect. An example of this is the impact of choosing NetFlow. NetFlow is a compact representation of traffic that is viable to collect on large networks, but since it lacks payload, signature-matching techniques are not possible with this log format. An observable attack space is therefore a parameterized representation of all possible forms of a particular attack, as observable using a particular form of log data. For this work, the observable attack space has two attributes: *aggressiveness* (a) and *success* (s). The aggressiveness is a natural number describing the number of distinct addresses with which the attacker communicates in the observation period. The success of an attack is the fraction of these communications that were successful, and is within the range $[0, 1]$.

When conducting simulations, we limit a to the range of $(0, \theta_d)$ because we treat the d IDS as deterministic — it will trigger if *and only if* $a \geq \theta_d$. In doing so, we ignore the possibility that during an attack, a benign host contacts more than θ_d addresses, thus "accidentally" causing a true detection even though $a < \theta_d$. This treatment also presumes that the attack is launched from a bot that is not also contributing benign traffic at the same time, *i.e.*, $a < \theta_d$ implies that the bot host does, in fact, contact fewer than θ_d addresses in a 30s interval. The other IDS' chances of detecting attacks are not so directly dependent on an attack's characteristics within the OAS.

Consider a particular IDS $x \in \{g, c, h, r\}$. Given an arbitrary log file of control data Λ^{ctl} that we are confident does not contain an attack, $\mathcal{P}^{x}_{\mathsf{det}}(a, s)$ is the probability that the IDS x raises an alarm for the log file resulting from Λ^{ctl} merged with an attack Λ^{atk} with aggressiveness a and success s. That is,

$$\mathcal{P}^{x}_{\mathsf{det}}(a, s) = \mathbb{P}\left[x(\Lambda^{\mathsf{atk}} \cup \Lambda^{\mathsf{ctl}}) \geq \theta_x\right] \tag{6}$$

where the probability is taken with respect to the selection of Λ^{ctl} and the generation of Λ^{atk} with aggressiveness a and success rate s. For a particular IDS x, the *detection surface of* x is the surface of values $\mathcal{P}^{x}_{\mathsf{det}}(a, s)$ for $a \in (0, \theta_d)$ and $s \in [0, 1]$.

More specifically, to estimate the probability of detection and the corresponding detection surface, we evaluate the distribution of state variables for normal behavior merged with randomly generated attacks meeting the aggressiveness and success requirements specified by a and s. For this paper, we limit our simulations to $a \in \{10, 20, 30, 40, \ldots, 140\}$ (recall $\theta_d = 150$) and $s \in \{0.1, 0.2, 0.3, \ldots, 1.0\}$. At each point, we conduct 100 simulations, each using one of fifty randomly selected 30s periods from the week of February 18–22 (the week following that used for training) for Λ^{ctl}. Λ^{atk} is randomly generated for

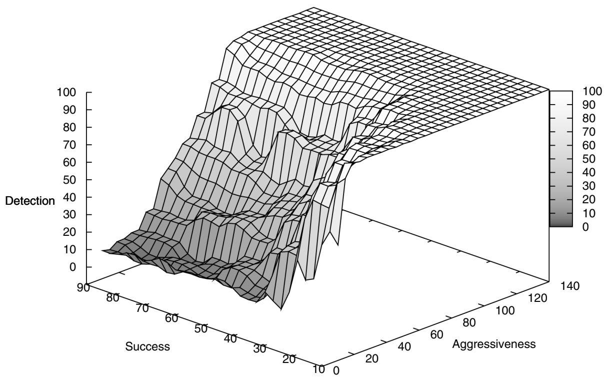


Fig. 1. Detection surface ($\mathcal{P}_{\mathsf{det}}^{\mathsf{all}}(a, s)$, as a percentage) for combined IDS

each simulation. Λ^{atk} contains a unique records, where each record has the same client address, and a different server address. The composition of the server addresses is a function of s: $a \cdot s$ addresses are chosen from a hit list of internal SSH servers identified in the training data[4] in order to approximate hit-list attacks; the remainder are sent to addresses with no listening server. We then merge Λ^{atk} with a randomly selected control log Λ^{ctl} and then calculate the state variables.

Four of the IDS examined by this paper (g, c, h, and d) are unaffected by the order of log records within the monitored 30s period. The fifth, r, is order-sensitive, however, in that TRW triggers an alert if any host is observed making more than θ_r failed connections *in a row*. This order sensitivity is a weakness, since an attacker can interleave scanning with connections to known hosts in order to avoid a failed connection run greater than θ_r [12]. To address this particular exploit, we randomly permute the records originating in each 30s interval. After this permutation, r is calculated for each host in the network.

Figure 1 plots the detection surface for all the IDS combined. As this figure shows, the combined detection mechanism generally increases proportionally to the aggressiveness of the attack and inversely relative to the success of the attack. Furthermore, the detection mechanisms tend to vary more as a function of the aggressiveness than due to the success rate.

The effectiveness of the aggregate IDS may be considered low, in the sense that an attacker with a relatively small hit list ($a = 40, s = 0.5$) can communicate with the network with little fear of detection. However, we should note that the attacks represented by this OAS are the most subtle space of attacks available. Our own experience indicates that the majority of attacks are orders of magnitude more aggressive than these hypothetical SSH scans, at which point *any* IDS will identify them. This latter point is particularly critical. As Figure 1 shows, once $a \geq 100$, the combined IDS will raise an alarm.

[4] This hit list is composed of all internal addresses in the training data which had one flow originating from them on port 22 and with a payload of greater than 1kB.

4.2 Detection Surface Comparison

In addition to the detection surface for the aggregate IDS, we have also calculated the detection surfaces for each component IDS in this system. We can use these results to evaluate the comparative effectiveness of each IDS.

Figure 2 plots detection surfaces for each IDS $x \in \{g, c, h, r\}$ as *contour plots*. A contour plot maps a 3-dimensional surface into a 2-dimensional representation using *contour lines*. Each contour line represents the axis coordinates where the surface takes on its labeled value.

These plots show that the most successful individual IDS are c and r : these IDS are the only ones to have significant ($\geq$ 10%) detection rates over the majority of the OAS. In contrast, the h IDS has a very *low* detection rate, less than 6% over the entire OAS. Of particular interest with c and r is their relative disconnectedness to each other: r's detection rate is dependent on s and less dependent on a. Conversely, c is largely independent of s, while a plays larger role in detection.

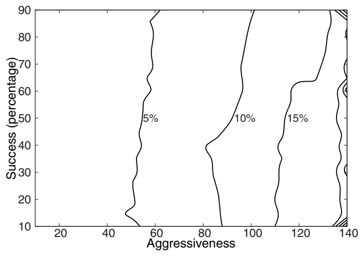
These IDS are calibrated to have an effective false positive rate of zero. As a result, they are largely insensitive to anomalies and have a relatively low detection rate. In addition, as noted above, the attacks represented here are extremely subtle. More aggressive attackers would be identified and eliminated *regardless* of the detection strategy used — by the time an attacker communicates with $\theta_d = 150$ addresses, the d IDS will raise an alarm, making other approaches effectively moot.

This phenomenon is partly observable in our models in Table 1. Recall that, for example, the model of graph size g, was 299 ± 42.47 hosts. If $g(\Lambda) = 299$ for some log file Λ, then an attacker will not trigger an anomaly until he has communicated with at least 149 hosts, at which point he is close to triggering d as well as g.

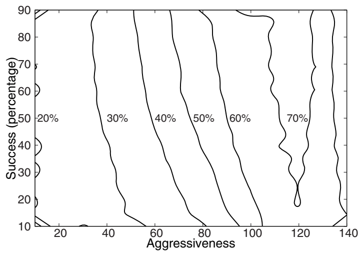
5 Modeling Acquisition

In §4.2 we examined the efficacy of the detection mechanisms purely as detectors: for a fixed false positive rate, we calculated the effective true positive rate. In this section, we use the detection surface in Figure 1 to examine the impact of IDS on *acquisition attacks*. We evaluate the efficacy of the detection surface by building a mathematical model for attacker payoff during an acquisition attack. Applying this model to the surface, we can determine how many hosts an attacker can expect to take over, and from these results determine how effective an IDS has to be in order to keep attackers from taking over hosts.

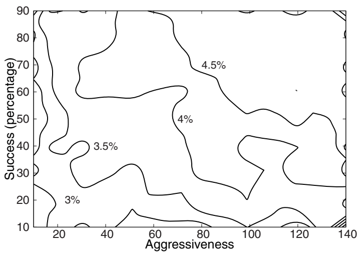
This section is divided as follows: §5.1 describes our model for acquisition attacks. §5.2 compares IDS efficiency using our payoff function. §5.3 considers the problem of IDS evaluation from a different perspective — instead of calculating efficiency in terms of true and false positives, we determine the minimum false positive rate required to counter an attacker.



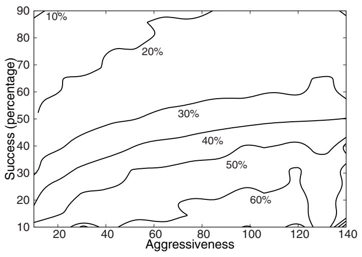
(a) Total graph size g



(b) Largest component size c



(c) Server entropy h



(d) Maximum failed connection run r

Fig. 2. Detection surfaces ($\mathcal{P}^x_{\mathsf{det}}(a, s)$, as a percentage) for individual IDS

5.1 Acquisition Payoff Model

We define an acquisition attack as a game between two parties who are competing for ownership of a single network. The two parties in the game are the *attacker*, who attempts to take over hosts on the network, and the *defender*, who attempts to prevent takeover of hosts on the network. In this game, the attacker uses a single *bot* to perform a series of *attempts*, during each of which the bot communicates with multiple hosts within the network using a *hit list* acquired previous to the attack.

In each attempt, the attacker communicates with some number of addresses (specified by the attacker's a), each of which has s chance of succeeding. For the purposes of the simulations, a successful attack is one that communicates with a real host, and a failed attack is one that communicates with a nonexistent host. That is, we assume that if an attacker talks with a host, the attacker takes the host over. The *payoff* of an attempt, $\mathcal{H}_{\mathsf{acq}}$, is the expected number of hosts with which the attacker communicates during an attempt.

The goal of the defender is to minimize $\mathcal{H}_{\mathsf{acq}}$, and the goal of the attacker to maximize the same. To do so, the defender deploys an IDS x, and so the probability of detecting a particular attempt with aggressiveness a and success rate s is $\mathcal{P}^x_{\mathsf{det}}(a, s)$. We assume that once the defender successfully identifies an

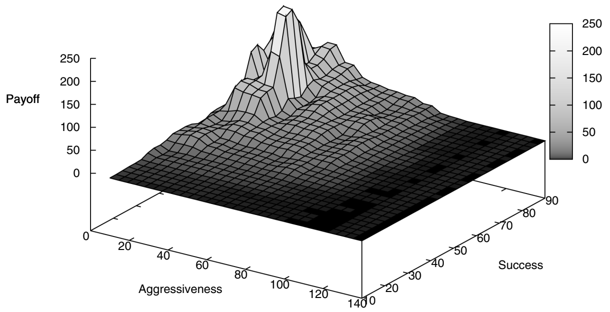


Fig. 3. Payoff $\mathcal{H}_{\mathsf{acq}}^{\mathsf{all}}(a, s, k_{\max}^{\mathsf{all}}(a, s))$ for acquisition attacks for combined IDS

attacker, it will block the attacker's bot, ending all further acquisition attempts by that bot. Furthermore, the defender will then recover all of the hosts that the bot communicated with during the game.

We note that this model assumes that the attacker and defender are perfect. That is, the probability that an attacker takes over a host that it contacts, and the probability that a defender correctly identifies an occupied host after being notified of an attack, are both one. The model can be modified by incorporating additional probabilities for measuring the attacker's takeover success per host contact and the defender's vigilance.

Let owned be a random variable indicating the number of hosts taken over, and let $\mathsf{alarmed}$ be the event that the bot is detected. Below, we assume that the probability of detection in each attempt is independent. If such is the case, then we can derive the payoff for an attack comprised of k attempts and for an IDS x as:

$$\begin{aligned}\mathcal{H}_{\mathsf{acq}}^{x}(a, s, k) &= \mathbb{E}[\mathsf{owned}] \\ &= \mathbb{P}[\mathsf{alarmed}]\,\mathbb{E}[\mathsf{owned} \mid \mathsf{alarmed}] + \mathbb{P}[\neg\mathsf{alarmed}]\,\mathbb{E}[\mathsf{owned} \mid \neg\mathsf{alarmed}] \\ &= (1 - \mathcal{P}_{\mathsf{det}}^{x}(a, s))^{k}(ask) \end{aligned} \tag{7}$$

The last step follows by taking $\mathbb{E}[\mathsf{owned} \mid \mathsf{alarmed}] = 0$, since we presume that if the defender detects an attacker during an attempt, then the defender recovers *all* of the hosts the attacker has communicated with using that particular bot. Note that the attacker maximizes his payoff by maximizing k subject to $\mathcal{H}_{\mathsf{acq}}(a, s, k) - \mathcal{H}_{\mathsf{acq}}(a, s, k-1) > 0$ or, in other words,

$$k < \frac{1 - \mathcal{P}_{\mathsf{det}}^{x}(a, s)}{\mathcal{P}_{\mathsf{det}}^{x}(a, s)} \tag{8}$$

We denote this value of k by $k_{\max}^{x}(a, s)$.

Figure 3 plots the payoff over the observed attack space using (7) with the maximum k satisfying (8). As this figure shows, aggressive attacks have a minimal payoff, a result that can be expected based on Figure 1. Above approximately $a \geq 80$, the attacker is consistently identified and stopped regardless of their success rate.

This behavior is the result of the interaction of two detectors: c and r. As s increases, the probability of the attacker combining previously separate components of the protocol graph increases, increasing the likelihood of detection by the c IDS. As the attacker's success rate decreases, he is more likely to generate a sufficiently long failed connection run to trigger the r detector. The other detectors will identify attackers, however their effectiveness is limited for attacks that are this subtle — an attacker who does disrupt g or h will typically already have disrupted d.

5.2 Calculating IDS Efficiency

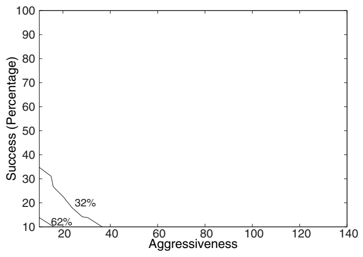
We can use (7) to also calculate a comparative efficiency metric for IDS. The volume under the surface specified by (7) is the ability of the attacker to take over hosts in the presence of a particular IDS. The *efficiency* of an IDS x is therefore the indicator of how much x reduces an attacker's payoff. We can express IDS efficiency as the ratio between the number of hosts an attacker expects to take over in the presence of an IDS x and the number of hosts the attacker can take over (in the same number of attempts) in that IDS' absence.

$$\mathcal{E}^x_{\mathsf{acq}} = 1 - \frac{\sum_{a \in (0,\theta_d)} \sum_{s \in (0,1]} \mathcal{H}^x_{\mathsf{acq}}(a, s, k^x_{\max}(a, s))}{\sum_{a \in (0,\theta_d)} \sum_{s \in (0,1]} ask^x_{\max}(a, s)} \tag{9}$$

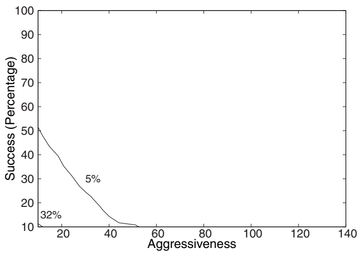
The subtraction in (9) is included simply to provide an intuitive progression for efficiency: if $\mathcal{E}$ is greater for IDS A than IDS B, then A is a better IDS than B. Based on (9), we can calculate an efficiency of 0.14 for g, 0.0099 for h, 0.73 for c and 0.22 for r. The effectiveness of the combined detector is 0.80.

Using Equation 9 we can examine the impact of alternative values for θ_r. Recall from §3.3 that θ_r is based on models of normal behavior and attacker behavior that can vary as a function of the protocol, the density of the observed network and other behaviors. Without revisiting the model, we can simply change the values of θ_r and examine how that changes the efficiency. In this case, we find that for $\theta_r = 3$, 4, and 5, $\mathcal{E}^x_{\mathsf{acq}} = 0.50$, 0.37, and 0.29, respectively. The most interesting result from these calculations is the relatively low efficiency of r as an IDS for acquisition attacks, despite its relatively good true positive rates (Figure 2). Because the detection mechanism relies on attacker failures, it is better at detecting attacks which have a relatively low s. IDS r is therefore very good at detecting attacks with low payoff.

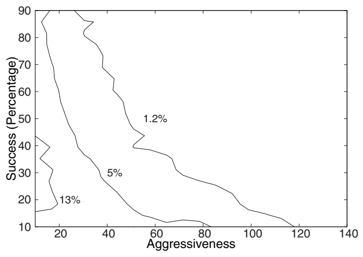
We expect that the comparative efficiency of these IDS will differ from one protocol to the next. g and h are affected by the aggregate traffic for one protocol, *e.g.*, the total number of hosts using a particular protocol. Conversely, r relies exclusively on per-host behavior. Consequently, using protocols with more clients or servers (such as HTTP) should result in less g and h efficiency, while r should have the same efficiency regardless of protocol.



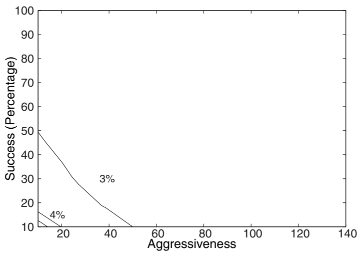
(a) Total graph size g



(b) Largest component size c



(c) Server entropy h



(d) Maximum failed connection run r

Fig. 4. False positive rates required to limit $\mathcal{H}^x_{\mathsf{acq}}(a, s, k) = 1$

5.3 Determining a Minimum False Positive Rate

As Figure 3 implies, even with all the detection mechanisms operating, attackers can still acquire a high rate of return with a sufficiently subtle hit-list attack. In this section, we will now address the question of detection from a different perspective: how high a false positive rate do we have to tolerate in order to prevent the attacker from seriously compromising the monitored network?

To do so, we invert (7) so that instead of calculating the attacker's payoff as a function of detectability, we calculate the probability of detection as a function of payoff. Solving for $\mathcal{P}^x_{\mathsf{det}}(a, s)$ in (7) yields

$$\mathcal{P}^x_{\mathsf{det}}(a, s) = 1 - \sqrt[k]{\frac{\mathcal{H}^x_{\mathsf{acq}}(a, s, k)}{ask}} \tag{10}$$

Suppose the defender wishes to minimize $\mathcal{P}^x_{\mathsf{det}}(a, s)$ (and hence also the false alarm rate) while restricting $\mathcal{H}^x_{\mathsf{acq}}(a, s, k) \leq 1$, and so the attacker wishes to maximize $\mathcal{P}^x_{\mathsf{det}}(a, s)$ in order to achieve $\mathcal{H}^x_{\mathsf{acq}}(a, s, k) = 1$. The attacker does so by choosing k so as to minimize $(ask)^{-1/k}$, for any a and s.

Using this strategy, we calculate the detection probability required to identify and stop attackers at points within the OAS. To calculate the resulting detection thresholds for each IDS, we use our simulated attacks with parameters a and s to calculate the threshold needed to filter off $\mathcal{P}^x_{\mathsf{det}}(a,s)$ of the attacks when overlaid on our training data.

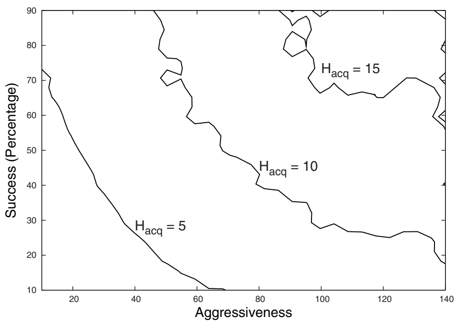


Fig. 5. Values of a and s for which $\mathcal{H}^c_{\mathsf{acq}}(a,s,k)$ can be limited to at most the specified value, using a threshold $\theta_c = \mu_C + 3.5\sigma_C$

The results of these runs are given in Figure 4. These figures are contour plots over the OAS as before. However, the contours for the figure are the false positive rates that would result from this analysis. For the g, c and h detectors, these values are calculated using (2). For r, this value is calculated by using (4).

As Figure 4 indicates, anomaly detection systems that are capable of defending against subtle attacks will require extremely high false positive rates. Recall that our measurement system conducts a test every 30s; for every 1% false positive rate we accept, we pay 10 alerts per eight-hour shift. As such, this figure indicates that the false positive rates for building systems that can limit the attacker to $\mathcal{H}^x_{\mathsf{acq}}(a,s,k) \leq 1$ are much higher than we can consider accepting.

One way to avoid such high false positive rates would be to not place such a stringent limit of $\mathcal{H}^x_{\mathsf{acq}}(a,s,k) \leq 1$. For example, if the defender insists on a near-zero false positive rate, we can determine if there is a higher threshold for the payoff that can accommodate this rate, such as $\mathcal{H}^x_{\mathsf{acq}}(a,s,k) \leq 5$. Figure 5 shows this for the c IDS, for $\mathcal{H}^c_{\mathsf{acq}}(a,s,k) \in \{5,10,15\}$. Specifically, each contour line shows the values of a and s for which $\mathcal{H}^c_{\mathsf{acq}}(a,s,k)$ can be limited to at most the specified value, using a threshold $\theta_c = \mu_C + 3.5\sigma_C$, which is large enough to ensure a false positive rate very close to zero. As this figure shows, the defender can effectively impose an upper limit on the attacker's payoff, but unfortunately this limit must be rather large ($\mathcal{H}^c_{\mathsf{acq}}(a,s,k) = 15$) in order to cover the majority of the attack space.

From Figures 4 and 5, we conclude that in order for an anomaly detection system to be a viable deterrent to host compromise, it must either use finer resolution data than NetFlow, develop mechanisms for coping with a high false positive rate, or permit higher attacker payoff than would be ideal.

6 Modeling Reconnaissance

In this section, we develop an alternative attack scenario, *reconnaissance*, where the attacker scouts out the network with his bots. In each attack, he communicates with addresses to simply determine the presence of hosts at certain addresses. The reconnaissance scenario differs from the acquisition scenario by the attacker's knowledge and goals. Specifically, the attacker's goal is to contact as many addresses as possible within a short period. To do so, the attacker uses a *chaff hit list* consisting of hosts that the attacker already knows about, and a target space of addresses to probe. The chaff hit list reduces the attacker's probability of detection by lowering his failure rate. However, it also reduces the attacker's payoff by requiring him to "sacrifice" a certain number of targets every round.

Let $\mathsf{alarmed} = i$ be the event that the bot is detected at the end of attempt i (and before attempt $i+1$); as before, an attempt is comprised of contacting a addresses with success rate of s (in this case, owing to the chaff hit list). Let $\mathsf{scanned}$ denote a random variable indicating the number of scans that one bot performs successfully (*i.e.*, determines whether the scanned address has a listening service or not), not counting the "chaff" that it introduces to avoid detection. Note that we suppose that the number of listening services the bot finds is sufficiently small that it does not relieve the bot from introducing a fraction s of chaff scans. We also presume that the probability the bot is detected in each attempt is independent.

$$\begin{aligned}
\mathcal{H}^x_{\mathsf{rec}}(a,s) &= \mathbb{E}\left[\mathsf{scanned}\right] \\
&= \sum_{i=1}^{\infty} \mathbb{P}\left[\mathsf{alarmed} = i\right] \mathbb{E}\left[\mathsf{scanned} \mid \mathsf{alarmed} = i\right] \\
&= \sum_{i=1}^{\infty} \left((1-\mathcal{P}^x_{\mathsf{det}}(a,s))^{i-1}\mathcal{P}^x_{\mathsf{det}}(a,s)\right)(ia(1-s)) \\
&= a(1-s)\frac{\mathcal{P}^x_{\mathsf{det}}(a,s)}{1-\mathcal{P}^x_{\mathsf{det}}(a,s)} \sum_{i=1}^{\infty} i(1-\mathcal{P}^x_{\mathsf{det}}(a,s))^i \\
&= a(1-s)\frac{\mathcal{P}^x_{\mathsf{det}}(a,s)}{1-\mathcal{P}^x_{\mathsf{det}}(a,s)} \frac{1-\mathcal{P}^x_{\mathsf{det}}(a,s)}{\mathcal{P}^x_{\mathsf{det}}(a,s)^2} \\
&= a(1-s)\frac{1}{\mathcal{P}^x_{\mathsf{det}}(a,s)} \qquad (11)
\end{aligned}$$

Applying (11) to the detection matrix over our OAS results in the payoff plot shown in Figure 6. This figure plots the aggregate payoff over the OAS for reconnaissance. Of particular note with this result is that it demonstrates that a sufficiently motivated and subtle attacker can scan a network by subtly exploiting attacks with high s rates. In this case, the attacker can slowly scan the network for an extended period — the observed peak at the $a = 20$ segment of the graph implies that the attacker scans for 25 minutes before being detected.

However, the attacker can achieve just as effective results by aggressively scanning the network. Recall that the effective aggressiveness of the attacker is bound by θ_d to less than 150 nodes. In the reconnaissance scenario, the attacker faces

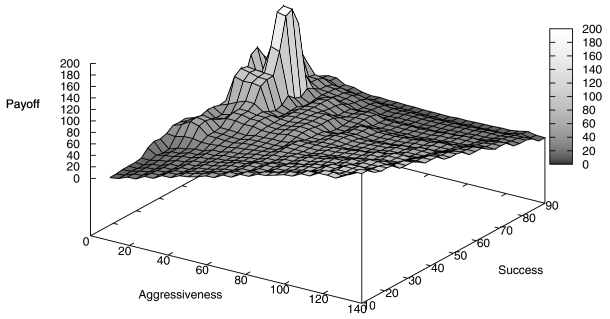


Fig. 6. Payoff $\mathcal{H}^{\mathsf{all}}_{\mathsf{rec}}(a, s)$ for reconnaissance attacks for combined IDS

no penalty for scanning at a higher aggressiveness rate, since the defender can only block an address. Consequently, this plot can continue out to whatever the practical upper limit for a is, a result which would correspond to the aggressive scanning we observe right now.

7 Conclusion

In this paper we have developed a new method for evaluating the performance of IDS based on an observable attack space, specifically the view of a harvesting or scanning attack that is available in flow logs that lack payload data. Our approach complements ROC-based analysis by enabling the creation of detection surfaces — models of an IDS' ability to detect different attacks. Moreover, we augment this analysis with a payoff-based metric. By incorporating payoffs, we are better able to characterize the deterrence offered by an IDS. In particular, instead of describing the detection of a system in terms of pure false positive and false negative rates, we are able to use payoff functions to calculate the gain that an attacker can expect from a certain type of attack. This also enables us to determine how high a false positive rate we must endure in order to limit the attacker's payoff to a target value.

Future work will focus on expanding the OAS approach to address different scenarios and parameters. First, our previous work on graph-based intrusion detection [6] considered the possibility of multiple bots being active simultaneously, and extending our OAS to account for this is a natural direction of future work. Second, generalizing from 30-second traffic samples to an approach considering multiple sample durations may provide additional detection capability [21]. Third, this work outlines two initial attack scenarios: harvesting and reconnaissance. However, a variety of other attacks may be considered and evaluated. In particular, different scanning strategies (such as topological scanning), bot command-and-control, and DDoS attacks all merit further investigation and similar payoff-based evaluation.

Several useful and, in some cases, discouraging results fall out of our analysis techniques as applied to SSH traffic observed on a very large network. For example, in §4.2 our analysis elucidated the complementary capabilities of detection using the size c of the largest component of a protocol graph [6] and the TRW scan detector r [11]. Consequently, there is good reason to use both simultaneously. Moreover, we showed that these detectors significantly outperform the server address entropy detector h, the graph-size detector g, and the degree-based detector d, for the stealthy attacks that form our observable attack space. That said, using our payoff analysis for acquisition attacks, we showed in §5.2 that r detection is primarily effective at detecting acquisition attacks with low payoff for the attacker, and so its utility for acquisition attacks is less compelling. In addition, we showed in §5.3 that to severely limit the attacker's acquisitions, the false positive rates that would need to be endured by any of the detectors we considered would be significant and, for a network of the size we studied, impractical. We showed how to derive more relaxed payoff limits that would enable near-zero false positive rates for an IDS.

References

1. Alata, E., Nicomette, V., Kaaniche, M., Dacier, M., Herrb, M.: Lessons learned from the deployment of a high-interaction honeypot. In: Proceedings of the 2006 European Dependable Computing Conference (2006)
2. Axelsson, S.: The base rate fallacy and the difficulty of intrusion detection. ACM Transactions on Information and System Security 3(3), 186–205 (2000)
3. Binkley, J.: An algorithm for anomaly-based botnet detection. In: Proceedings of the 2006 USENIX Workshop on Steps for Reducing Unwanted Traffic on the Internet (SRUTI) (2006)
4. Cárdenas, A., Baras, J., Seamon, K.: A framework for evaluation of intrusion detection systems. In: Proceedings of the 2006 IEEE Symposium on Security and Privacy (2006)
5. Claffy, K., Braun, H., Polyzos, G.: A parameterizable methodology for internet traffic flow profiling. IEEE Journal on Selected Areas in Communications 13(8), 1481–1494 (1995)
6. Collins, M.P., Reiter, M.: Hit-list worm detection and bot identification in large networks using protocol graphs. In: Kruegel, C., Lippmann, R., Clark, A. (eds.) RAID 2007. LNCS, vol. 4637, pp. 276–295. Springer, Heidelberg (2007)
7. Collins, M.P., Reiter, M.K.: Anomaly detection amidst constant anomalies: Training IDS on constantly attacked data. Technical Report CMU-CYLAB-08-006, Carnegie Mellon University, CyLab (2008)
8. Gaffney, J., Ulvila, J.: Evaluation of intrusion detectors: A decision theory approach. In: Proceedings of the 2001 IEEE Symposium on Security and Privacy (2001)
9. Gates, C., Taylor, C.: Challenging the anomaly detection paradigm, a provocative discussion. In: Proceedings of the 2006 New Security Paradigms Workshop, pp. 22–29 (2006)
10. Jung, J.: Real-Time Detection of Malicious Network Activity Using Stochastic Models. PhD thesis, Massachuesetts Institute of Technology (2006)

11. Jung, J., Paxson, V., Berger, A.W., Balakrishnan, H.: Fast portscan detection using sequential hypothesis testing. In: Proceedings of the 2004 IEEE Symposium on Security and Privacy (2004)
12. Kang, M., Caballero, J., Song, D.: Distributed evasive scan techniques and countermeasures. In: Hämmerli, B.M., Sommer, R. (eds.) DIMVA 2007. LNCS, vol. 4579, pp. 157–174. Springer, Heidelberg (2007)
13. Killourhy, K., Maxion, R., Tan, K.: A defense-centric taxonomy based on attack manifestations. In: Proceedings of the 2004 Conference on Dependable Systems and Networks (DSN) (2004)
14. Kreyszig, E.: Advanced Engineering Mathematics, 9th edn. J. Wiley and Sons, Chichester (2005)
15. Lakhina, A., Crovella, M., Diot, C.: Mining anomalies using traffic feature distributions. In: Proceedings of the 2005 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM), pp. 217–228 (2005)
16. Lippmann, R., Fried, D., Graf, I., Haines, J., Kendall, K., McClung, D., Weber, D., Webster, S., Wyschogrod, D., Cunningham, R., Zissman, M.: Evaluating intrusion detection systems: The 1998 DARPA off-line intrusion detection evaluation. In: Proceedings of the DARPA Information Survivability Conference and Exposition (2000)
17. Maxion, R., Tan, K.: Benchmarking anomaly-based detection systems. In: Proceedings of the 2000 Conference on Dependable Systems and Networks (DSN) (2000)
18. McHugh, J.: Testing intrusion detection systems: A critique of the 1998 and 1998 DARPA intrusion detection system evaluations as performed by Lincoln Laboratory. ACM Transactions on Information and Systems Security 3(4), 262–294 (2000)
19. Northcutt, S.: Network Intrusion Detection: An Analyst's Handbook. New Riders (1999)
20. Paxson, V.: Bro: A system for detection network intruders in real time. In: Proceedings of the 2008 Usenix Security Symposium (1998)
21. Sekar, V., Xie, Y., Reiter, M.K., Zhang, H.: A multi-resolution approach for worm detection and containment. In: Proceedings of the 36th International Conference on Dependable Systems and Networks, June 2006, pp. 189–198 (2006)
22. Shapiro, S., Wilk, M.: An analysis of variance test for normality (complete samples). Biometrika 52(3–4), 591–611 (1965)
23. Staniford-Chen, S., Cheung, S., Crawford, R., Dilger, M., Frank, J., Hoagland, J., Levitt, K., Wee, C., Yip, R., Zerkle, D.: GrIDS – A graph-based intrusion detection system for large networks. In: Proceedings of the 19th National Information Systems Security Conference, pp. 361–370 (1996)
24. Stolfo, S., Fan, W., Lee, W., Prodromidis, A., Chan, P.: Cost-based modeling for fraud and intrusion detection: Results from the JAM project. In: Proceedings of the 2000 DARPA Information Survivability Conference and Exposition (2000)
25. Tan, K., Maxion, R.: The effects of algorithmic diversity on anomaly detector performance. In: Proceedings of the 2005 Conference on Dependable Systems and Networks (DSN) (2005)

Determining Placement of Intrusion Detectors for a Distributed Application through Bayesian Network Modeling

Gaspar Modelo-Howard, Saurabh Bagchi, and Guy Lebanon

School of Electrical and Computer Engineering, Purdue University
465 Northwestern Avenue, West Lafayette, IN 47907 USA
{gmodeloh,sbagchi,lebanon}@purdue.edu

Abstract. To secure today's computer systems, it is critical to have different intrusion detection sensors embedded in them. The complexity of *distributed* computer systems makes it difficult to determine the appropriate configuration of these detectors, i.e., their choice and placement. In this paper, we describe a method to evaluate the effect of the detector configuration on the accuracy and precision of determining security goals in the system. For this, we develop a Bayesian network model for the distributed system, from an attack graph representation of multi-stage attacks in the system. We use Bayesian inference to solve the problem of determining the likelihood that an attack goal has been achieved, *given* a certain set of detector alerts. We quantify the overall detection performance in the system for different detector settings, namely, choice and placement of the detectors, their quality, and levels of uncertainty of adversarial behavior. These observations lead us to a greedy algorithm for determining the optimal detector settings in a large-scale distributed system. We present the results of experiments on Bayesian networks representing two real distributed systems and real attacks on them.

Keywords: Intrusion detection, detector placement, Bayesian networks, attack graph.

1 Introduction

It is critical to provide intrusion detection to secure today's distributed computer systems. The overall intrusion detection strategy involves placing multiple detectors at different points of the system, at network ingress or combination points, specific hosts executing parts of the distributed system, or embedded in specific applications that form part of the distributed system. At the current time, the placement of the detectors and the choice of the detectors are more an art than a science, relying on expert knowledge of the system administrator.

The impact of the choice is significant on the accuracy and precision of the overall detection function in the system. The detectors are of different qualities, in terms of their false positive (FP) and false negative (FN) rates, some may have overlapping functionalities, and there may be many possible positions for deploying a detector. Therefore the entire space of exploration is large and yet not much exists today to serve as a scientific basis for the choices. This paper is a step in that direction.

In the choice of the number of detectors, more is not always better. There are several reasons why an extreme design choice of a detector at every possible network point, host, and application may not be ideal. First, there is the economic cost of acquiring, configuring, and maintaining the detectors. Detectors are well-known to need tuning to achieve their best performance and to meet the targeted needs of the application (specifically in terms of the false positive-false negative performance balance). Second, a large number of detectors would mean a large number of alert streams under attack as well as benign conditions. These could overwhelm the manual or automated process in place to respond to intrusion alerts. Third, detectors impose a performance penalty on the distributed system that they are meant to protect. The penalty arises because the detectors typically share the computational cycles and the bandwidth along with the application. Fourth, a system owner may have specific security goals, e.g., detecting a security goal may be very important and requires high sensitivity, while another may need to be done with less tolerance for false positives.

The problem that we address in this paper is, given the security goals in a system and a model for the way multi-stage attacks can spread in the system, how can we automatically and based on scientific principles, select the right set of detectors and their placements. Right is determined by an application-specific requirement on the true positive (TP) - true negative (TN) rate of detection in the system. We explore the space of the configuration of the individual detectors, their placement on the different hosts or network points, and their number.

Our solution approach starts with attack graphs, which are a popular representation for multi-stage attacks [9]. Attack graphs are a graphical representation of the different ways multi-stage attacks can be launched against system. The nodes depict successful intermediate attack goals with the end nodes representing the ultimate goal of an attack. The edges represent the relation that one attack goal is a stepping stone to another goal and will thus have to be achieved before the other. The nodes can be represented at different levels of abstraction, thus the attack graph representation can bypass the criticism that detailed attack methods and steps will need to be known a priori to be represented (which is almost never the case for reasonably complex systems). Research in the area of attack graphs has included automation techniques to generate these graphs [11], [25], to analyze them [14], [21], and to reason about the completeness of these graphs [14].

We model the probabilistic relation between attack steps and the detectors using the statistical Bayesian network formalism. Bayesian network is particularly appealing in this setting since it enables computationally efficient inference for the unobserved nodes—the attack goals—based on the observed nodes—the detector alerts. The important question that Bayesian inference can answer for us is, given a set of detector alerts, what is the likelihood that an attack goal has been achieved. Further the Bayesian network can be relatively easily created from an attack graph structure for the system, which we assume is given by existing methods.

We design an algorithm to systematically perform Bayesian inference and determine the accuracy and precision for determining that attack goals have been achieved. The algorithm then chooses the number, placement, and choice of detectors that gives the highest value of an application-specific utility function. We apply our technique to two specific systems—a distributed e-commerce system and a Voice-over-IP (VoIP) system and demonstrate the optimal choice under different conditions. The conditions we explore are different qualities of detectors, different level of knowledge of attack paths, and different threshold settings by the system administrator for determining if an attack goal is reached. Our exploration also shows that the value of a detector for determining an attack step degrades exponentially with distance from the site of the attack.

The rest of this document is organized as follows. Section 2 introduces the attack graphs model and provides a brief presentation of inference in Bayesian networks. Section 3 describes the model and algorithm used to determine an appropriate location for detectors. Section 4 provides a description of the systems used in our experiments. Section 5 presents a complete description of the experiments along with their motivations to help determine the location of the intrusion detectors. Section 6 presents related work and section 7 concludes the paper and discusses future work.

2 Background

2.1 Attack Graphs

An attack graph is a representation of the different methods by which a distributed system can be compromised. It represents the intermediate attack goals for a hypothetical adversary leading up to some high level attack goals. The attack goal may be in terms of violating one or more of confidentiality, integrity, or availability of a component in the system. It is particularly suitable for representing multi-stage attacks, in which a successful attack step (or steps) is used to achieve success in a subsequent attack step. An edge will connect the antecedent (or precondition) stage to the consequent (or postcondition) stage. To be accurate, this discussion reflects the notion of one kind of attack graph, called the exploit-dependency attack graph [11], [14], [25], but this is by far the most common type and considering the other subclass will not be discussed further in this paper.

Recent advances in attack graph generation have been able to create graphs for systems of up to hundreds and thousands of hosts [11], [25].

For our detector-location framework, exploit-dependency attack graphs are used as the base graph from which we build the Bayesian network. For the rest of this paper, the vertex representing an exploit in the distributed system will be called an attack step.

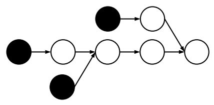


Fig. 1. Attack graph model for a sample web server. There are three starting vertices, representing three vulnerabilities found in different services of the server, from where the attacker can elevate the privileges in order to reach the final goal of compromising the password file.

2.2 Inference in Bayesian Networks

Bayesian networks [13] provide a convenient framework for modeling the relationship between attack steps and detector alerts. Using Bayesian networks we can infer which unobserved attack steps have been achieved based on the observed detector alerts.

Formally, a Bayesian network is a joint probabilistic model for n random variables $(x_1, \ldots, x_n)$ based on a directed acyclic graph $G = (V, E)$ where V is a set of nodes corresponding to the variables $V = (x_1, \ldots, x_n)$ and $E \subseteq VxV$ contains directed edges connecting some of these nodes in an acyclic manner. Instead of weights, the graph edges are described by conditional probabilities of nodes given their parents that are used to construct a joint distribution $P(V)$ or $P(x_1, \ldots, x_n)$.

There are three main tasks associated with Bayesian networks. The first is inferring values of variables corresponding to nodes that are unobserved given values of variables corresponding to observed nodes. In our context this corresponds to predicting whether an attack step has been achieved based on detector alerts. The second task is learning the conditional probabilities in the model based on available data which in our context corresponds to estimating the reliability of the detectors and the probabilistic relations between different attack steps. The third task is learning the structure of the network based on available data. All three tasks have been extensively studied in the machine learning literature and, despite their difficulty in the general case, may be accomplished relatively easily in the case of a Bayesian network.

We focus in this paper mainly on the first task. For the second task, to estimate the conditional probabilities, we can use characterization of the quality of detectors [20] and the perceived difficulty of achieving an attack step, say through risk assessment. We consider the fact that the estimate is unlikely to be perfectly accurate and provide experiments to characterize the loss in performance due to imperfections. For the third task, we rely on extensive prior work on attack graph generation and provide a mapping from the attack graph to the Bayesian network.

In our Bayesian network, the network contains nodes of two different types $V = V_a \bigcup V_b$. The first set of nodes V_a corresponds to binary variables indicating whether specific attack steps in the attack graph occurred or not. The second set of nodes V_b corresponds to binary variables indicating whether a specific detector issued an alert. The first set of nodes representing attack

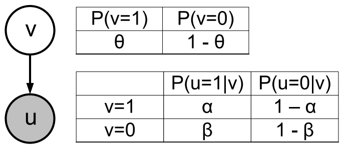


Fig. 2. Simple Bayesian network with two types of nodes: an observed node (u) and an unobserved node (v). The observed node correspond to the detector alert in our framework and its conditional probability table includes the true positive (α) and false positive (β).

steps are typically unobserved while the second set of nodes corresponding to alerts are observed and constitute the evidence. The Bayesian network defines a joint distribution $P(V) = P(V_a, V_b)$ which can be used to compute the marginal probability of the unobserved values $P(V_a)$ and the conditional probability $P(V_a|V_b) = P(V_a, V_b)/P(V_b)$ of the unobserved values given the observed values. The conditional probability $P(V_a|V_b)$ can be used to infer the likely values of the unobserved attack steps given the evidence from the detectors. Comparing the value of the conditional $P(V_a|V_b)$ with the marginal $P(V_a)$ reflects the gain in information about estimating successful attack steps given the current set of detectors. Alternatively, we may estimate the suitability of the detectors by computing classification error rate, precision, recall and Receiver Operating Characteristic (ROC) curve associated with the prediction of V_a based on V_b.

Note that the analysis above is based on emulation done prior to deployment with attacks injected through the vulnerability analysis tools, a plethora of which exist in the commercial and research domains, including integrated infrastructures combining multiple tools.

Some attack steps have one or more detectors that specifically measure whether an attack step has been achieved while other attack steps do not have such detectors. We create an edge in the Bayesian network between nodes representing attack steps and nodes representing the corresponding detector alerts. Consider a specific pair of nodes $v \in V_a, u \in V_b$ representing an attack step and a corresponding detector alert. The conditional probability $P(v|u)$ determines the values $P(v = 1|u = 0), P(v = 0|u = 1), P(v = 0|u = 0), P(v = 1|u = 1)$. These probabilities representing false negative, false positive, and correct behavior (last two) can be obtained from an evaluation of the detectors quality.

3 System Design

3.1 Framework Description

Our framework uses a Bayesian network to represent the causal relationships between attack steps and also between attack steps and detectors. Such relationships are expressed quantitatively, using conditional probabilities. To produce

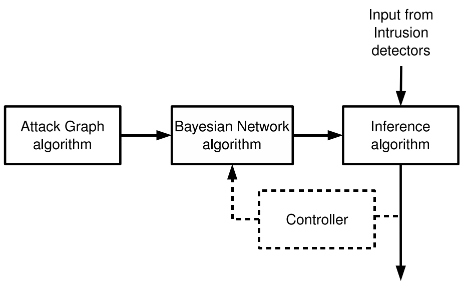


Fig. 3. A block diagram of the framework to determine placement of intrusion detectors. The dotted lines indicate a future component, controller, not included currently in the framework. It would provide for a feedback mechanism to adjust location of detectors.

the Bayesian network[1], an attack graph is used as input. The structure of the attack graph maps exactly to the structure of the Bayesian network. Each node in the Bayesian network can be in one of two states. Each attack stage node can either be achieved or not by the attacker. Each detector node can be in one of two states: alarm generated state or not. The leaf nodes correspond to the starting stages of the attack, which do not need any precondition, and the end nodes correspond to end goals for an adversary. Typically, there are multiple leaf nodes and multiple end nodes.

The Bayesian network requires that the sets of vertices and directed edges form a directed acyclic graph (DAG). This property is also found in attack graphs. The idea is that the attacker follows a monotonic path, in which an attack step does not have to be revisited after moving to a subsequent attack step. This assumption can be considered reasonable in many scenarios according to experiences from real systems.

A Bayesian network quantifies the causal relation that is implied by an edge in an attack graph. In the cases when an attack step has a parent, determined by the existence of an edge coming to this child vertex from another attack step, a conditional probability table is attached to the child vertex. As such, the probability values for each state of the child are conditioned by the state(s) of the parent(s). In these cases, the conditional probability is defined as the probability of a packet from an attacker that already achieved the parent attack step, achieving the child attack step. All values associated to the child are included in a conditional probability table (CPT). As an example, all values for node u in Figure 2 are conditioned on the possible states of its parent, node v. In conclusion, we are assuming that the path taken by the attacker is fully probabilistic. The attacker is following a strategy to maximize the probability of success, to reach the security goal. To achieve it, the attacker is well informed about the vulnerabilities associated to a component of the distributed system

[1] Henceforth, when we refer to a node, we mean a node in the Bayesian network, as opposed to a node in the attack graph. The clarifying phrase is thus implied.

and how to exploit it. The fact that an attack graph is generated from databases of vulnerabilities support this assumption.

The CPTs have been estimated for the Bayesian networks created. Input values are a mixture of estimates based on testing specific elements of the system, like using a certain detector such as IPTables [12] or Snort [28], and subjective estimates, using judgment of a system administrator. From the perspective of the expert (administrator), the probability values reflect the difficulty of reaching a higher level attack goal, having achieved some lower level attack goal.

A potential problem when building the Bayesian network is to obtain a good source for the values used in the CPTs of all nodes. The question is then how to deal with possible imperfect knowledge when building Bayesian networks. We took two approaches to deal with this issue: (1) use data from past work and industry sources and (2) evaluate and measure in our experiments the impact such imperfect knowledge might have.

For the purposes of the experiments explained in section 5, we have chosen the junction tree algorithm to do inference, the task of estimating probabilities given a Bayesian network and the observations or evidence. There are many different algorithms that could be chosen, making different tradeoffs between speed, complexity, and accuracy. Still, the junction tree engine is a general-purpose inference algorithm well suited for our experiments since it works under our scenario: allows discrete nodes, as we have defined our two-states nodes, in direct acyclic graphs such as Bayesian networks, and does exact inference. This last characteristic refers to the algorithm computing the posterior probability distribution for all nodes in network, given some evidence.

3.2 Algorithm

We present here an algorithm to achieve an optimal choice and placement of detectors. It takes as input (i) a Bayesian network with all attack vertices, their corresponding CPTs and the host impacted by the attack vertex; (ii) a set of detectors, the possible attack vertices each detector can be associated with, and the CPTs for each detector with respect to all applicable attack vertices.

Input: (i) Bayesian network $BN = (V, CPT(V), H(V))$ where V is the set of attack vertices, $CPT(V)$ is the set of conditional probability tables associated with the attack vertices, and $H(V)$ is the set of hosts affected if the attack vertex is achieved.

(ii) Set of detectors $D = (d_i, V(d_i), CPT[i][j])$ where d_i is the ith detector, $V(d_i)$ is the set of attack vertices that the detector d_i can be attached to (i.e., the detector can possibly detect those attack goals being achieved), and $CPT[i][j]$ $\forall j \in V(d_i)$ is the CPT tables associated with detector i and attack vertex j.

Output: Set of tuples $\theta = (d_i, \pi_i)$ where d_i is the ith detector selected and π_i is the set of attack vertices that it is attached to.

```
DETECTOR-PLACEMENT $(BN, D)$
1   System-Cost = 0
2   Sort all $(d_i, a_j), a_j \in V(d_i), \forall i$ by BENEFIT$(d_i, a_j)$. Sorted list = $L$
3   Length$(L) = N$
4   for $(i = 1 to N)$
5       System-Cost = System-Cost + COST$(d_i, a_j)$
6       /* COST$(d_i, a_j)$ can be in terms of economic cost, cost due
        to false alarms and missed alarms, etc. */
7       if (System-Cost > Threshold $\tau$) break
8       if $(d_i \in \theta)$ add $a_j$ to $\pi_i \in \theta$
9       else add $(d_i, \pi_i = a_j)$ to $\theta$
10  end for
11  return $\theta$
```

```
BENEFIT $(d, a)$
    /* This is to calculate the benefit from attaching detector $d$
    to attack vertex $a$ */
1   Let the end attack vertices in the $BN$ be $F = f_i, i = 1, \ldots, M$
2   For each $f_i$, the following cost-benefit table exists
3   Perform Bayesian inference with $d$ as the only detector
    in the network and connected to attack vertex $a$
4   Calculate for each $f_i$, the precision and recall, call them,
    Precision$(f_i, d, a)$, Recall$(f_i, d, a)$
5   System-Benefit = $\sum_{i=1}^{M}$ [Benefit$_{f_i}$(True Negative) $\times$ Precision$(f_i, d, a)$
                          + Benefit$_{f_i}$(True Positive) $\times$ Recall$(f_i, d, a)$]
6   return System-Benefit
```

The algorithm starts by sorting all combinations of detectors and their associated attack vertices according to their benefit to the overall system (line 2). The system benefit is calculated by the BENEFIT function. This specific design considers only the end nodes in the BN, corresponding to the ultimate attack goals. Other nodes that are of value to the system owner may also be considered. Note that a greedy decision is made in the BENEFIT calculation each detector is considered singly. From the sorted list, (detector, attack vertex) combinations are added in order, till the overall system cost due to detection is exceeded (line 7). Note that we use a cost-benefit table (line 2 of BENEFIT function), which is likely specified for each attack vertex at the finest level of granularity. One may also specify it for each host or each subnet in the system.

The worst-case complexity of this algorithm is $O(dv\ B(v, CPT(v)) + dv \log(dv) + dv)$, where d is the number of detectors and v is the number of attack vertices. $B(v, CPT(v))$ is the cost of Bayesian inference on a BN with v nodes and $CPT(v)$ defining the edges. The first term is due to calling Bayesian inference with up to d times v terms. The second term is the sorting cost and the third term is the cost of going through the for loop dv times. In practice, each detector will be applicable to only a constant number of attack vertices and therefore the

dv terms can be replaced by a constant times d, which will be only d considering order statistics.

The reader would have observed that the presented algorithm is greedy-choice of detectors is done according to a pre-computed order, in a linear sweep through the sorted list L (the for loop starting in line 4). This is not guaranteed to provide an optimal solution. For example, detectors d_2 and d_3 taken together may provide greater benefit even though detector d1 being ranked higher would have been considered first in the DETECTOR-PLACEMENT algorithm. This is due to the observation that the problem of optimal detector choice and placement can be mapped to the 0-1 knapsack problem which is known to be NP-hard. The mapping is obvious, consider $D \times A$ (D: Detectors and A: Attack vertices). We have to include as many of these tuples so as to maximize the benefit without the cost exceeding, the system cost of detection.

4 Experimental Systems

We created three Bayesian networks for our experiments modeling two real systems and one synthetic network. These are a distributed electronic commerce (e-commerce) system, a Voice-over-IP (VoIP) network, and a synthetic generic Bayesian network that is larger than the other two. The Bayesian networks were manually created from attack graphs that include several multi-step attacks for the vulnerabilities found in the software used for each system. These vulnerabilities are associated with specific versions of the particular software, and are taken from popular databases [6], [23]. An explanation for each Bayesian network follows.

4.1 E-Commerce System

The distributed e-commerce system used to build the first Bayesian network is a three tier architecture connected to the Internet and composed of an Apache web server, the Tomcat application server, and the MySQL database backend. All servers are running a Unix-based operating system. The web server sits in a demilitarized zone (DMZ) separated by a firewall from the other two servers, which are connected to a network not accessible from the Internet. All connections from the Internet and through servers are controlled by the firewall. Rules state that the web and application servers can communicate, as well as the web server can be reached from the Internet. The attack scenarios are designed with the assumption that the attacker is an external one and thus her starting point is the Internet. The goal for the attacker is to have access to the MySQL database (specifically access customer confidential data such as credit card information node 19 in the Bayesian network of Figure 4).

As an example, an attack step would be a portscan on the application server (node 10). This node has a child node, which represents a buffer overflow vulnerability present in the rpc.statd service running on the application server (node 12). The other attack steps in the network follow a similar logic and represent other phases of an attack to the distributed system. The system includes four

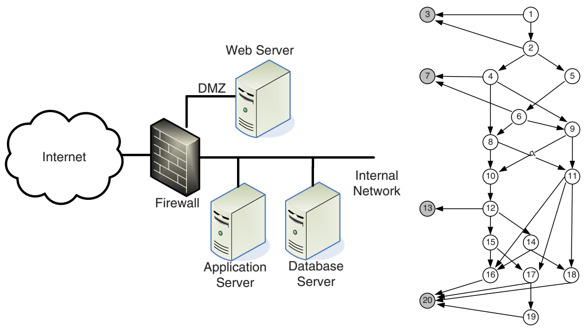


Fig. 4. Network diagram for the e-commerce system and its corresponding Bayesian network. The white nodes are the attack steps and the gray nodes are the detectors.

detectors: IPtables, Snort, Libsafe, and a database IDS. As shown in Figure 4, each detector has a causal relationship to at least one attack step.

4.2 Voice-over-IP (VoIP) System

The VoIP system used to build the second network has a few more components, making the resulting Bayesian network more complex. The system is divided into three zones: a DMZ for the servers accessible from the Internet, an internal network for local resources such as desktop computers, mail server and DNS server, and an internal network only for VoIP components. This separation of the internal network into two units follows the security guidelines for deploying a secure VoIP system [18].

The VoIP network includes a PBX/Proxy, voicemail server and software-based and hardware-based phones. A firewall provides all the rules to control the traffic between zones. The DNS and mail servers in the DMZ are the only accessible hosts from the Internet. The PBX server can route calls to the Internet or to a public-switched telephone network (PSTN). The ultimate goal of this multi-stage

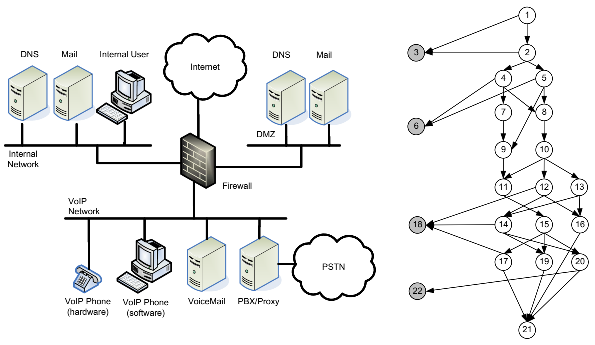


Fig. 5. VoIP system and its corresponding Bayesian network

	Attack = True	*Attack = False*
Detection = True	*TP*	*FP*
Detection = False	*FN*	*TN*

$$Recall = \frac{TP}{TP + FN} \qquad Precision = \frac{TP}{TP + FP}$$

Fig. 6. Parameters used for our experiments: True Positive (TP), False Positive (FP), True Negative (TN), False Negative (FN), precision, and recall

attack is to eavesdrop on VoIP communication. There are 4 detectors Iptables, and three network IDSs on the different subnets.

A third synthetic Bayesian network was built to test our framework for experiments where a larger network, than the other two, was required. This network is shown in Figure 7(a).

5 Experiments

The correct number, accuracy, and location of the detectors can provide an advantage to the systems owner when deploying an intrusion detection system. Several metrics have been developed for evaluation of intrusion detection systems. In our work, we concentrate on the precision and recall. Precision is the fraction of true positives determined among all attacks flagged by the detection system. Recall is the fraction of true positives determined among all real positives in the system. The notions of true positive, false positive, etc. are shown in Figure 6. We also plot the ROC curve which is a traditional method for characterizing detector performanceit is a plot of the true positive against the false positive.

For the experiments we create a dataset of 50,000 samples or attacks, based on the respective Bayesian network. We use the Matlab Bayesian network toolbox [3] for our Bayesian inference and sample generation. Each sample consists of a set of binary values, for each attack vertex and each detector vertex. A one (zero) value for an attack vertex indicates that attack step was achieved (not achieved) and a one (zero) value for a detector vertex indicates the detector generated (did not generate) an alert. Separately, we perform inference on the Bayesian network to determine the conditional probability of different attack vertices. The probability is then converted to a binary determination whether the detection system flagged that particular attack step or not, using a threshold. This determination is then compared with reality, as given by the attack samples which leads to a determination of the systems accuracy. There are several experimental parameters which specific attack vertex is to be considered, the threshold, CPT values, etc. and their values (or variations) are mentioned in the appropriate experiment. The CPTs of each node in the network are manually configured according to the authors experience administering security for distributed systems and frequency of occurrences of attacks from references such as vulnerability databases, as mentioned earlier.

5.1 Experiment 1: Distance from Detectors

The objective of experiment 1 was to quantify for a system designer what is the gain in placing a detector close to a service where a security event may occur. Here we used the synthetic network since it provided a larger range of distances between attack steps and detector alerts.

The CPTs were fixed to manually determined values on each attack step. Detectors were used as evidence, one at a time, on the Bayesian network and the respective conditional probability for each attack node was determined. The effect of the single detector on different attack vertices was studied, thereby varying the distance between the node and the detector. The output metric is the difference of two terms. The first term is the conditional probability that the attack step is achieved, conditioned on a specific detector firing. The second term is the probability that the attack step is achieved, without use of any detector evidence. The larger the difference is, the greater is the value of the information provided by the detector. In Figure 7(b), we show the effect due to detector corresponding to node 24 and in Figure 7(c), we consider all the detectors (again one at a time). The effect of all the detectors shows that the conclusions from node 24 are general.

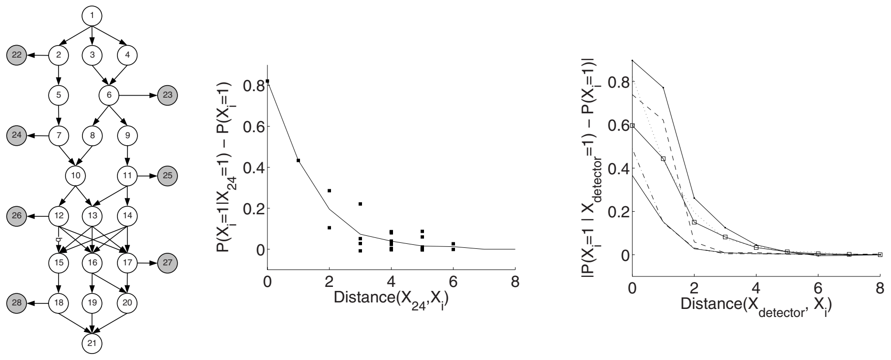


Fig. 7. Results of experiment 1: Impact of distance to a set of attack steps. (a) Generic Bayesian network used. (b) Using node 24 as the detector (evidence), the line shows mean values for rate of change. (c) Comparison between different detectors as evidence, showing the mean rate of change for case.

The results show that a detector can affect nodes inside a radius of up to three edges from the detector. The change in probability for a node within this radius, compared to one outside the radius, can be two times greater when the detector is used as evidence. For all Bayesian networks tested, the results were consistent to the three edges radius observation.

5.2 Experiment 2: Impact of Imperfect Knowledge

The objective of experiment 2 was to determine the performance of the detection system in the face of attacks. In the first part of the experiment *(Exp 2a)*, the

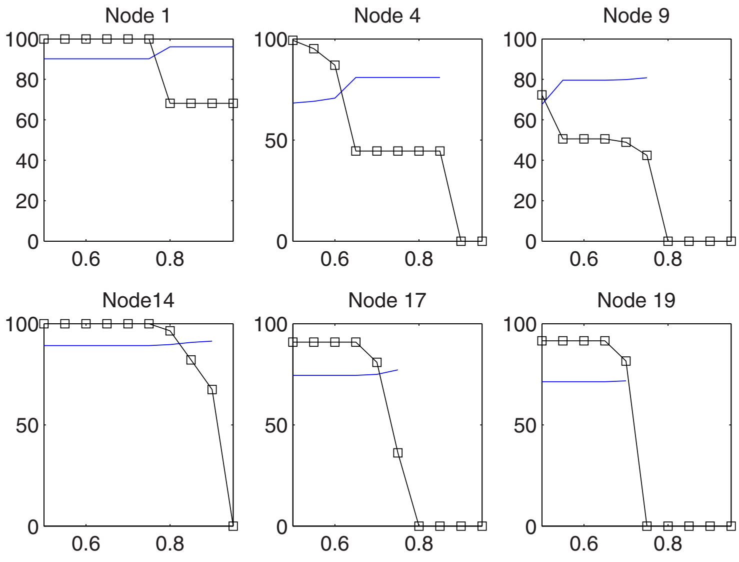


Fig. 8. Precision and recall as a function of detection threshold, for the e-commerce Bayesian network. The line with square markers is recall and other line is for precision.

effect of the threshold, that is used in converting the conditional probability of an attack step into a binary determination, is studied. This corresponds to the practical situation that a system administrator has to make a binary decision based on the result of a probabilistic framework and there is no oracle at hand to help. For the second part of the experiment *(Exp 2b)*, the CPT values in the Bayesian network are perturbed by introducing variances of different magnitudes. This corresponds to the practical situation that the system administrator cannot accurately gauge the level of difficulty for the adversary to achieve attack goals. The impact of the imperfect knowledge is studied through a ROC curve.

For Exp 2a, precision and recall were plotted as a function of the threshold value. This was done for all the attack nodes in the Bayesian network and the results for a representative sample of six nodes are shown in Figure 8. We used threshold values from 0.5 to 0.95, since anything below 0.5 would imply the Bayesian network is useless in its predictive ability.

Expectedly, as the threshold is increased, there are fewer false positives and the precision of the detection system improves. The opposite is true for the recall of the system since there are more false negatives. However, an illuminating observation is that the precision is relatively insensitive to the threshold variation while the recall has a sharp cutoff. Clearly, the desired threshold is to the left of the cutoff point. Therefore, this provides a scientific basis for an administrator to set the threshold for drawing conclusions from a Bayesian network representing the system.

In experiment 2b we introduced variance to the CPT values of all the attack nodes, mimicking different levels of imperfect knowledge an admin may

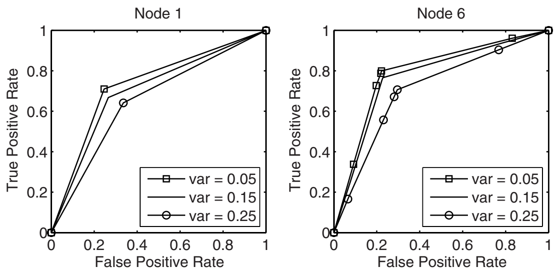


Fig. 9. ROC curves for two attack steps in e-commerce Bayesian network. Each curve corresponds to a different variance added to the CTP values.

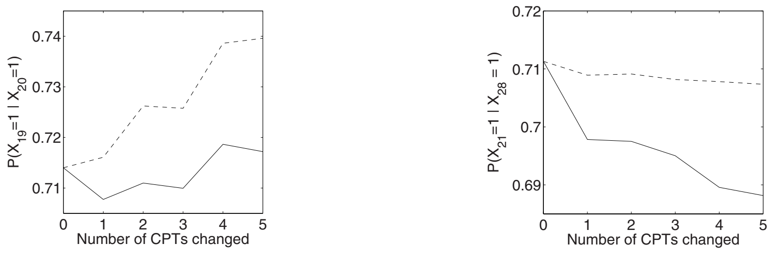


Fig. 10. Impact of deviation from correct CPT values, for the (a) e-commerce and (b) generic Bayesian networks

have about the adversary's attack strategies. When generating the samples corresponding to the attacks, we used three variance values: 0.05, 0.15, and 0.25. Each value could be associated with a different level of knowledge from an administrator: expert, intermediate, and nave, respectively. For each variance value, ten batches of 1,000 samples were generated and the detection results were averaged over all batches.

In Figure 9, we show the ROC curves for nodes 1 and 6 of the e-commerce system, with all four detectors in place. Expectedly, as the variance increases, the performance suffers. However, the process of Bayesian inference shows an inherent resilience since the performance does not degrade significantly with the increase in variance. For node 1, several points are placed so close together that only one marker shows up. On the contrary, for node 6, multiple well spread out TP-FP value pairs are observed. We hypothesize that since node 1 is directly connected to the detector node 3, its influence over node 1 dominates that of all other detectors. Hence fewer number of sharp transitions are seen compared to node 6, which is more centrally placed with respect to multiple detectors.

Experiment 2c also looked at the impact of imperfect knowledge when defining the CPT values in the Bayesian network. Here we progressively changed the CPT values for several attack steps in order to determine how much we would deviate from the correct value. We used two values 0.6 and 0.8 for each CPT cell (only two are independent) giving rise to four possible CPT tables for each node. We plot the minimum and maximum conditional probabilities for a representative attack node for a given detector flagging. We change the number of CPTs that we perturb from the ideal values. Expectedly as the number of CPTs changed increases, the difference between the minimum and the maximum increases, but the range is within 0.03. Note that the point at the left end of the curve for zero CPTs changed gives the correct value.

Both experiments indicate that the BN formalism is relatively robust to imperfect assumptions concerning the CPT values. This is an important fact since it is likely that the values determined by an experienced system administrator would still be somewhat imperfect. Overall, as long as the deviation of the assumed CPTs from the truth is not overwhelming, the network performance degrades gracefully.

5.3 Experiment 3: Impact on Choice and Placement of Detectors

The objective of experiment 3 was to determine the impact of selecting the detectors and their corresponding locations. To achieve this, we ran experiments on the e-commerce and the VoIP Bayesian networks to determine a pair of detectors that would be most effective. This pair, called the optimal pair, is chosen according to the algorithm described in Section 3.2. The performance of the optimal pair is compared against additional pairs selected at random. We show the result using the ROC curve for the two ultimate attack goals, namely node 19 and node 21 in the e-commerce and the VoIP systems.

To calculate the performance of each pair of detectors, we created 10,000 samples from each Bayesian network, corresponding to that many actual attacks. Then we performed Bayesian inference and calculated the conditional probability of the attack step, given the pair of detectors. We determined the true positive rate and false positive rate by sweeping across threshold values.

Results show that the pair of detectors determined from the algorithm performs better than the other randomly selected pairs. Figure 11a shows the situation in which a single detector (d_{20}) attached to two attack nodes (x_{19}, x_{18}) performs better than two detectors (d_{13} and d_7, or d_{12} and d_3). The placement of the detector d_{20} affects the performance. This can be explained by the fact that node 18 is more

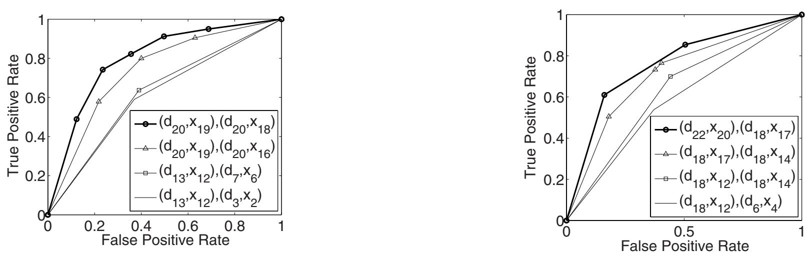


Fig. 11. ROC curves for detection of attack steps, using pairs of detectors, in the e-commerce network (left) and the VoIP network (right)

highly connected in the attack graph and therefore attaching detector d_{20} to that node, rather than node 16, provides better predictive performance.

There is a cost of adding detectors to a system, but there is also a cost of having a detector attached to more attack nodes, in terms of the bandwidth and computation. Thus adding further edges in the Bayesian network between a detector node and an attack node, even if feasible, may not be desirable. For the VoIP network, detector pair d_{22} and d_{18} performs best. This time two separate detectors outperform a single high quality detector (d_{18}) connected to two nodes.

Further details on all experiments performed, including all the probability values used for the Bayesian networks, are available at [22]. These are omitted here due to space constraints and the interested party is welcome to further read. All the experiments validate the intuition behind our algorithm that the greedy choice of the detectors also gives good results when multiple detectors are considered together and over the entire Bayesian network.

6 Related Work

Bayesian networks have been used in intrusion detection to perform classification of events. Kruegel et al. [17] proposed the usage of Bayesian networks to reduce the number of false alarms. Bayesian networks are used to improve the aggregation of different model outputs and allow integration of additional information. The experimental results show an improvement in the accuracy of detections, compared to threshold-based schemes. Ben Amor et al. [4] studied the use of nave Bayes in intrusion detection, which included a performance comparison with decision trees. Due to similar performance and simpler structure, nave Bayes is an attractive alternative for intrusion detection. Other researchers have also used nave Bayesian inference for classifying intrusion events [29].

To the best of our knowledge, the problem of determining an appropriate location for detectors has not been systematically explored by the intrusion detection community. However, analogous problems have been studied to some extent in the physical security and the sensor network fields.

Jones *et al.* [15] developed a Markov Decision Process (MDP) model of how an intruder might try to penetrate the various barriers designed to protect a physical facility. The model output includes the probability of a successful intrusion and the most likely paths for success. These paths provide a basis to determine the location of new barriers to deter a future intrusion.

In the case of sensor networks, the placement problem has been studied to identify multiple phenomena such as determining location of an intrusion [1], contamination source [5], [27], and atmospheric conditions [16]. Anjum *et al.* [1] determined which nodes should act as intrusion detectors in order to provide detection capabilities in a hierarchical sensor network. The adversary is trying to send malicious traffic to a destination node (say, the base node). In their model, only some nodes called tamper-resistant nodes are capable of executing a signature-based intrusion detection algorithm and these nodes cannot be compromised by an adversary. Since these nodes are expensive, the goal is to minimize the number of such nodes and the authors provide a distributed approximate algorithm for this based on minimum cut-set and minimum dominating set. The solution is applicable to a specific kind of topology, widely used in sensor networks, namely clusters with a cluster head in each cluster capable of communicating with the nodes at the higher layer of the network hierarchy.

In [5], the sensor placement problem is studied to detect the contamination of air or water supplies from a single source. The goal is to detect that contamination has happened and the source of the contamination, under the constraints that the number of sensors and the time for detection are limited. The authors show that the problem with sensor constraint or time constraint are both NP-hard and they come up with approximation algorithms. They also solve the problem exactly for two specific cases, the uniform clique and rooted trees. A significant contribution of this work is the time efficient method of calculating the sensor placement. However, several simplifying assumptions are made—sensing is perfect and no sensor failure (either natural or malicious) occurs, there is a single contaminating source, and the flow is stable.

Krause *et al.* [16] also point out the intractability of the placement problem and present a polynomial-time algorithm to provide near-optimal placement which incurs low communication cost between the sensors. The approximation algorithm exploits two properties of this problem: submodularity, formalizing the intuition that adding a node to a small deployment can help more than adding a node to a large deployment; and locality, under which nodes that are far from each other provide almost independent information. In our current work, we also experienced the locality property of the placement problem. The proposed solution *learns* a probabilistic model (based on Gaussian processes) of the underlying phenomenon (variation of temperature, light, and precipitation) and for the expected communication cost between any two locations from a small, short-term initial deployment.

In [27], the authors present an approach for determining the location in an indoor environment based on which sensors cover the location. The key idea is to ensure that each resolvable position is covered by a unique set of sensors, which then serves as its signature. They make use of identifying code theory to reduce the number of active sensors required by the system and yet provide unique localization for each position. The algorithm also considers robustness, in terms of the number of sensor failures that can be corrected, and provides solutions in harsh environments, such as presence of noise and changes in the structural topology. The objective for deploying sensors here is quite different from our current work.

For all the previous work on placement of detectors, the authors are looking to detect events of interest, which propagate using some well-defined models, such as, through the cluster head *en route* to a base node. Some of the work (such as [16]) is focused on detecting natural events, that do not have a malicious motive in avoiding detection. In our case, we deal with malicious adversaries who have an active goal of trying to bypass the security of the system. The adversaries' methods of attacking the system do not follow a well-known model making our problem challenging. As an example of how our solution handles this, we use noise in our BN model to emulate the lack of an accurate attack model.

There are some similarities between the work done in alert correlation and ours, primarily the interest to reduce the number of alerts to be analyzed from an intrusion. Approaches such as [24] have proposed modeling attack scenarios to correlate alerts and identify causal relationships among the alerts. Our work aims to closely integrate the vulnerability analysis into the placement process, whereas the alert correlation proposals have not suggested such importance.

The idea of using Bayes theorem for detector placement is suggested in [26]. No formal definition is given, but several metrics such as accuracy, sensitivity, and specificity are presented to help an administrator make informed choices about placing detectors in a distributed system. These metrics are associated to different areas or sub-networks of the system to help in the decision process.

Many studies have been done on developing performance metrics for the evaluation of intrusion detection systems (IDS), which have influenced our choice of metrics here. Axelsson [2] showed the applicability of estimation theory in the intrusion detection field and presented the Bayesian detection rate as a metric for the performance of an IDS. His observation that the base rate, and not only the false alarm rate, is an important factor on the Bayesian detection rate, was included in our work by using low base rates as part of probability values in the Bayesian network. The MAFTIA Project [8] proposed precision and recall to effectively determine when a vulnerability was exploited in the system. A difference from our approach is that they expand the metrics to consider a set of IDSes and not only a single detector. The idea of using ROC curves to measure performance of intrusion detectors has been explored many times, most recently in [7], [10].

Extensive work has been done for many years with attack graphs. Recent work has concentrated on the problems of generating attack graphs for large networks and automating the process to describe and analyze vulnerabilities and system components to create the graphs. The NetSPA system [11] uses a breath-first technique to generate a graph that grows almost linearly with the size of the distributed system. Ou et al. [25] proposed a graph building algorithm using a formal logical technique that allows to create graphs of polynomial size to the network being analyzed.

7 Conclusions and Future Work

Bayesian networks have proven to be a useful tool in representing complex probability distributions, such as in our case of determining the likelihood that an attack goal has been achieved, given evidence from a set of detectors. By using attack graphs and Bayesian inference, we can quantify the overall detection performance in the systems by looking at different choices and placements of detectors and the detection parameter settings. We also quantified the information gain due to a detector as a function of its distance from the attack step. Also, the effectiveness of the Bayesian networks can be affected by imperfect knowledge when defining the conditional probability values. Nevertheless, the Bayesian network exhibits considerable resiliency to these factors as our experiments showed.

Future work should include looking at the scalability issues of Bayesian networks and its impact on determining the location for a set of detectors in a distributed system. The probability values acquisition problem can be handled by using techniques such as the recursive noisy-OR modeling [19] but experimentation is required to determine its benefits and limitations for our scenario.

Acknowledgments. Gaspar Modelo-Howard was partly supported by an IFARHU-SENACYT Scholarship from the Republic of Panama. Saurabh Bagchi was partly supported in this work by an endowment grant from Purdue's Center for Education and Research in Information Assurance and Security (CERIAS).

References

1. Anjum, F., Subhadrabandhu, D., Sarkar, S., Shetty, R.: On Optimal Placement of Intrusion Detection Modules in Sensor Networks. In: 1st IEEE International Conference on Broadband Networks, pp. 690–699. IEEE Press, New York (2004)
2. Axelsson, S.: The base-rate fallacy and the difficulty of intrusion detection. ACM Trans. Inf. Syst. Secur. 3-3, 186–205 (2000)
3. Bayes Net Toolbox for Matlab, `http://www.cs.ubc.ca/~murphyk/Software`
4. Ben Amor, N., Benferhat, S., Elouedi, Z.: Naive Bayes vs decision trees in intrusion detection systems. In: 19th ACM Symposium on Applied computing, pp. 420–424. ACM Press, New York (2004)
5. Berger-Wolf, T., Hart, W., Saia, J.: Discrete Sensor Placement Problems in Distribution Networks. J. Math. and Comp. Model. 42, 1385–1396 (2005)
6. Bugtraq Vulnerability Database, `http://www.securityfocus.com/vulnerabilities`
7. Cardenas, A., Baras, J., Seamon, K.: A Framework for the Evaluation of Intrusion Detection Systems. In: 27th IEEE Symposium on Security and Privacy, p. 15. IEEE Press, New York (2006)
8. Dacier, M. (ed.): Design of an Intrusion-Tolerant Intrusion Detection System. Research Report, Maftia Project (2002)
9. Foo, B., Wu, Y., Mao, Y., Bagchi, S., Spafford, E.: ADEPTS: Adaptive Intrusion Response using Attack Graphs in an E-Commerce Environment. In: International Conference on Dependable Systems and Networks, pp. 508–517 (2005)
10. Gu, G., Fogla, P., Dagon, D., Lee, W., Skoric, B.: Measuring Intrusion Detection Capability: An Information-Theoretic Approach. In: 1st ACM Symposium on Information, Computer and Communications Security, pp. 90–101. ACM Press, New York (2006)
11. Ingols, K., Lippmann, R., Piwowarski, K.: Practical Attack Graph Generation for Network Defense. In: 22nd Annual Computer Security Applications Conference, pp. 121–130. IEEE Press, New York (2006)
12. IPTables Firewall, `http://www.netfilters.org/projects/iptables`
13. Jensen, F.: Bayesian Networks and Decision Graphs. Springer, Heidelberg (2001)
14. Jha, S., Sheyner, O., Wing, J.: Two Formal Analyses of Attack Graphs. In: 15th IEEE Computer Security Foundations Workshop, pp. 49–63. IEEE Press, New York (2002)

15. Jones, D., Davis, C., Turnquist, M., Nozick, L.: Physical Security and Vulnerability Modeling for Infrastructure Facilities. Technical Report, Sandia National Laboratories (2006)
16. Krause, A., Guestrin, C., Gupta, A., Kleinberg, J.: Near-optimal Sensor Placements: Maximizing Information while Minimizing Communication Cost. In: 5th International Conference on Information Processing in Sensor Networks, pp. 2–10. ACM Press, New York (2006)
17. Krügel, C., Mutz, D., Robertson, W., Valeyr, F.: Bayesian Event Classification for Intrusion Detection. In: 19th Annual Computer Security Applications Conference, pp. 14–23. IEEE Press, New York (2003)
18. Kuhn, D., Walsh, T., Fires, S.: Security Considerations for Voice Over IP Systems. Special Publication 800-58, National Institute of Standards and Technology (2005)
19. Lemmer, J., Gossink, D.: Recursive Noisy OR - A Rule for Estimating Complex Probabilistic Interactions. IEEE Trans. Syst. Man. Cybern. B. 34, 2252–2261 (2004)
20. Lippmann, R., et al.: Evaluating Intrusion Detection Systems: The 1998 DARPA Off-line Intrusion Detection Evaluation. In: 1st DARPA Information Survivability Conference and Exposition, pp. 81–89 (2000)
21. Mehta, V., Bartzis, C., Zhu, H., Clarke, E., Wing, J.: Ranking Attack Graphs. In: Zamboni, D., Krügel, C. (eds.) RAID 2006. LNCS, vol. 4219, pp. 127–144. Springer, Heidelberg (2006)
22. Modelo-Howard, G.: Addendum to Determining Placement of Intrusion Detectors for a Distributed Application through Bayesian Network Modeling, http://cobweb.ecn.purdue.edu/ dcsl/publications/ detectors-location_addendum.pdf
23. National Vulnerability Database, http://nvd.nist.gov/nvd.cfm
24. Ning, P., Cui, Y., Reeves, D.: Constructing Attack Scenarios through Correlation of Intrusion Alerts. In: 9th ACM Conference on Computers & Communications Security, pp. 245–254 (2002)
25. Ou, X., Boyer, W., McQueen, M.: A Scalable Approach to Attack Graph Generation. In: 13th ACM Conference on Computer & Communications Security, pp. 336–345 (2006)
26. Peikari, C., Chuvakin, A.: Security Warrior. O'Reilly, New York (2004)
27. Ray, S., Starobinski, D., Trachtenberg, A., Ungrangsi, R.: Robust Location Detection with Sensor Networks. IEEE J. on Selected Areas in Comm. 22, 1016–1025 (2004)
28. Snort Intrusion Detection System, http://www.snort.org
29. Valdes, A., Skinner, K.: Adaptive, Model-based Monitoring for Cyber Attack Detection. In: Debar, H., Mé, L., Wu, S.F. (eds.) RAID 2000. LNCS, vol. 1907, pp. 80–92. Springer, Heidelberg (2000)

A Multi-Sensor Model to Improve Automated Attack Detection

Magnus Almgren[1], Ulf Lindqvist[2], and Erland Jonsson[1]

[1] Department of Computer Science and Engineering
Chalmers University of Technology
SE-412 96 Göteborg, Sweden
[2] Computer Science Laboratory
SRI International
333 Ravenswood Ave
Menlo Park, CA 94025, USA

Abstract. Most intrusion detection systems available today are using a single audit source for detection, even though attacks have distinct manifestations in different parts of the system. In this paper we investigate how to use the alerts from several audit sources to improve the accuracy of the intrusion detection system (IDS). Concentrating on web server attacks, we design a theoretical model to automatically reason about alerts from different sensors, thereby also giving security operators a better understanding of possible attacks against their systems. Our model takes sensor status and capability into account, and therefore enables reasoning about the absence of expected alerts. We require an explicit model for each sensor in the system, which allows us to reason about the quality of information from each particular sensor and to resolve apparent contradictions in a set of alerts.

Our model, which is built using Bayesian networks, needs some initial parameter values that can be provided by the IDS operator. We apply this model in two different scenarios for web server security. The scenarios show the importance of having a model that dynamically can adapt to local transitional traffic conditions, such as encrypted requests, when using conflicting evidence from sensors to reason about attacks.

Keywords: intrusion detection, alert reasoning.

1 Introduction

The accuracy of an intrusion detection system (IDS), meaning the degree to which the security officer can trust the IDS to recognize attacks and at the same time not produce false alarms, can be considered the most important property of an IDS (a general definition of detector accuracy can be found in [19]). However, many IDSs do not provide a very high degree of accuracy, because of two common shortcomings. First, the IDS tends to rely on a single detection method applied to a single audit source such as network packets, which is not sufficient to accurately recognize all types of attacks. Second, many IDSs have a propensity for producing massive amounts of alerts, many of which are irrelevant or false. Over time, the security officer monitoring the alerts from the IDS could learn which types of alerts can be safely ignored, and even which combinations of alerts indicate a more serious incident.

A significant amount of research has been conducted to improve the accuracy of IDSs. Some of that work has been focused on the implementation of detectors, such as speed improvements or techniques to detect obfuscated attacks. Other work has been focused on alert processing techniques in the form of alert aggregation and correlation, such as root-cause analysis. Diversity has been proposed as a principle to be adopted to improve detection coverage. One form of diversity is to use a combination of detection techniques, for example, signature-based detection combined with statistical anomaly detection. Another form of diversity is to simultaneously use input from different audit streams, for example, network packets and application event logs.

While diversity is a promising approach to increasing IDS accuracy, complementary sensors are currently not widely developed or deployed. This happens because without automated procedures to take advantage of multiple diverse alert sources and make the correct inferences, the burden on the security officer will increase rather than decrease, especially in the cases of conflicting sensor reports. This paper proposes and investigates a model upon which such automated reasoning can be based, ultimately presenting the security officer with actionable and highly accurate information about ongoing attacks, and reducing the need for operator experience and expertise.

The model presented in this paper is applied to the output of traditional correlators—we assume that these techniques preprocess the alert stream and present us with an aggregated set of related alerts concerning an attack (see Section 7 for a discussion of correlation techniques). We propose a model to combine the alerts from several intrusion detection systems using different audit sources. We show how our model can resolve seemingly conflicting evidence about possible attacks as well as properly account for transient failure modes of the sensors in the system, such as the network IDS being blind when analyzing encrypted traffic.

Our approach would benefit from sensors having an intrinsic knowledge of their own detection capability as well as having modes to specifically monitor the outcomes of certain attacks. However, our approach also works with traditional sensors. By using correlated alerts as input, we benefit from previous research into correlation. As we present them with further refined information, the security officers can spend their time actively protecting the system instead of trying to understand the mixed message from intrusion detection sensors.

The rest of the paper is organized as follows. In Section 2 we describe the notation used in the paper and outline the problem we are investigating. We then formally describe the assumptions and requirements we need before we introduce our decision framework in Section 3. We use two scenarios to exemplify the model, and these are presented in Section 4. The test bed and the experiments are described in Section 5. We summarize our findings in Section 6 and discuss related work in Section 7. The paper is concluded in Section 9.

2 Theory

2.1 Notation

We use the term *sensor* to denote a component that monitors an event stream (audit source) for indications of suspicious activity according to a detection algorithm and produces alerts as a result. A simple IDS, such as a typical deployment of the popular tool Snort, in most cases constitutes a single sensor. We therefore use the terms *sensor* and *IDS* interchangeably in this paper. More advanced IDS deployments could be composed of several sensors that feed into a common alerting framework.

Let us assume that we use a set of intrusion detection sensors, S, to detect attacks, and in particular the attack, $\mathcal{A}$. A sensor S_i may alert for ongoing attacks but sometimes gives false alerts. We denote such alerts by ${}_{\mathcal{A}}a^i_j$, where each sensor may give several alerts for each attack (j-index). If S_i is present in the system and alerts for the attack $\mathcal{A}$, we denote this by $S_i : {}_{\mathcal{A}}a^i_j$. For simplicity, we are going to concentrate on a single attack in the discussion below, so the index $\mathcal{A}$ is not shown explicitly. When the attack $\mathcal{A}$ does not trigger any alert in the sensor S_i, we denote this by $S_i : \neg_{\mathcal{A}}a^i_j$ or simpler as $S_i : \neg a^i_j$. To simplify the discussion, we show only the missing alerts that are actually relevant for the current situation. Finally, a sensor may temporarily be missing in a system or a specific sensor may not work as intended. Following the same notation as for alerts, we denote such a malfunctioning or missing sensor with $\neg S_i$. Observing this state directly is very difficult in current intrusion detection systems, and often we can only indirectly assume that a sensor is not working correctly. For example, a heartbeat message may tell us that a network IDS is still running, but if the traffic is encrypted, the sensor cannot properly analyze it.

Following this notation, we have the four separate cases shown in Table 1. Each alert for these cases may be true or false, but if the sensor is not working (iii and iv) we treat all alerts as being false, as they would only coincidentally be true. We consider case (iv) to be uncommon with more sophisticated sensors, but if it happens the security officer will investigate the (false) alert and discover the malfunctioning sensor. Finally, as the sensor status cannot directly be observed, cases (i) and (iv) would look similar to a security operator without further investigation, as would cases (ii) and (iii).

2.2 Example with Two Sensors

Now consider a particular system with two sensors S_1 and S_2, where each sensor can output a single alert for the $\mathcal{A}$-attack (we drop the j-index in this example).

Table 1. The four possible sensor / alert states

(i)	S_i: a_j^i	S_i is working correctly and outputs alert a_j^i.
(ii)	S_i:$\neg a_j^i$	S_i is working correctly and has not found any signs that warrant the output of alert a_j^i.
(iii)	$\neg S_i$:$\neg a_j^i$	S_i is not working correctly and does not output alert a_j^i regardless of the attack status.
(iv)	$\neg S_i$: a_j^i	S_i is not working correctly but still outputs an alert regardless of the attack status, for example, when traffic is encrypted but happens to contain a pattern that triggers the alert a_j^i.

Table 2. An example using two sensors with one alert each

$\neg a^1$	$\neg a^2$	(2a)
a^1	a^2	(2b)
$\neg a^1$	a^2	(2c)
a^1	$\neg a^2$	(2d)

As hinted earlier, the sensor status is seldom directly measured, so with such a setup only the four cases shown in Table 2 can be directly observed.

The interpretations of (2a) (i.e., the first case in Table 2) and (2b) are straightforward. In the first case, no sensor reports anything so we do not investigate further. In the second case, both sensors warn for an attack and for that reason it is worth investigating further. However, cases (2c) and (2d) are interesting, as only one of the two sensors reports the possible ongoing attack. How should these two cases be interpreted?

Clearly, we need more information to draw any conclusions for cases (2c) and (2d). The burden of collecting this extra information, and using it to reason about the attack status for these cases, has often fallen on the security operator. In our opinion, many correlator techniques have so far taken a simplified but safe view: if any sensor reports an attack, it should be treated as an attack, i.e., the security operator needs to investigate this issue manually. The problem we are investigating is whether we can improve the automatic analysis and provide a more refined answer to the security operator. In the remainder of this section, we describe some reasons for why traditional correlation technologies may have used the simplified view described above. These reasons are used as a basis for the discussion in Section 3, where we set up the necessary framework to provide a model that aids the security operator by solving cases with seemingly conflicting evidence.

2.3 The Problem of Conflicting Evidence

In Section 2.2, we showed the possible outputs of two sensors, which in certain cases may be conflicting. Here, we discuss how to interpret those cases of conflicting sensor output. This discussion serves as a background to understand the requirements of our model that we introduce in Section 3.

No Complementary Sensors Deployed. First we note that in many typical environments, cases (2c) and (2d) described above may not be very common, because the same type of sensor is duplicated across the network to allow for some partial redundancy in the monitoring capacity. For example, a company may have two instances of Snort analyzing almost the same traffic. In this case, the only reason the identical sensors would disagree is if one is broken. Thus, the only recourse is to interpret both (2c) and (2d) as a sign of an attack. We would like to point out that even though it may be common to have only one type of sensor, research shows the benefits of using several different sensors for attack detection ([3]).

Ambiguity between 'No Attack' and a Broken Sensor. Even when we use different types of sensors we are still faced with the problem of what a missing alert means. The sensor state is often unknown, and as we showed in Table 1, a sensor reporting no alert may signify one of two conditions: $S_i : \neg a^i$ or $\neg S_i : \neg a^i$. In the first case, the sensor is working as intended and does not detect any signs of an attack. In the second case, the sensor is broken, and for that reason it cannot reliably detect attacks. Not only is it difficult to determine the stationary state of a sensor through direct observation, but the conditions for when a sensor may detect attacks may also change dynamically; a network IDS is blind to the single encrypted web request, or a request that is routed around it, and so on. Not knowing the sensor state, the operator cannot confidently disregard an alert just because only one sensor produced it.

Detailed Sensor Alert Information Missing. Let us say that we do know that both sensors are working as intended but they report conflicting evidence as in (2c). Without any detailed information about the particulars of a sensor and its proclivity to produce false alerts, one cannot automatically decide which sensor to believe. One might not even know that a^1 is missing in (2c). Unless we have a sensor model, this decision must be left to the human operator.

Generality Has Been Prioritized. Many correlation techniques try to group alerts belonging to the same attack to make the security operator's task easier. As correlation techniques have been developed after most intrusion detection techniques, the developers of correlation techniques have not had much influence on the operation of IDS sensors; instead, the focus has been to work with *any* sensor.

3 Intrusion Detection Sensor Model

We first describe our assumptions and our requirements. With these clearly in mind, we then describe our proposed model and its advantages and disadvantages.

3.1 Assumptions and Requirements

As shown in the previous section, we need more information to be able to handle cases with conflicting information from different sensors. We make the following assumption in our work:

Assumption 1. *We assume that the absence of a particular alert may constitute evidence that an attack is not taking place.* At first glance this assumption may look strange, but this is already the case in any IDS deployment today; with no alert, one assumes that all is well and one does not follow up with an investigation (this is case (2a) in Section 2.2).

We can then formulate the following requirements.

Requirement 1. *We require a sensor model, which tells us whether an alert for a particular attack is in the set of possible alerts that this sensor can produce.* Such a model could possibly be created by automatic tools [15].

Requirement 2. *Furthermore, we require that this sensor model describe the sensor's accuracy with respect to detecting a particular attack.* Knowing the sensor's accuracy helps us resolve cases with conflicting evidence.

Requirement 3. *We require sensors to have some degree of functional independence.* Additional identical sensors analyzing the same event stream do not provide added value beyond redundancy. However, independence is a difficult requirement to satisfy and to verify. More work is needed to develop different types of sensors and to measure the functional independence between them.

Requirement 4. *We require knowledge of the sensor status to draw the correct conclusion.* There are two reasons why a sensor may not produce an alert; the sensor is either functioning normally and has concluded that an attack is not in progress, or the sensor is malfunctioning. Only in the former case should we consider a missing alert to be evidence that an attack is not occurring. The sensor model needs to describe under what conditions a sensor will not work (for example, when it encounters encrypted traffic).

Now let us return to case (2c) described in Section 2.2 with conflicting information: $\neg a^1$, a^2. Using *assumption 1* above (and knowing to look for the missing alert from *req. 1*), we can conclude that one of the sensors is not reliable in this case. There are two possible interpretations:

- *An attack is indeed in progress*
  - S_1 is not working correctly, and therefore did not produce an alert, or
  - the attack detection mechanism in S_1 does not cover all variants of this attack.
- *There is no attack in progress*
  - S_2 is not working correctly and produced this alert as a result of malfunctioning, or
  - the attack detection mechanism in S_2 falsely concluded that an attack was in progress based on its analysis of the audit source (a traditional false alarm).

Thus, we are faced with first deciding if all sensors are working in the system (*req. 4*). Clearly, if a sensor is malfunctioning in an easily discernible way, we

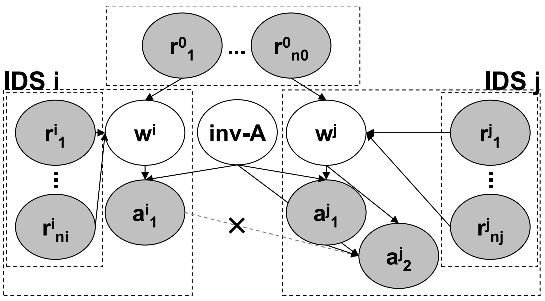


Fig. 1. A template of the Bayesian model we use. Each IDS is treated as somewhat independent of the others, but the complete details are given in Section 3.2.

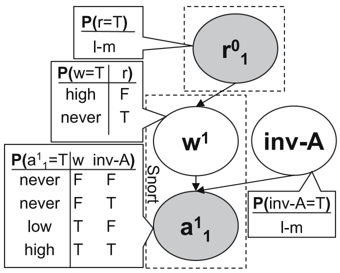


Fig. 2. An example of the model using Snort with the rule for detecting the *phf attack* (sid=1762). We use conditional probability tables with labels (where l-m stands for low-medium), which are later replaced with actual probabilities.

can differentiate between these situations. However, if all sensors seem to work correctly we need to weigh the evidence given by one sensor for this particular attack against the evidence given by another sensor (*req. 2*). Simply put, if it is known that S_2 is prone to false alarms for this particular attack, while S_1 is more accurate, we can ignore the combined alert for now. This analysis is possible only if the sensors are somewhat independent (*req. 3*). Thus, our model needs to account for the sensor status (and its known weaknesses) as well as the detection capability of the sensor (rule) for this particular attack.

3.2 Model Description

We use a Bayesian framework to model the sensors and their interdependence. Such a framework has several advantages, and among them is the intuitive mix between the graphical model with the underlying formal reliance on probability. The model is shown in Figure 1. As shown, the model consists of a series of nodes and the directional connections between these nodes. The model can graphically be represented as a DAG (directionally acyclic graph). The nodes represents variables, and the edges between nodes signify a dependence between these particular variables. There are efficient algorithms to calculate the posterior probability of a variable, given the observed evidence of other variables.

We use the model to find out whether an attack that should be further investigated (node **inv**estigate**-A**) is occurring, based on evidence in the form of alerts (nodes a^*_*) collected from a set of intrusion detection sensors. Based on several parameters and observations (nodes r^*_*), a sensor may accurately detect an attack or fail to do so, and this is accounted for in the model (nodes w^*). The value of each node may be observed in some circumstances (for example, a specific alert is triggered). In this particular application domain, some nodes will probably almost always be observed while some others will never be observed. In Figure 1, the observable nodes are shaded, while the white nodes are seldom observed directly.

To summarize, in the model we use four types of nodes:

Node inv-A is used to determine if the ongoing attack is serious enough to be further investigated. Obviously, the value of this node is never directly observed.

Nodes $\mathbf{a}_*^*$ signify whether we have received particular alerts, and thus serve as the basis to calculate the node *inv-A*.

Nodes $\mathbf{w}^*$ model the sensor status, as a missing alert may mean two different conditions: no attack or a broken sensor. These nodes cannot directly be observed.

Nodes $\mathbf{r}_*^*$ are used to calculate the sensor status (nodes w^*) in a fashion similar to how the nodes a_*^* are used to calculate *inv-A*. These nodes are often observed and populated with particular observations from the sensor environment.

The nodes are informally organized into groups based on which intrusion detection system they belong to. Keeping each IDS as isolated from others as possible leads to a simpler model and below we elaborate on this topic and describe the dotted edge with the x found in Figure 1.

Even though each IDS has its own particular failure modes, some observations are important to several IDSs. For that reason, there are both *global* (r^0) and *local* r-nodes. Furthermore, some of the r-nodes report transient observations while others may report more stationary conditions where a value is *sticky*, i.e., remains until explicitly changed. We describe the implementation in further detail in Section 5. Below we expand on the features of the model.

Parameter Estimation. As with any other probability model, one needs to estimate parameters for each node. This is difficult, but there are several reasons why we believe it is feasible for our model.

Using Independence Assumptions: the model takes advantage of the independence assumptions visible in the graphical structure and thus reduces the number of estimates that are necessary as compared with a full joint distribution without explicit independence assumptions.

Robust Parameter Estimation: furthermore, it is many times enough to capture the ratio between the parameters while their actual values are less important [5].

Local Parameters: the model parameters are expressed as something the security officer is already familiar with, e.g., false positives and false negatives for each rule.

We envision that most of these parameters have reasonable default values that can be estimated by the IDS vendor, and that the security officer then only needs to fine-tune the settings at the local site. It is possible that some of this fine-tuning can be performed by machine learning algorithms based on current traffic seen at the site.

Problematic Interdependence between IDSs. We would like to highlight the problem concerning independence assumptions. Clearly, the model in Figure 1 is simplified. Keeping a simple and modular structure introduces some incorrect independence assumptions. For example, let us assume that IDS i and IDS j are both signature-based IDSs. IDS i has one alert for $\mathcal{A}$ while IDS j has two alerts. In Figure 1, we show that the two alerts from IDS j are dependent, but that the alert from IDS i is independent of the others. In reality, it is likely that a_2^j is dependent on, for example, a_1^i as indicated in Figure 1 with the dotted line with the x. Even different commercial IDSs many times use similar signatures to detect attacks. As will be seen in the examples shown in Section 4, we sometimes ignore this particular dependence. The reasons are the following:

- First, a model may work very well despite some broken independence assumptions; consider for example the Naive Bayes model, which works surprisingly well despite being very inaccurate in terms of independence assumptions [11].
- Second, excluding the inter-IDS dependence simplifies the model. If we include these dependencies between IDSs, it would mean that the inclusion of a new IDS to the whole system would necessitate a re-evaluation of many parameters (and thus invalidate the opportunity to have pre-set default values).
- Third, estimating this dependence is difficult. Someone would have to be an expert on both IDSs to be able to set the level of dependence between two alerts.

For these reasons, we sometimes explicitly ignore the inter-IDS dependencies even though we acknowledge that they exist. Thus, we balance the simplicity of the model against its accuracy.

3.3 Model Example: Estimating the Parameters

In Figure 2 we show a simplified example of the model, where we have limited the number of nodes to make it more understandable. In this case, we concentrate on the *phf attack* [12]. The background and execution of the attack can be found in Almgren et al. [3]. By sending a request to run the vulnerable cgi script *phf* and including the newline character in hex encoding (%0a), the attacker may be able to execute arbitrary commands. The script passes control to the shell without removing the newline character, which has special meaning to the shell.

The open-source IDS Snort has several potential ways to detect this attack ([3]). For example, one can use *rule 1762*, which detects the string "/phf" matched with a newline character within the URI. Snort is a network-based IDS, and for that reason it cannot detect attacks in encrypted traffic (among other things).

Now let us consider how to estimate the necessary parameters for the model shown in Figure 2. The structure is given from Figure 1 and we have restricted each node to be either *true (T)* or *false (F)*. Even though one can give probability distributions over each node, we use *conditional probability tables* (CPTs) in this paper. A full joint probability distribution would need 16 parameters, but taking advantage of the independence shown in the figure, we are left with only 8 parameters. As we will show, several of these parameters are easy to specify and we can also use some conditional independence not visible in the structure. For example, there is no need to compare the effectiveness of different IDSs, but all values are set in relation to the current IDS and the underlying attack it tries to detect. In principle, one needs to consider the following three areas: the underlying risk of the attack, the likelihood of IDS degrading tricks, and the false positive / false negative rate. These are discussed in detail below.

Underlying Risk of the Attack. Starting with *inv-A*, being the node that signifies whether the attack is serious enough to warrant an investigation, we need to set a value for the probability $P(\textit{inv-A} = \mathrm{T})$, known as the *prior probability* in a Bayesian model. We consider this to be the most difficult value to specify in the model. Axelsson [4] has discussed the problems of setting certain parameters for an anomaly detection system. If false positives are more acceptable than false negatives one should exaggerate the risk of the attack, which we have done in our example.

Likelihood of IDS Degrading Tricks. In Section 3.2, we introduced the r-nodes for observations that may affect the IDS's detection capability. For a network-based IDS, this may include encrypted traffic, different types of obfuscating techniques, a heartbeat message, and so on. In Figure 2, we have only one such node, r_1^0. This node signifies whether the web request is encrypted (a typical failure mode for a network IDS). Thus, we estimate how often the web requests are encrypted.

We can then move on to w^1, a node that signifies whether the sensor is working correctly but which is never directly observed. If the traffic is encrypted, we consider it very unlikely that the sensor is working. However, the sensor may fail for conditions other than encrypted traffic, and thus we let this be reflected in the estimate for $P(w^1|\neg r_1^0)$.

False Positive / False Negative Rate. For the node a_1^1, which signifies whether the sensor outputs an alert from rule 1762, the first two parameters are easy to set. When the sensor is not working (the first two rows), we do not expect to see any alerts. Formally, a_1^1 is conditionally independent of *inv-A*, given that the sensor is broken ($\neg w^1$). For the last two rows, we need to determine the value of these parameters:

The false positive rate, $P(a_1^1|w^1, \neg \textit{inv-A})$, i.e., the probability that the alert is triggered when there is no attack.

The power, $P(a_1^1|w^1, \textit{inv-A})$, i.e., how likely the attack will trigger the rule 1762 in Snort. This is easily calculated from the *false negative rate* (FNR) of the rule: $1 - \mathrm{FNR}$.

Both of these values are well known to the security officer and already indirectly used when manually deciding whether an alert is worth extra examination. If the alert comes from a rule that has many false alarms, the security officer will probably not follow up unless there is further evidence.

To summarize, we need to specify eight parameters, but because of the domain and the inherent structure we are left with four values that the security officer is already familiar with. These values can be estimated by the IDS vendor, and then fine-tuned at the local site. As we specified in Section 3.2, parameter estimation for Bayesian networks is quite robust and a correct absolute number is seldom necessary as long as the magnitude of the numbers correspond. To emphasize this fact, we have used a set of predefined ranges to define our model: `never`, `low`, `low-medium`, `medium`, `medium-high`, `high`, and `always`. In Section 5 we replace these labels with their corresponding numerical probabilities and show that the model is robust against some error when estimating these values. When running the examples with real traffic, one would rather tune the values according to knowledge of the network environment and the specifics of the actual alert rule.

4 Example Scenarios

We use two scenarios to exemplify the model. We limit our discussion to attacks directed at web servers and related software. The web server is a complex piece of software, often outside the perimeter defense and accessible to anyone in the world. To complicate matters, the web server often forwards requests to inside resources (legacy databases) that were never designed with a robust security model. Being the analogy of the front door to a company, numerous attacks have been directed toward web servers and the resources with which they communicate. There also exist open-source web server alternatives that are mature enough to allow direct instrumentation (to collect security-relevant events). We use the basic *phf attack* to illustrate the principles of our model (see Section 3.3).

Example 1 is how we foresee the typical use of our model: using several complementary sensors together to increase the accuracy of the system as a whole. This is also the easiest application of the models, as the parameters can be reused.

Example 2 shows a deployment that is fairly common among users of IDSs, with one sensor on the outside of a firewall and another one on the inside. Even though it is more complicated than the typical use of our model (example 1), one can easily foresee how some of the settings could automatically be set by an automatic tool.

4.1 Example 1: Two Sensors Using Different Audit Streams

In this scenario we deploy two sensors using different audit sources (Figure 3). The first is the networked-based IDS Snort used previously, and the other one is developed by us and called *webIDS* in this paper. *webIDS* uses events from within the web server for its analysis. It is a variant of the system described by Almgren et al. [2], i.e., a signature-based system using pattern matching similar to that in Snort.

Using two complementary systems improves the attack detection capability, as shown by Almgren [3]. The Snort system uses rule 1762 described above, while the webIDS has the following rule for this scenario:

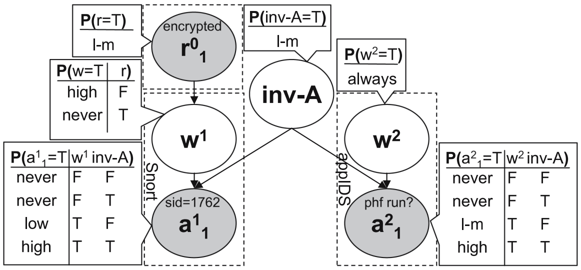


Fig. 3. Model for using Snort together with a sensor within the web server (example 1)

webIDS 1: detects whether the phf program is run successfully by the web server.

Clearly, this rule will have some false positives if the phf program is normally used in the environment. However, an alert from webIDS 1 coupled with an alert from Snort 1762 means that an attack was launched and the script executed successfully.

The model is shown in Figure 3. Adding independent IDSs to the model does not change the already-existing parts. For that reason, the Snort part remains the same and we reuse the values from Figure 2. We only need to add parameters for w^2 and a_1^2. To simplify the model, we assume that the webIDS is very resistant to failures and set $P(w^2 = \text{T})$ to be close to one (always). We define the CPT for a_1^2 in a similar fashion as was done for a_1^1 in Section 3.3. Note that we exclude any dependency between the IDSs to simplify the model.

4.2 Example 2: Two Sensors on Opposite Sides of a Firewall Proxy

In this scenario we monitor an internal web server, protected by a firewall / web proxy. We use one instance of Snort (S_2) to monitor traffic outside the proxy and another instance of Snort for the inside traffic (S_1). The resulting model is shown in Figure 4. The proxy *should* block all web-related traffic. As long as the proxy works as expected, we expect that all attacks are blocked. Thus, even if S_2 reports about attacks, these can safely be ignored as long as S_1 is quiet. However, if S_1 is broken or taken offline, one should ensure that the proxy is working as expected.

In this scenario, we want to show how two (identical) versions of Snort still can be seen as somewhat independent given that they analyze different traffic streams and thus are used in collaboration in our model.

We show the resulting model in Figure 4. For this scenario, we made several changes compared to the model shown in Figure 2. First, we replaced the type of failure observation node from an observation of encrypted traffic to an observation of a heartbeat message. This change is done to show that one should use a diversity of r-nodes, even though we restrict them in this paper for clarity. Having S_1 inside of the proxy implies that alerts from this sensor are more serious than alerts from a sensor without a filtering proxy (as the one in Example 1, shown in Figure 3).

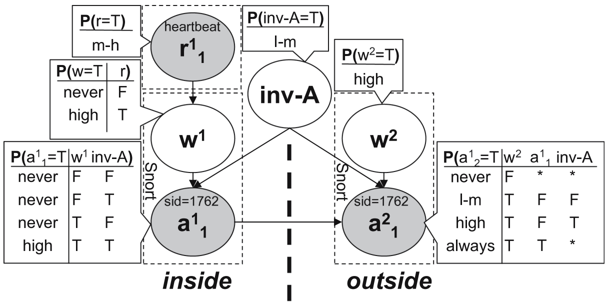


Fig. 4. Model for using two versions of Snort with one outside a proxy and the other one inside (example 2)

Thus, we lowered the probability for $P(a_1^1|w^1, \neg inv\text{-}A)$, meaning that we expect fewer false alarms (in the sense that they are not worth further investigation) from this sensor. For this example, we say that all alerts from the inside sensor should be investigated.

Furthermore, we have added an explicit dependence between a_1^1 and a_1^2. As we run two versions of Snort, we expect that any alert-raising traffic on the inside also exists outside, i.e., S_1 sees a subset of all the traffic passing by S_2. The CPT for a_1^2 is shown in the figure. We omitted special r-nodes for w^2 to keep the model simple and to the point.

5 Experiment

We base the simulations and experiments on the two examples described in Section 4. We first simulated the models presented in Figure 3 and Figure 4, and then we implemented the models on our test bed. As described above, we concentrate on the *phf attack* described in Section 3.3. Even though this may seem limiting at first glance, it clearly illustrates the principles of our approach. Other attacks can easily be added later.

5.1 Experiment Setup

The models for Example 1 and Example 2 each have three observable nodes. The test series used for the simulations are shown in Table 3. When we refer to the different experiments, we replace the x in the first column with the corresponding example number. As we noted in Section 2.1, we consider case (iv) in Table 1 uncommon with modern sensors. For example, as Snort cannot rebuild the HTTP transaction for an encrypted request, the string matching on the URI fails for rule 1762 and Snort does not produce an alert. Hence, the case TT* for example 1 is unusual in practice and is not included in the test series. Similar reasoning goes for FT* for example 2.

Table 3. Results of the simulation of example 1 and example 2

	Example 1				**Example 2**		
Exp #	P(inv-A	$r_1^1 a_1^1 a_2^1$)	$P(w^1\|\ldots)$	*Comment*	P(inv-A	$r_1^1 a_1^1 a_2^1$)	$P(w^2\|\ldots)$
x-1	0.01	FFF	0.89	no attack / script run	0.02	TFF	0.87
x-2	0.20	FFT	0.80	normal phf (no attack)	0.19	TFT	1.00
x-3	0.30	FTF	0.99	phf attack against server without the script	0.96	TTF	0.08
x-4	0.91	FTT	1.00	attack and script run	0.96	TTT	1.00
x-5	0.05	TFF	0.01	broken sensor, but no script invocation	0.08	FFF	0.84
x-6	0.54	TFT	0.01	broken sensor, and script run	0.54	FFT	1.00

As our focus is on alert reasoning, normal traffic causing no alerts is quite uninteresting. Normal web traffic can be used to track down false alarms on a sensor-per-sensor basis, but it adds little to our analysis of reasoning with the available alerts. Rather, the analysis of these false alarms would be used to tune the parameter estimation of our model. For these reasons, we do not use any normal web requests in our experiment.

In the Comment column in Table 3, we explain what the node status implies and we use this as a basis to decide what type of traffic to use in the experiment for that particular case. Based on the observable events, we then calculate the posterior probability for node *inv-A*. In a real system, we would most likely collapse this posterior to either *investigate* or *do not investigate* using a decision function. In this paper, we use a simple decision function with a threshold parameter, $0 \leq \tau \leq 1$, where all values that are less than τ are not considered worthy of investigation. We let $\tau = 0.5$, thus choosing the most probable class.

For the simulation and the experiment, we map our labels to actual probabilities. Instead of a range (represented by our labels), a Bayesian network needs an actual probability (as we have shown in Section 3.3). Thus, we collapse the ranges and let `never`, `low`, `low-medium`, `medium`, `medium-high`, `high`, and `always` map to the following values (in order): $[0.01, 0.1, 0.25, 0.5, 0.75, 0.9, 0.99]$. These values could be fine-tuned. However, in Section 5.4 we show that such fine-tuning is not always necessary because the model is robust against minor estimation errors.

5.2 Simulation

We simulated the models used in Examples 1 and 2 by setting the observable nodes (columns 3 and 7) to a specific value and then calculating the posterior probability for the other nodes. The simulation results of Examples 1 and 2 are presented in Table 3. Each experimental series is prefixed with the scenario number when we discuss it in the text.

5.3 Experiment

We implemented the two scenarios in our test bed. The implementation used the SMILE reasoning engine [10].

For Example 1, we ran *normal phf requests* and *attack phf requests* past a Snort sensor to a web server that either *had a script called phf installed*, or *no such script*. We repeated the experiment with *encrypted traffic*. The alerts were collected, and analyzed by our reasoning engine. The results correspond to the values shown in Table 3.

For Example 2, we sent normal phf requests and attack phf requests to a web proxy. The proxy either forwarded the result to a web server (proxy broken) or returned a message stating that the script did not exist (proxy working correctly). We used two versions of Snort to monitor the traffic. The first version monitored the traffic to the proxy, while the second monitored the traffic to the web server. We added a special rule to simulate a heartbeat for Snort. We simulated the failure of the inside Snort sensor by simply killing the process. All alerts were collected and analyzed. The results correspond to the values shown in Table 3. Not shown in the table is $P(w^1 = T|\ldots)$ for Example 2. These values are similar to the ones shown in column 4 for Example 1. For example, when there is no heartbeat the probability of S_1 working is only 0.01.

Based on our decision function, we would thus further investigate case *1-4* and case *1-6* for Example 1 and case *2-3*, case *2-4*, and case *2-6* for Example 2.

5.4 Sensitivity Analysis

One weakness of the model is the difficulty of accurately estimating the parameters. For that reason, we exhaustively perturbed each estimate in the models by 20% and then recalculated the probability for node *inv-A*. In Example 1, we have 13 independent parameters and the option of subtracting or adding 20% to each estimate gives a total of 2^{13} test cases. For each test case, we then compared the outcome of the decision function (i.e., *investigate* or *do not investigate*) with the outcome from the unperturbed network. There was no difference, implying that the model is relatively robust against estimation errors.

6 Discussion

Column 2 and column 6 in Table 3 show the probability of having a serious attack that needs investigation, given the observable evidence in the network. In a real system, as we specified above, we would most likely collapse the values to *investigate* or *do not investigate* using a decision function. Now let us go through the results in detail.

6.1 Analysis of Scenario 1

In the upper half of Table 3 for Scenario 1, we do not encrypt the requests. Thus, both sensors work and we require alerts from both sensors to investigate the attack. If only the web sensor raises an alert, (*1-2*), no attack code was detected by Snort and it was most likely a normal request. This is reflected in the relatively low value of $P(\text{inv-A}|\ldots)$ for this case. The same holds for case *1-3*. Here, only Snort raises an alert so no phf script was run on the server and thus the attack did not propagate within the web server. If both sensors alert, we should investigate further as illustrated by the high value for case *1-4*.

In the lower part of Table 3, we encrypt all requests. There is no alert, but observed encrypted traffic is not regarded as very sensitive, and thus case *1-5* is rather low. However, note that it is five times that of case *1-1* as we have only one working sensor. If we have an alert from the webIDS when the traffic is encrypted (case *1-6*), the system indicates that we should investigate further. The missing alert from Snort is either because there is no attack, or because Snort is not working. Looking closer at the model for this case, we can determine that $P(w^1 = T|\ldots) = 0.01$, i.e., that the Snort IDS cannot properly analyze the request.

6.2 Analysis of Scenario 2

Now let us have a look at the results for Scenario 2 in Table 3. In the upper half of the table, we receive a heartbeat from the Snort sensor placed inside the proxy, meaning that it should work. When we have no alert (case *2-1*) or when only the outside sensor raises an alert (case *2-2*), the risks are relatively low as indicated in Table 3. Case *2-3*, on the other hand, is interesting. We have an alert only on the inside, which seems impossible as the outside Snort sensor should see all traffic that the inside sensor can see. The only explanation, properly deducted in the model, is that the outside sensor is broken ($P(w^2 = T|\ldots) = 0.08$). The alert should be investigated as indicated in the table, and sensor 2 should most likely be restarted. In case *2-4*, we have alerts from both sensors and thus the alert should be investigated.

In the lower part of Table 3, there is no heartbeat from the Snort sensor placed inside the proxy. For that reason, an alert from the outside sensor is deemed to be much more serious, as can be seen for case *2-6*. In this case, the lack of alert from the inside sensor is explained by a broken sensor (S_1) as there is no longer a heartbeat (not explicitly shown in the table but discussed in Section 5.3).

6.3 Summary

The interesting cases are thus how the model can directly adapt to changes in the environment. The evaluation of case *1-2* is very different from case *1-6*, despite the fact that we in both these cases have an alert only from the webIDS. The same goes for case *2-2* and case *2-6*. The model can also make predictions for when a sensor is broken, as in case *1-5*, case *1-6*, case *2-3*, case *2-5*, and case *2-6*.

The examples we used were designed to illustrate the basic principles of our model while being easy to understand. In real operational settings, the models would be slightly more complex. For example, when using Snort, it would be prudent to have indicators of both encrypted traffic (as in Example 1) and heartbeats (as in Example 2). It would be easy to modify the model to incorporate information from two such nodes. One can also imagine using the results from a vulnerability scan to adjust the weight of the evidence—indications of an attack for which the target is not vulnerable would be given lower weight.

7 Related Work

Several research groups have presented correlation techniques that are able to cluster related alerts, thus presenting the security operator with sets of related alerts belonging to (it is hoped) a single attack. Even though these techniques reduce the number of alerts the security officer needs to consider at a single time, they do not alleviate the actual analysis of the alerts to decide whether an attack is in progress. As we stated in Section 1, we find our approach complementary and we even assume that such a traditional correlator preprocesses the data before it is given to the model presented in this paper. See the excellent overview given by Kruegel et al. [14] and the references therein.

Other correlation efforts have tried to recognize predefined attack scenarios (Debar et al. [9]) or correlating based on the capabilities gained by the attacker from each attack step (Ning et al. [18], Cheung et al. [6], Cuppens et al. [7], and Zhou et al. [23]). Even though some of these approaches account for an imperfect alert stream with missed attack steps, they do not resolve conflicting evidence or model the IDS failure modes as we do. Our approach increases the accuracy of the alert stream and would thus also increase the performance of these higher-level correlation efforts.

Other researchers have focused on using several sensors to collect information about attacks. Abad et al. [1] use a data mining approach to correlate information from different logs on a single host. The underlying ideas are similar to those of our approach. However, we include negative information (no alert) when judging whether an attack is in progress and also try to explain the missing information.

Dagorn [8] discusses a cooperative intrusion detection system for web applications. Thresholds are avoided and instead a Bayesian framework is used, where each node has a twin to measure its confidence level. In our approach, we use a much more constrained view of the sensors and their capabilities but in return we can then reason more about alerts we have.

Tombini et al. [20] combine an anomaly-based IDS with a misuse IDS. They have an enlightening discussion concerning combinations of alerts from the two systems but they focus on a serial combination of the IDSs as opposed to our approach, and they do not consider sensor failure.

Morin et al. [17] introduce a formal model for correlation and discuss some scenarios where this model can be used. Even though Morin et al. describe the need to solve inconsistencies in the output from several IDSs, they do not show any model to do so. Morin et al. [16] also show an application of Chronicles to IDS. However, in this paper they explicitly state that they only use available alerts. In our approach, we also take advantage of false negatives.

Kruegel et al. [13] describe an IDS for analyzing operating system calls to detect attacks against programs. They use the same decision framework from artificial intelligence (i.e., Bayesian networks) as we do, but explore a different problem from the one presented here.

The two approaches most similar to ours are Yu et al. [21] and Zhai et al. [22]. The former tries to predict the intruder's next goal with hidden colored petri nets. They infer missing alerts and reason about alerts using an exponentially weighted Dempster-Shafer theory of confidence. They do not, as we do, explicitly model an IDS weakness to use missing alerts as evidence against an ongoing attack.

Zhai et al. [22] use Bayesian networks to combine event-based evidence (intrusion detection alerts) with state-based evidence (observations in the environment) by chaining them together in a causal structure. Even though their model considers false negatives in a limited way, they do not account for the failure modes of the IDS and thus cannot explain why or how an attack was missed. A consequence is that they also do not use true negatives as evidence against an attack in the same way we do.

Finally, we would like to mention tools such as Thor (Marty [15]). An extension to Thor, for example, would automate the need to manually build correlation tables and set the parameters that are needed for our model.

8 Future Work

We would like to extend the sensor models we have started to build. We would also like to run the system in more challenging environments to learn more about its limitations and how the model can be improved. For example, we have considered adding a general *threat* node. This would allow the system to increase its sensitivity in certain scenarios and lower it in others, based on input from the security operator. In addition, we would like to investigate how to build a sensor that is better tailored to the requirements posed by our model.

9 Conclusions

We have proposed and investigated an intrusion detection model that can analyze alerts from several audit sources to improve the detection accuracy of the intrusion detection system (IDS) as a whole. Our model, expressed in the form of a Bayesian network, can resolve seemingly conflicting evidence collected from different audit sources, thus making it different from other cluster-based correlation approaches. We explicitly model the transitory state of the IDS sensor and can therefore reason about the case when an alert is not produced (a negative) in addition to the case when an alert is produced (a positive).

We validate our model in two scenarios in our test bed. We show that not only can the model correctly reason about evidence collected from several audit sources, but it can also point out when a sensor seems to have failed.

Acknowledgments. The authors are grateful for valuable comments from our colleagues Daniel Hedin, Marina Papatriantafilou, David Sands and Alfonso Valdes. This material is based upon work supported by the Swedish Emergency Management Agency.

References

1. Abad, C., Taylor, J., Sengul, C., Yurcik, W., Zhou, Y., Rowe, K.: Log correlation for intrusion detection: A proof of concept. In: ACSAC 2003: Proceedings of the 19th Annual Computer Security Applications Conference, p. 255. IEEE Computer Society, Los Alamitos (2003)
2. Almgren, M., Debar, H., Dacier, M.: A lightweight tool for detecting web server attacks. In: Tsudik, G., Rubin, A. (eds.) Network and Distributed System Security Symposium (NDSS 2000), San Diego, USA, Feburary 3–4, 2000, pp. 157–170. Internet Society (2000)
3. Almgren, M., Jonsson, E., Lindqvist, U.: A comparison of alternative audit sources for web server attack detection. In: Erlingsson, Ú., Sabelfeld, A. (eds.) 12th Nordic Workshop on Secure IT Systems (NordSec 2007), October 11–12, pp. 101–112. Reykjavík University, Iceland (2007)
4. Axelsson, S.: The base-rate fallacy and its implications for the difficulty of intrusion detection. In: Proceedings of the 6th ACM Conference on Computer and Communications Security, November 1999. Kent Ridge Digital Labs (1999)
5. Breese, J., Koller, D.: Tutorial on Bayesian Networks. Internet (1997), http://robotics.stanford.edu/~koller/BNtut/BNtut.ppt
6. Cheung, S., Lindqvist, U., Fong, M.W.: Modeling multistep cyber attacks for scenario recognition. In: DARPA Information Survivability Conference and Exposition (DISCEX III), Washington, DC, April 22–24, 2003, vol. I, pp. 284–292 (2003)
7. Cuppens, F., Miege, A.: Alert correlation in a cooperative intrusion detection framework. In: Proceedings of the IEEE Symposium on Security and Privacy, Oakland, CA, May 2002, pp. 202–215. IEEE Press, Los Alamitos (2002)
8. Dagorn, N.: Cooperative intrusion detection for web applications. In: Pointcheval, D., Mu, Y., Chen, K. (eds.) CANS 2006. LNCS, vol. 4301, pp. 286–302. Springer, Heidelberg (2006)
9. Debar, H., Wespi, A.: Aggregation and correlation of intrusion-detection alerts. In: RAID 2000: Proceedings of the 4th International Symposium on Recent Advances in Intrusion Detection, pp. 85–103. Springer, Heidelberg (2001)
10. Decision Systems Laboratory, University of Pittsburgh. SMILE reasoning engine for graphical probabilistic model (2008), http://dsl.sis.pitt.edu
11. Domingos, P., Pazzani, M.: On the optimality of the simple Bayesian classifier under zero-one loss. Machine Learning 29(2-3), 103–130 (1997)
12. Hernan, S.V.: 'phf' CGI script fails to guard against newline characters. CERT/CC; Internet (January 2001), http://www.kb.cert.org/vuls/id/20276
13. Kruegel, C., Mutz, D., Robertson, W., Valeur, F.: Bayesian event classification for intrusion detection. In: ACSAC 2003: Proceedings of the 19th Annual Computer Security Applications Conference, p. 14. IEEE Computer Society, Los Alamitos (2003)
14. Kruegel, C., Valeur, F., Vigna, G.: Intrusion Detection and Correlation. Advances in Information Security, vol. 14. Springer, Heidelberg (2005)

15. Marty, R.: Thor - a tool to test intrusion detection systems by variations of attacks. Master's thesis, Swiss Federal Institute of Technology (ETH), Institut für Technische Informatik und Kommunikationsnetze (TIK), Zurich, Switzerland (2002), http://www.raffy.ch/projects/ids/thor.ps.gz
16. Morin, B., Debar, H.: Correlation of intrusion symptoms: An application of Chronicles. In: Vigna, G., Jonsson, E., Kruegel, C. (eds.) RAID 2003. LNCS, vol. 2820, pp. 94–112. Springer, Heidelberg (2003)
17. Morin, B., Mé, L., Debar, H., Ducassé, M.: M2D2: A formal data model for IDS alert correlation. In: Wespi, A., Vigna, G., Deri, L. (eds.) RAID 2002. LNCS, vol. 2516, pp. 115–137. Springer, Heidelberg (2002)
18. Ning, P., Cui, Y., Reeves, D.S.: Analyzing intensive intrusion alerts via correlation. In: Wespi, A., Vigna, G., Deri, L. (eds.) RAID 2002. LNCS, vol. 2516, pp. 74–94. Springer, Heidelberg (2002)
19. Swets, J.A.: Measuring the accuracy of diagnostic systems. Science 240(4857), 1285–1293 (1988)
20. Tombini, E., Debar, H., Mé, L., Ducassé, M.: A serial combination of anomaly and misuse IDSes applied to HTTP traffic. In: ACSAC 2004: Proceedings of the 20th Annual Computer Security Applications Conference (ACSAC 2004). IEEE Computer Society, Los Alamitos (2004)
21. Yu, D., Frincke, D.: Improving the quality of alerts and predicting intruder's next goal with hidden colored petri-net. Comput. Netw. 51(3), 632–654 (2007)
22. Zhai, Y., Ning, P., Iyer, P., Reeves, D.S.: Reasoning about complementary intrusion evidence. In: ACSAC 2004: Proceedings of the 20th Annual Computer Security Applications Conference, Washington, DC, USA, pp. 39–48. IEEE Computer Society, Los Alamitos (2004)
23. Zhou, J., Heckman, M., Reynolds, B., Carlson, A., Bishop, M.: Modeling network intrusion detection alerts for correlation. ACM Trans. Inf. Syst. Secur. 10(1), 4 (2007)

Monitoring SIP Traffic Using Support Vector Machines

Mohamed Nassar, Radu State, and Olivier Festor

Centre de Recherche INRIA Nancy - Grand Est
615, rue du jardin botanique, 54602
Villers-Lès-Nancy, France

Abstract. We propose a novel online monitoring approach to distinguish between attacks and normal activity in SIP-based Voice over IP environments. We demonstrate the efficiency of the approach even when only limited data sets are used in learning phase. The solution builds on the monitoring of a set of 38 features in VoIP flows and uses Support Vector Machines for classification. We validate our proposal through large offline experiments performed over a mix of real world traces from a large VoIP provider and attacks locally generated on our own testbed. Results show high accuracy of detecting SPIT and flooding attacks and promising performance for an online deployment are measured.

1 Introduction

The voice over IP world is facing a large set of threats. SPAM on email systems takes a new and very annoying form on IP telephony advertising. This threat is known as SPIT (Spam over Internet Telephony). However, SPIT is not the only threat vector. The numerous software flaws in IP phones and servers affect their reliability and open the door to remotely attack previously unseen in the "stable" world of telecommunication operators (PSTN), which was based on mutual trust among few peers. Leveraging the IP to support voice communications exposes this service (voice) to the known denial of service attacks that can be easily implemented by service or network request flooding on the Internet. Resource exhaustion thus automatically finds its place against SIP proxies and back-to-back user agents, which are essential to support this critical infrastructure. The list of potential threats is huge and ranges from VoIP bots (that could spread by malware and perform distributed attacks, perform SPIT or toll fraud), to eavesdropping and Vishing (similar attack to the Phishing are using VoIP as the transport vehicle) [1].

Securing VoIP infrastructures constitutes one of the major challenges for both the operational and research communities because security by design was not a key component in the early phases of both VoIP research and development. VoIP-specific security solutions are currently required by the market because the research and standardization efforts are still trying hard to address the issues of securing and monitoring VoIP infrastructures.

Our work fits into these efforts and addresses a new monitoring approach for VoIP specific environments. Our monitoring scheme is based on Support Vector Machines for efficient classification. We continuously monitor a set of 38 features in signaling time slices and use these features as the raw input to the classification engine. A threshold based alarm generator is placed on top of the classification engine. We show that the system is both efficient and accurate and study the impact of the various features on the efficiency.

We start the presentation with a short survey on VoIP security with focus on flooding attacks and SPIT. We then give a functional description of our monitoring solution together with the definition of the 38 features computed in our system for classification (section 3). In section 4, we provide a short mathematical background of the SVM learning machine model used in the monitoring process. Offline traces inspection is presented in section 5 where we also describe the data set. Section 6 demonstrates the performances of our approach to detect different types of attacks. Related work is addressed in section 7. Section 8 concludes the paper and enumerates some future work.

2 The Threat Model

2.1 Flooding Attacks

Denial of service attacks can target the signaling plane elements (e.g. proxy, gateway, etc.) with the objective to take them down and produce havoc in the VoIP network. Such attacks are launched by either flooding the signaling plane with a large quantity of messages, malformed messages or executing exploits against device specific vulnerabilities.

The authors of [2] categorize some of these attacks based on the request URI and perform a comparative study of these ones against popular open source VoIP equipment. We adopt the same categorization, i.e.:

- UDP flooding: Since the vast majority of SIP systems use UDP as the transport protocol, a large amount of random UDP packets are sent in an attempt to congest the network bandwidth. Such attacks produce a high packet loss. Legitimate call signaling has thus a reduced probability to reach the target and to be processed.
- INVITE flooding with a valid SIP URI: The attacker calls one user/phone registered at a server/proxy. The proxy relays the calls to the phone. If the proxy is stateful it will manage a state machine for every transaction. The phone is quickly overloaded by the high rate of calls and is no more able to terminate the calls. As a result, the server is allocating resources for a long time and it will run out of memory.
- INVITE flooding with a non existent SIP URI: If the attacker doesn't know a valid SIP URI registered on the target, it can send calls to an invalid address. The proxy/server responds with an error response like "user not found". When the attack rate is higher than the server capabilities, the resources are exhausted. This type of flooding is less disturbing than the previous one but

the target CPU is loaded with useless transactions and legitimate requests may be rejected.
- INVITE flooding with an invalid IP domain address: The attacker calls a user with a rogue IP address of the destination domain. The target is led to connect several times to an unreachable host/network while keeping the state of the current SIP transaction. This attack is efficient on some proxies like OpenSER [2].
- INVITE flooding with an invalid domain name: The attacker calls a user with a false destination domain name. The target is trapped to send DNS requests to resolve the domain name. The target may issue different DNS types (A, AAAA, SRV, NAPTR, ENUM) and repeat them multiple times. In the same time, the target is managing the transactions waiting for a valid DNS response to proceed. Memory is quickly exhausted. The effect of this attack on the performance of OpenSER is shown in Fig. 1. The impact is evaluated in terms of duration, number of messages exchanged and final state of sessions or transactions. The behavior of the server can be divided in two successive phases. In the first phase, the first few requests are correctly handled (REJECTED) but the session duration is increasing and the proxy is slowing down. The number of messages is increasing because of response retransmissions (no ACK is sent by the attacker). In the second phase, the proxy is no more able to handle the requests (still in CALLSET state) so the proxy is taken down. The take down time is about 20 seconds for an attack having just one INVITE/s rate.
- INVITE flooding with an invalid SIP URI in another domain: The attacker calls a user/phone located in another domain than the target's one. The target relays all requests to the server/proxy of the other domain. The latter replies with an error response. In this way, multiple targets are hit at the same time and cascading failures occur.
- INVITE flooding with a valid SIP URI in another domain: The attacker calls a user/phone registered in another domain. The target relays all requests to the server/proxy of the other domain which sends them to the phone. The phone gets quickly out of service and maintaining the state by the intermediary servers will exhaust the resources from all the servers in the forwarding chain.
- INVITE/REGISTER flooding when authentication is enabled: The attacker sends INVITE or REGISTER messages and then stops the handshaking process. The proxy/registrar responds with a challenge and waits for the request to be send again with the proper authentication credentials. This process is costly for the proxy/registrar in term of computing (generating challenges and nonces) and memory (dialogs/transaction state machines).

2.2 Social Threats and SPIT

Social threats are attacks ranging from the generation of unsolicited communications which are annoying and disturbing for the users to more dangerous data stealing (Vishing) attacks. The threat is classified as social since the term

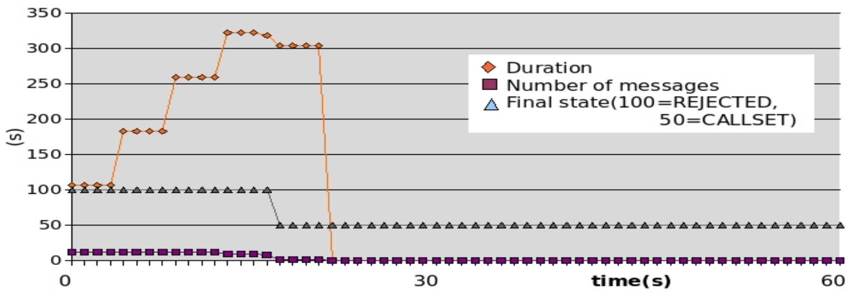


Fig. 1. OpenSER Response to an INVITE Flooding with Invalid Domain Name

"unsolicited" depends on user-specific preferences. This makes this kind of attack difficult to identify. An example of this is a threat commonly referred to as SPam over Internet Telephony (SPIT). This threat is similar to spam in the email systems but is delivered by means of voice calls. This leverages the cheap cost of VoIP when compared with legacy phone systems. It's currently estimated that generating VoIP calls is three order of magnitude cheaper than generating PSTN calls. Such SPIT calls can be telemarketing calls that sell products. A subtle variant of SPIT is the so-called Vishing (VoIP phishing) attack, which aims either to make the callees dial expensive numbers in order to get the promised prize or to collect personal data redirecting the users towards an Interactive Voice Responder (IVR) pretended to be trusted. Most of these attacks are going to be generated by machines (bot-nets) programmed to do such a job. Unsolicited communications (like SPIT or Vishing) are, from a signalling point of view, technically correct transactions. It is not possible to determine from the INVITE message (in the case of SIP) if a VoIP transaction is SPIT or not. From a technical point of view, the challenge is actually higher since the content is not available to help in the detection until the phone rings (disturbing the user) and the callee answers the call. For this reason, techniques successfully used against e-mail spam like text filtering are hardly applicable in the VoIP sphere. Even if a transaction is identified as unsolicited how to handle such a transaction highly depends on the legal environment in the country of the caller.

3 Our Monitoring Solution

When facing the mentioned threats, monitoring of the signalling traffic can detect anomalous situations and prevent them. The monitoring scheme can be quite simple and flexible to support different techniques. Thus, our approach follows these principles. As shown in Fig. 2, we track SIP messages in a queue of predefined size. Once the queue is full, this slice of messages is used to compute a vector of statistics/features. The classifier decides if a vector represents a certain anomaly and issues an alarm event if necessary. This approach is based on a learning phase in which couples (vector, class Id) have been used to feed the engine for learning. This learning process can be made on the fly during the operational phase of the monitoring system by allowing it to update the prediction model over time. Finally, an event correlator or decider has to filter and correlate the events. It generates an alarm for a group of events if they trigger one of the rules/conditions. e.g. if the number of events of type i bypasses a certain threshold in a period of time t.

The architecture is modular and enables experimenting with different classification and artificial intelligence techniques ranging from statistics and information theory to pattern classification and machine learning. The pace of the system t_{pace} is the time it takes to make a decision about one slice without accounting for the time needed by the event correlation stage. This time is composed of two components: the analysis time of the processor and the machine time of the classifier. The design achieves real time pace if t_{pace} is less than the size of the slice S divided by the arrival rate of messages λ:

$$t_{pace} = t_{analysis} + t_{machine}$$
$$t_{pace} < \frac{S}{\lambda}$$

We define in the following the important features that characterize a slice of SIP traffic and motivate why we collect them. We divide these features in four groups:

– **General Statistics:** are number of requests, number of responses, number of requests carrying an SDP (Session Description Protocol) body, average inter requests arrival time, average inter response arrival time and average inter requests arrival time for requests having SDP bodies; these statistics represent the general shape of the traffic and indicate the degree of congestion. The fraction of requests carrying SDP bodies (normally INVITE, ACK or UPDATE) is a good indicator because it will not exceed a certain threshold. An excessive use of re-INVITE or UPDATE for media negotiation or maybe QoS theft increases the number of SDP bodies exchanged and decrements the average inter-arrival of them. Flooding attacks are associated with peaks of all these statistics.
– **Call-Id Based Statistics:** are number of Call-Ids, average of the duration between the first and the last message having the same Call-Id, the average number of messages having the same Call-Id, the number of different senders (the URI in the From header of a message carrying a new Call-Id) and the number of different receivers (the URI in the To header of a message carrying a new Call-Id). Similar to the Erlang model used in the telecommunication networks, where the arrival rate of calls and the average duration of a call characterize the underling traffic, the arrival rate of Call-Ids (can be starting a call or any kind of SIP dialog) and the interval time of messages having the same Call-Ids, can be used to characterize the overlay SIP traffic. Nevertheless, we notice that non-INVITE dialogs have shorter durations and fewer number of messages than INVITE dialogs. Thus their Call-Id statistics can be taken as different features.

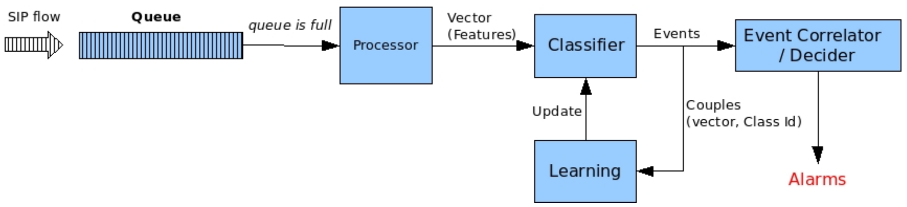


Fig. 2. Real-time Online SIP Traffic Monitoring

- **Distribution of Final State of Dialogs/Call-Ids:** Since we are using a limited number of messages in the traffic analysis unit, dialogs can be partitioned into two or several units/slices. The final state of a dialog at the analysis moment is considered and this one is not necessarily the final state when all the messages of the dialog can be taken into account. The following states are defined: NOTACALL: for all non-INVITE dialogs, CALLSET: for all calls/INVITE dialogs that do not complete the initiation, CANCELED: when the call is cancelled before it is established, REJECTED: for all redirected or erroneous sessions, INCALL: when the call is established but not realized yet, COMPLETED: for a successful and ended call and RESIDUE: when the dialog does not start with a request. This latter is a residual of messages in a previous slice. In a normal situation where the size of the unit is large enough, NOTACALL, COMPLETED and REJECTED (in busy or not found situations) dominate this distribution. Major deviations may indicate an erroneous situation.
- **Distribution of SIP Requests:** are INVITE, REGISTER, BYE, ACK, OPTIONS, CANCEL, UPDATE, REFER, SUBSCRIBE, NOTIFY, MESSAGE, INFO, PRACK. Although the first five types represent the main methods used in SIP, every other type may point out a specified application running above. The number of REGISTER sent by a user within a time interval is indirect proportional to the period of registration (`expires` parameter or `Expires` header). Obviously, the total number of REGISTER messages is proportional to the number of users of the domain and inversely proportional to the average period of registration among all users. The existence of SUBSCRIBE and NOTIFY messages indicates SIP presence services. Instant messaging can also be revealed by MESSAGE requests. REFER requests may reveal a SIP peer to peer application or some call transfer running above. INFO requests are normally used to carry out of band DTMF tones within PSTN-VoIP calls. Finally, PRACK requests may reveal VoIP to PSTN activity.
- **Distribution of SIP Responses:** are Informational, Success, Redirection, Client Error, Server Error, Global Error. An unexpected high rate of error responses is a good indication for error situations.

Among the different scientific approaches in the area of classification (Bayesian networks, decision trees, neural networks), we have chosen the support vector machines approach for their superior ability to process high dimensional data [3,4]. SVM is a relatively novel (1995) technique for data classification and exploration. It has demonstrated good performance in many domains like bioinformatics and pattern recognition (e.g. [5] and [6]). SVM has been used in network-based anomaly detection and has demonstrated better performance than neural networks in term of accuracy and processing proficiency [7]. In the next section, we give a short description of the SVM concept and methodology.

4 Support Vector Machines

Principle. Given a set of couples $S = (\overrightarrow{x_l}, y_l)_{1 \leq l \leq p}$, with $y_l \in \{-1, +1\}$, which denotes the correct classification of the training data, the SVM method tries to distinguish between the two classes by mean of a dividing hyperplane which has as equation $\overrightarrow{w}.\overrightarrow{x} + b = 0$. If the training data are linearly separable, the solution consists in maximizing the margin between the two hyperplanes, $\overrightarrow{w}.\overrightarrow{x} + b = +1$ and $\overrightarrow{w}.\overrightarrow{x} + b = -1$, such that for all points either $\overrightarrow{w}.\overrightarrow{x} + b \geq +1$ (1) or $\overrightarrow{w}.\overrightarrow{x} + b \leq -1$ (2). This is equivalent to minimizing the module $|\overrightarrow{w}|$ because the distance between the two mentioned hyperplanes is $2/|\overrightarrow{w}|$. The resulting quadratic problem where the conditions (1) and (2) are aggregated is formulated as:

$$\boxed{\begin{array}{c}\text{Find } \overrightarrow{w} \text{ and } b \text{ to minimize } \frac{1}{2}\overrightarrow{w}.\overrightarrow{w} \\ \text{so that } y_l(\overrightarrow{w}.\overrightarrow{x_l}) + b \geq 1 \forall (\overrightarrow{x_l}, y_l) \in S\end{array}}$$

The non linear separation has a similar formulation except that we replace the dot product by a non-linear kernel function. The kernel function takes the data set to a transformed feature space where it searches the optimal classifier. The transformation may be non-linear and the transformed space high dimensional. The maximum-margin hyperplane in the high-dimensional feature space may be non-linear in the original input space. The following kernels can be used :

- linear $K_l(\overrightarrow{x}, \overrightarrow{z}) = \overrightarrow{x}.\overrightarrow{z}$
- polynomial $K_d(\overrightarrow{x}, \overrightarrow{z}) = (\gamma \overrightarrow{x}.\overrightarrow{z} + r)^d$, $\gamma > 0$
- radial basis function $k_{rbf}(\overrightarrow{x}, \overrightarrow{z}) = exp(-\gamma|\overrightarrow{x} - \overrightarrow{z}|^2)$ where $\gamma > 0$
- sigmoid $k_s(\overrightarrow{x}, \overrightarrow{z}) = tanh(\gamma \overrightarrow{x}.\overrightarrow{z} + r)$, $\gamma > 0$ and $r < 0$

The C-SVC (C parameter - Support Vector Classification) approach is particularly interesting when the training data is not linearly separable.

C-SVC. For the general case where the data S is not separable, the solution allows some points to be mislabeled by the separating hyperplane. This method, so called soft margin, is expressed by the introduction of slack variables ξ_l where ξ_l measures the degree of misclassification of a point x_l. The objective function is then increased by a function which penalizes non-zero ξ_l, and the optimization becomes a trade off between a large margin, and a small error penalty.

$$\boxed{\begin{array}{l}\text{Find } \overrightarrow{w}\text{, b and } \xi \text{ to minimize } \frac{1}{2}\overrightarrow{w}.\overrightarrow{w} + C\sum_l \xi_l \\ \text{so that } \begin{cases} y_l(\overrightarrow{w}.\overrightarrow{x_l}) + b \geq 1 - \xi_l, \forall(\overrightarrow{x_l}, y_l) \in S \\ \xi_l \geq 0, \forall l \end{cases}\end{array}}$$

5 Monitoring SIP

We aim to detect anomalies within a SIP traffic capture, demonstrate the accuracy of the learning machine to identify attacks and non-attacks and distinguish between different types of attacks. We have performed an extensive analysis on offline VoIP traces in order to assess the performance of our approach.

We use the popular LibSVM tool [8] which contains several algorithms for classification and regression. In addition, the tool provides support for multi-class classification and probability estimates (so a test vector x_i seems to be of class i with a probability p_i) as well as support for one class SVM training. LibSVM is bound to other several tools such as an algorithm that performs a grid search over the SVM parameters space and optimizes their values by cross validation (divide the data into n subsets, for i going from 1 until n, learn over all the subsets except subset number i then test over subset number i). At the end, we can measure the test accuracy for each subset. The aggregation of all results is the accuracy given by the selected parameters. In Fig. 3 we illustrate this tool's flow. The data we use in this study originates from two different sources. The first source is traffic from a real-world VoIP provider and it is supposed to be completely normal. The second source is signaling traffic from a small test-bed installed by us to generate different forms of SIP attacks. We have three types of data files: clean and normal trace, clean attack trace, and mixed trace which is a normal trace where attack is injected.

To be processed by the SVM tool, each data file is cut into slices and entered into the analyzer. For each slice, the analyzer evaluates a set of predefined features (38 variables are defined in our study) and builds a vector for the LibSVM. All vectors are assembled in one file and annotated as either attack vector or normal vector. In Fig. 4, this process is shown for a mixed trace.

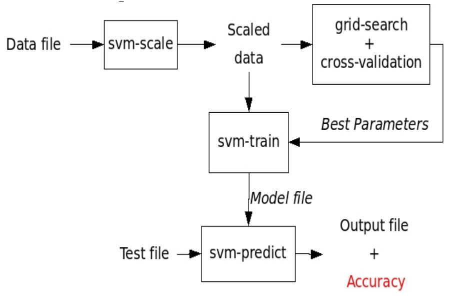


Fig. 3. SVM Flow Chart

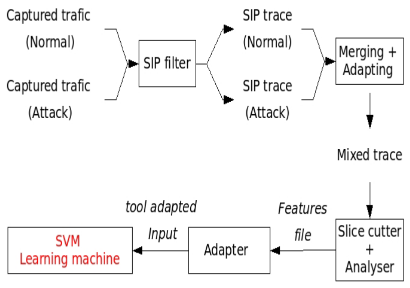


Fig. 4. Analysis Flow Chart

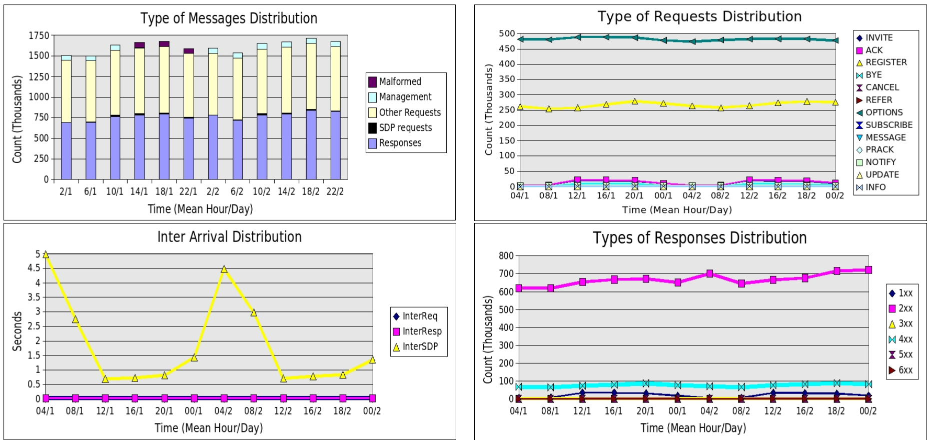


Fig. 5. Long Term Statistics over Real World Traces

5.1 Normal Traffic

The input data is a SIP trace from a two days continuous capture at a real world VoIP provider server. We performed a preliminary long term analysis of the traces with a two hours step. We depict the results in the four charts of Fig. 5. If we consider the distribution of different SIP messages, we can remark the following:

- The two main components of the traffic are the OPTIONS messages in the first place and then the REGISTER messages.
- Some methods are absent from the capture such a MESSAGE, PRACK and UPDATE.
- Some methods like NOTIFY have constant statistics over all periods of the day which reveal SIP devices remaining always connected and periodically sending notifications.
- The three main components of the call signalling (INVITE, BYE and ACK) have practically constant ratios over all the slots, with an average ratio $\#INVITE/\#BYE = 2.15$ and $\#INVITE/\#ACK = 0.92$.

Response distribution is dominated by the 2nd response class (most of them belong to OPTIONS and REGISTER transactions). 3xx, 5xx and 6xx are very rare while informational responses (1xx) follow INVITE messages because they are exclusively used in INVITE transactions (the average ratio $\#INVITE/\#1xx = 0.59$ can be explained by the fact that a call probably regroups one 100 Trying and one 180 Ringing so two 1xx responses). Average inter-request arrival and average inter-response arrival seem to be constant over all periods and they are about 20 ms. While average inter-request carrying SDP bodies which are exchanged in call dialogs move inversely to the quadruple (INVITE-BYE-ACK-1xx) curve, they reach 3s in quiet hours and decrease to 0.5s in rush hours.

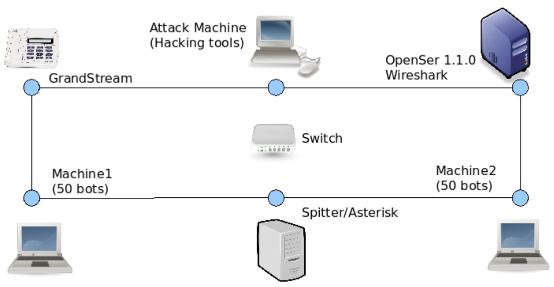


Fig. 6. Testbed of Attack Generation

5.2 The Testbed

The testbed consists of one OpenSER server and three other machines: the first machine plays the role of the attacker and uses a number of hacking tools (scanning, flooding, SPIT). The two other machines play the role of victims where one hundred SIP bots are equally distributed and running. The bots are programmable SIP agents and are controlled by an IRC channel[1]. All SIP bots and a GrandStream hardphone are registered to the OpenSER server and all machines belong to the same domain. Traces of attacks performed by the attacker machine are collected at the OpenSER server.

6 Performance and Accuracy

All experiments are done in a machine which has an Intel Pentium 4 CPU 3.40GHz and 2GB RAM memory running a Linux kernel 2.6.18-1. In term of performance, experiments show that a file containing 2449 slices/vectors of 38 features takes between 196 and 994 ms in the SVM prediction stage (depending on the used kernel).

Coherence Test

The first question we addressed was how many of the normal traces are self-similar and consistent. For example, is traffic from 22:00 to 02:00 from a day similar to traffic of the same period in another day? To test the coherence between two given traces, we used the following procedure: the analyzer evaluates feature vectors from each trace. Vectors are then labeled with respect to the origin trace and scaled. We make a 2-fold training test over all the vectors. In a 2-fold test, training is done over one part of the file and the testing is performed over the second. We define the coherence to be indirect proportional to the resulting accuracy of the 2-fold cross training. As long as the SVM can not distinguish between the two traces, they are tagged to the same class. In Table 1, we summarize some results:

[1] `http://www.loria.fr/~nassar/readme.html`

Table 1. Coherence Test for two Successive Days

Day 1	06-10	10-14	14-18	18-22
Day 2	06-10	10-14	14-18	18-22
Accuracy(%)	55.91	53.72	52.83	56.90

Table 2. Coherence Test for Different Periods of the Same Day

Day 1	02-06	02-06	02-06	02-06	22-02
Day 1	06-10	10-14	14-18	18-22	22-02
Accuracy(%)	51.82	62.79	63.72	63.76	60.80

We tested the coherence of a period with respect to other periods. In Table 2, we show the results of the same procedure for a period of 2-6 of Day 1 compared to other periods of the same day. SVM is not able to label 50% of vectors in the correct class while proceeding with the same period of two successive days and 40% of vectors during different periods of the same day. The second table reveals that period 02-06 is more coherent with neighboring periods (06-10 and 22-02) than with other periods of the day. In conclusion, the coherence of the data is acceptable.

Multi-Class Detection Experiment

We also tested SVM's ability to distinguish between different classes of traffic: for instance traces coming form different VoIP platforms. We built a group of four traces, each representing a different traffic class : normal traffic, a burst of flooding DoS, a trace generated by the KIF stateful fuzzer [9], and a trace generated by an unknown testbed as shown in Table 3. The size of the analyzed slice is fixed to 30 messages. After analysis, a 2-fold training/testing cross test is performed over all respectively labeled vectors (2449 vectors). The test Accuracy is defined as the percentage of correctly classified vectors over all test vectors. When the RBF (Radial Basis Function) kernel is used with default parameters (C=1 and $\gamma = 1/38$), the accuracy is 98.24%.

Table 3. Multi-Class SIP Traffic Data Set

Trace	**Normal**	**DoS**	**KIF**	**Unknown**
SIP pkts	57960	6076	2305	7033
Duration	8.6(min)	3.1(min)	50.9 (min)	83.7(day)

Comparison between Different Kernel Experiments

The RBF kernel is a reasonable first choice if one can assume that the classes are not linearly separable and because it has few numerical settings (small number of

Table 4. Testing Results for Different Kernels

Kernel	Parameters	Accuracy(%)	Time(ms)
Linear	$C = 1$	99.79	196
Polynomial	$C = 1$; $\gamma = 1/38$; $r = 0; d = 3$	79.09	570
Sigmoid	$C = 1$; $\gamma = 1/38$; $r = 0$	93.83	994
RBF	$C = 1$; $\gamma = 1/38$	98.24	668
Linear	$C = 2$	99.83	157
RBF	$C = 2$; $\gamma = 0.5$	99.83	294

parameters, exponential function bounded between 0 and 1). On the other hand, linear and RBF kernels have comparable performance if the number of features is significantly higher than the number of instances or if both are to large [8]. Therefore, we have tested all kernels with their default parameters over our dataset. The accuracy (defined as the percentage of correctly classified messages over all the test results) for 2-fold cross and machine dependent running time are shown in Table 4. The last two lines of the table are for RBF and linear kernels after parameter selection. Machine running time is given for comparison purpose only and it is averaged over ten runs. RBF and linear kernels have clearly better accuracy and execution time. We expect that RBF kernel will bypass linear kernel performance when dealing with larger sets of data.

Size of SIP Slice Experiment

The analyzer window is an important parameter in the feature evaluation process. The size of the slice can be fixed or variable with respect to other monitoring parameters. In this experiment, we report the accuracy of our solution, when changing the size of the analyzed slice. The results shown in Table 5 were obtained using a 5-fold cross test using a RBF kernel and the default parameters. The time the analyzer takes to process the packets is critical in online monitoring. This is the reason why we show the analysis time of the overall trace: (note that values are for comparison purpose). As expected, the accuracy improves with larger window size, which incurs an increased analysis time.

Feature Selection

The 38 features are chosen based on domain specific knowledge and experience, but other features might be also relevant. The selection of highly relevant features is essential in our approach. In the following experiments, we rank these features with respect to their relevance. We can thus reduce the number of features by gradually excluding less important features from the analysis. In Table 6, the

Table 5. Testing Results for Different Kernels

Window size	5	15	30	60	90	120	150
Accuracy (%)	95.4	99.32	99.30	99.67	99.63	100	100
Analysis Time (min)	1.12	2.40	2.56	4.31	6.39	7.42	8.51

Table 6. Results for Decreasing Size of Features Set

# of features	38	31	18	12	7
Accuracy (%)	99.30	99.39	98.90	98.65	98.22
Machine Time (s)	1.85	1.59	1.42	1.28	0.57

results of a preliminary experiment, where we exclude one group of features at each column in the following order: the distribution of final state of dialogs, the distribution of SIP requests, the distribution of SIP responses, and the Call-Id based statistics are given. The last column of the table represents only the general statistics group of features. Experiments use a 5-fold cross test over our data set with RBF kernel and its default parameters. The test accuracy is the percentage of correctly classified vectors over all the vectors in the test data set.

Although we notice a sudden jump between 12 and 7 features, the associated accuracy is not strictly decreasing as a function of number of features used. It is thus reasonable to inquire on the dependencies among the features.

Detection of Flooding Attacks

We have used the Inviteflood tool [2] to launch SIP flooding attacks. We have used INVITE flooding with an invalid domain name (which is the most impacting on the OpenSER server). We have generated five attacks at five different rates, where each attack lasts for one minute. After adaptation (we assume that one machine of the real world platform is performing the attack), each one minute attack period is injected into a normal trace of two hours duration. The time of the attack is fixed to five minutes after the start of the two hours period. Each mixed trace is then analyzed and labeled properly (positively along the period of attack and negatively in all the remaining time).

We have trained the system with the mixed trace (flooding at 100 INVITE/s - normal trace) in the learning stage. This means that 100 INVITE messages are taken as a critical rate (the rate we consider as threshold to launch an alarm).

As shown in Fig. 7 (for simplification and clarity sake a slice is sized to only three packets), we take the period of attack and we calculate the corresponding SVM estimation. The estimated probability is the average of the estimated probabilities for the elementary slices composing the attack traffic. This granular probability is given by the LibSVM tool and is useful for both the probability estimate option in both learning and testing stages. We define the detection accuracy as the percentage of vectors correctly classified as attack over all vectors of the attack period. The results are in Table 7: the detection accuracy-1 is obtained without a parameter selection (Default parameters : $C = 1$, $\gamma = 1/38$,

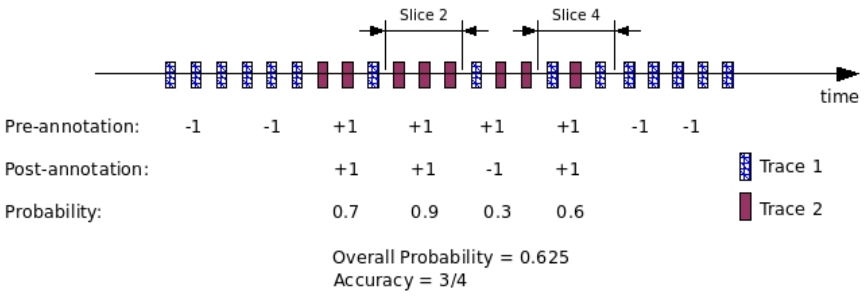

Slice 2
Slice 4
time
Pre-annotation: -1 -1 +1 +1 +1 +1 -1 -1
Post-annotation: +1 +1 -1 +1
Probability: 0.7 0.9 0.3 0.6
Trace 1
Trace 2
Overall Probability = 0.625
Accuracy = 3/4

Fig. 7. Attack Detection in a Mixed Trace

Table 7. Attack Estimation for Different Rates of Flooding

Flooding Rate (INVITE/s)	0.5	1	10	100	1000
Detection Accuracy-1 (%)	0	0	5.47	67.57	97.36
Detection Accuracy-2 (%)	0	1.48	30.13	88.82	98.24
Pr(Normal)	0.96	0.95	0.73	0.24	0.07
Pr(Attack)	0.04	0.05	0.27	0.76	0.93

training accuracy: 90.95), detection accuracy-2 and calculated probabilities are after parameter selection ($C = 32$, $\gamma = 0.5$, training accuracy is of 93.93). We tested the coherence of a period with respect to other periods. In Table 2, we show the results of the same procedure for a period of 2-6 of Day 1 compared to other periods of the same day.

Even though stealthy attacks cannot to be detected, the results show a promising opportunity to fine-tune the defensive solution. The threshold rate can be learnt by a dual trace : the ongoing normal/daily traffic and a stress condition where the server was troubleshooted or was noticed to be under-operating. In this way, SVM is promising for an adaptive online monitoring solution against flooding attacks.

Detection of SPIT Attacks

SPIT mitigation is one of the open issues in VoIP security today. Detection of SPIT alone is not sufficient if it is not accompanied by a prevention system. In-depth search in the suspicious traffic is needed to build a prevention system to block the attack in the future. Elements like IP source and URI in the SIP headers can be automatically extracted.

To generate SPIT calls, we used a well known tool which is the Spitter/Asterisk tool [2]. Spitter is able to generate call instances described in a ".call" file using the open source Asterisk PBX. The rate of simultaneous concurrent calls can also be specified as an option of the attack. We profiled our programmable bots to receive SPIT calls. Once an INVITE is received, the bot chooses randomly between three different responses :

– the first choice is to ring for a random time interval between one and six seconds and then to pick up the phone. This emulates two cases : a voice mail which is dumping a message or a human which is responding. The bot then listens during a random time between five and ten seconds and hangs up,
– the second choice is to respond with ‘Busy’,
– the last choice is to ring for some time and then to send a redirection response informing the caller that the call has to be directed to another destination (destination that we assume to not be served by this proxy). Other similar scenarios like forking (by the proxy) or transferring (by the bot) can also be supported.

We have performed two experiments with two different hit rates. The former is a partial SPIT: Spitter targets the proxy with hundred destinations and among these only ten are actually registered bots. In this case the hit rate is just 10%. This emulates the real world scenario where attackers are blindly trying a list of extensions. The latter is a total SPIT: we assume that attackers knew already the good extensions so the hit rate is 100%. This emulates the real world scenario where attackers knew already the good extensions either by a previous enumerating attack or from a web crawler.

In the partial SPIT experiment (SPIT not covering all the domain extensions, hit rate < 100 %), we send four successive campaigns with respectively one, ten, fifty and hundred concurrent calls. In the first campaign, Spitter does not start a dialog before the previous dialog is finished. In the second campaign, ten dialogs go on at the same time and only when a dialog is finished, a new dialog is started.

The four resulting traces (duration about two minutes each) are injected - after adaptation (we assume that one agent of the real trace is performing the attack against the hundred other agents) - in four different normal traces (duration of two hours each). The traces are then cut into slices of thirty messages and analyzed. These are annotated positively for the period of attack and negatively in all the remaining duration. The mixed trace with fifty concurrent calls SPIT is used in the training stage. The SVM prediction results are shown in Table 8. True positives are the percentage of vectors correctly classified as attack over all the vectors of the attack period. True negatives are the percentage of vectors correctly classified as normal over all the vectors of the normal period. These results should be considered under the larger umbrella of event correlation. For instance, the example with ten concurrent calls:

– Most of the two hours traffic is normal and is correctly detected (47436 slices).
– 16 out of the 766 slices that compose the attack traffic are detected. This means that we have ten correct events in a period of two minutes, because the detection of one slice is highly relevant to all ongoing traffic around this slice.

In addition, the attacks are partial since they target a small fraction of the users of the VoIP server (more than 3000 users are identified in the two hours period).

Table 8. Detection of Partial SPIT in Four Mixed Traces With Different Intensities

# of Concurrent Calls	**True Positives (%)**	**True Negatives (%)**
RBF; C= 1; $\gamma = 1/38$; Training accuracy = 99.0249		
1	0 (0/3697)	100
10	1.30 (10/766)	
50	10.01 (62/619)	
100	18.31 (102/557)	
Linear ; C=1 ; Training accuracy = 99.0197		
1	0 (0/3697)	100
10	2.09 (16/766)	
50	10.66 (66/619)	
100	19.39 (108/557)	

We agree that a stealthy SPIT of the magnitude of one concurrent call is never detected, but in the case of hundred concurrent calls, one over five positives is successfully detected when training was done using a half of this intensity attack.

With the help of a set of deterministic event correlation rules, our online monitoring system is able to detect the attacks efficiently:

Predicate	**SPIT intensity**
10 distributed positives in a 2 minutes period	Low
Multiple Series of 5 Successive Positives	Medium
Multiple Series of 10 Successive Positives	High

Table 9. Detection of Full SPIT in Four Mixed Traces With Different Intensities

# of Concurrent calls	1	10	50	100
RBF; C= 1; $\gamma = 1/8$; Training accuracy = 98.9057				
True Positives	0.03	3.05	12.18	23.41
	2/7015	15/492	85/698	184/786
True Negatives	100			

In the full SPIT experiment, we request the hundred bots to register with the proxy. Spitter hits all the bots in four successive campaigns with increasing intensity. Results are slightly better than in the partial SPIT experiment (Table 9). Partial SPIT generates an abnormal traffic at the same level as full SPIT does.

7 Related Works

VoIP security is a recent research domain that emerged over the last few years with the increasing use of this technology by enterprises and individuals. Combating SPIT and DoS is the subject of many research proceedings. Quittek et al. [10] apply hidden Turing tests and compare the resulting patterns with typical human communication patterns. Passing these tests causes significant resource consumption in the SPIT generation side. The authors of [11] propose a call rank mechanism based on call duration, social networks and global reputation to filter SPIT calls. Other ideas include a progressive and multi (short term -long term) grey level algorithm [12] and incorporating active stack fingerprinting [13].

The authors of [14] design application and transport sensors to protect enterprise networks from VoIP DoS attacks based on previous works on TCP DoS protection and study different recovery algorithms. The authors of [15] modify the original state machine of SIP transactions to detect transaction anomalies and apply different thresholds to detect flooding attacks. More adaptive to such attacks is the work of Sengar et al. [16] where the Hellinger distance between learning and testing periods is used to detect TCP SYN, SIP INVITE and RTP floods. Their approach shows good performances. There have many papers in the community on generic intrusion detection methods [17,18,19] without to extend to the fine tuned session, dialog, transaction related parameters found in SIP. Over the past, many security related applications have leveraged machine learning techniques and the reader is referred to [20] and [21] for an overview.

The closest work to ours is the study of [22] where the authors have presented a traffic behavior profiling methodology and demonstrated its applications in problem diagnosis and anomaly detection. Our work is more oriented towards attack detection and classification rather than proposing a global and multi level profiling methodology. We have addressed the VoIP specific event correlation and honeypots in previous published work [23] and [24], which did not cover SIP-level monitoring.

8 Conclusion and Future Works

As attacks on VoIP are popping-up in different forms with increasing impact on both the users and infrastructure, more monitoring and security management is needed. In this paper, we proposed an online monitoring methodology based on support vector machines. Our idea is to cut the ongoing signalling (SIP) traffic into small slices and to extract a vector of defined features characterizing each slice. Vectors are then pushed into a SVM for classification based on a learning model. We then use a deterministic event correlator to raise an alarm when suspicious and abnormal situations occur.

We validated our approach by offline tests over a set of real world traces and attacks which are generated in our customized testbed and inserted in the normal traffic traces. Results showed a real time performance and a high accuracy of detecting flooding and SPIT attacks especially when coupled with efficient event correlation rules. Detection of other types of attacks are future work.

Unsupervised learning techniques are appealing because they don't need a priori knowledge of the traffic and can detect new and previously unknown attacks. We consider currently to redefine and reorder our set of features based on different features selection algorithms. We will extend the current event correlation and filtering algorithm in order to reveal attack strategies and improve intrusion prevention/detection accuracy.

Acknowledgment. We would like to thank Mr Dorgham Sisalem and Mr. Sven Ehlert, both from Fraunhofer Institute in Berlin for their comments and feedback on discussing the analysis of SIP traces.

References

1. VoIPSA: VoIP security and privacy threat taxonomy. Public Realease 1.0 (October 2005), http://www.voipsa.org/Activities/VOIPSA_Threat_Taxonomy_0.1.pdf
2. Endler, D., Collier, M.: Hacking Exposed VoIP: Voice Over IP Security Secrets and Solutions. McGraw-Hill Professional Publishing, New York (2007)
3. Vapnik, V.N.: The nature of statistical learning theory. Springer, New York (1995)
4. Vapnik, V.: Statistical Learning Theory, New York (1998)
5. Guyon, I., Weston, J., Barnhill, S., Vapnik, V.: Gene selection for cancer classification using support vector machines. Mach. Learn. 46(1-3), 389–422 (2002)
6. Romano, R.A., Aragon, C.R., Ding, C.: Supernova recognition using support vector machines. In: ICMLA 2006: Proceedings of the 5th International Conference on Machine Learning and Applications, Washington, DC, USA, pp. 77–82. IEEE Computer Society, Los Alamitos (2006)
7. Mukkamala, S., Janoski, G., Sung, A.: Intrusion detection: Support vector machines and neural networks. The IEEE Computer Society Student Magazine 10(2) (2002)
8. Chang, C.C., Lin, C.J.: LIBSVM: a library for support vector machines (2001), http://www.csie.ntu.edu.tw/~cjlin/libsvm
9. Abdelnur, H.J., State, R., Festor, O.: KiF: a stateful SIP fuzzer. In: IPTComm 2007: Proceedings of the 1st international conference on Principles, systems and applications of IP telecommunications, pp. 47–56. ACM, New York (2007)
10. Quittek, J., Niccolini, S., Tartarelli, S., Stiemerling, M., Brunner, M., Ewald, T.: Detecting SPIT calls by checking communication patterns. In: IEEE International Conference on Communications (ICC 2007) (June 2007)
11. Balasubramaniyan, V.A., Ahamad, M., Park, H.: CallRank: Combating SPIT using call duration, social networks and global reputation. In: Fourth Conference on Email and Anti-Spam (CEAS 2007). Mountain View, California (2007)
12. Shin, D., Shim, C.: Progressive multi gray-leveling: A voice Spam protection algorithm. IEEE Network 20
13. Yan, H., Sripanidkulchai, K., Zhang, H., Shae, Z.Y., Saha, D.: Incorporating active fingerprinting into SPIT prevention systems. In: Third annual security workshop (VSW 2006), June 2006, ACM Press, New York (2006)
14. Reynolds, B., Ghosal, D.: Secure IP Telephony using Multi-layered Protection. In: Proceedings of The 10th Annual Network and Distributed System Security Symposium, San Diego, CA, USA (February 2003)
15. Chen, E.: Detecting DoS attacks on SIP systems. In: Proceedings of 1st IEEE Workshop on VoIP Management and Security, San Diego, CA, USA, April 2006, pp. 53–58 (2006)
16. Sengar, H., Wang, H., Wijesekera, D., Jajodia, S.: Detecting VoIP Floods using the Hellinger Distance. Transactions on Parallel and Distributed Systems (acepted for future publication, September 2007)

17. Valdes, A., Skinner, K.: Adaptive, model-based monitoring for cyber attack detection. In: Debar, H., Mé, L., Wu, S.F. (eds.) RAID 2000. LNCS, vol. 1907, pp. 80–92. Springer, Heidelberg (2000)
18. Denning, D.E.: An intrusion-detection model. In: IEEE Symposium on Security and Privacy, April 1986, pp. 118–133. IEEE Computer Society Press, Los Alamitos (1986)
19. Krügel, C., Toth, T., Kirda, E.: Service specific anomaly detection for network intrusion detection. In: SAC 2002: Proceedings of the 2002 ACM symposium on Applied computing, pp. 201–208. ACM Press, New York (2002)
20. Ning, P., Jajodia, S.: Intrusion Detection in Distributed Systems: An Abstraction-Based Approach. Springer, Heidelberg (2003)
21. Maloof, M.: Machine Learning and Data Mining for Computer Security: Methods and Applications. Springer, Heidelberg (2005)
22. Kang, H.J., Zhang, Z.L., Ranjan, S., Nucci, A.: Sip-based voip traffic behavior profiling and its applications. In: MineNet 2007: Proceedings of the 3rd annual ACM workshop on Mining network data, pp. 39–44. ACM, New York (2007)
23. Nassar, M., State, R., Festor, O.: Intrusion detections mechanisms for VoIP applications. In: Third annual security workshop (VSW 2006), June 2006. ACM Press, New York (2006)
24. Nassar, M., State, R., Festor, O.: VoIP honeypot architecture. In: Proc. of 10 th. IEEE/IFIP Symposium on Integrated Management. (June 2007)

Table 10. Appendix: List of features

Number	Name	Description
Group 1 - General Statistics		
1	Duration	Total time of the slice
2	NbReq	# of requests / Total # of messages
3	NbResp	# of responses / Total # of messages
4	NbSdp	# of messages carrying SDP / Total # of messages
5	AvInterReq	Average inter arrival of requests
6	AvInterResp	Average inter arrival of responses
7	AvInterSdp	Average inter arrival of messages carrying SDP bodies
Group 2 - Call-ID Based Statistics		
8	NbSess	# of different Call-IDs
9	AvDuration	Average duration of a Call-ID
10	NbSenders	# of different senders / Total # of Call-IDs
11	NbReceivers	# of different receivers / Total # of Call-IDs
12	AvMsg	Average # of messages per Call-ID
Group 3 - Dialogs Final State Distribution		
13	NbNOTACALL	# of NOTACALL/ Total # of Call-ID
14	NbCALLSET	# of CALLSET/ Total # of Call-ID
15	NbCANCELED	# of CANCELED/ Total # of Call-ID
16	NbREJECTED	# of REJECTED/ Total # of Call-ID
17	NbINCALL	# of INCALL/ Total # of Call-ID
18	NbCOMPLETED	# of COMPLETED/ Total # of Call-ID
19	NbRESIDUE	# of RESIDUE/ Total # of Call-ID
Group 4 - Requests Distribution		
20	NbInv	# of INVITE / Total # of requests
21	NbReg	# of REGISTER/ Total # of requests
22	NbBye	# of BYE/ Total # of requests
23	NbAck	# of ACK/ Total # of requests
24	NbCan	# of CANCEL/ Total # of requests
25	NbOpt	# of OPTIONS / Total # of requests
26	Nb Ref	# of REFER/ Total # of requests
27	NbSub	# of SUBSCRIBE/ Total # of requests
28	NbNot	# of NOTIFY/ Total # of requests
29	NbMes	# of MESSAGE/ Total # of requests
30	NbInf	# of INFO/ Total # of requests
31	NbPra	# of PRACK/ Total # of requests
32	NbUpd	# of UPDATE/ Total # of requests
Group 5 - Responses Distribution		
33	Nb1xx	# of Informational responses / Total # of responses
34	Nb2xx	# of Success responses / Total # of responses
35	Nb3xx	# of Redirection responses / Total # of responses
36	Nb4xx	# of Client error responses / Total # of responses
37	Nb5xx	# of Server error responses / Total # of responses
38	Nb6xx	# of Global error responses / Total # of responses

The Effect of Clock Resolution on Keystroke Dynamics

Kevin Killourhy and Roy Maxion

Dependable Systems Laboratory
Carnegie Mellon University
5000 Forbes Ave,
Pittsburgh PA, 15213
{ksk,maxion}@cs.cmu.edu

Abstract. Keystroke dynamics—the analysis of individuals' distinctive typing rhythms—has been proposed as a biometric to discriminate legitimate users from impostors (whether insiders or external attackers). Anomaly detectors have reportedly performed well at this discrimination task, but there is room for improvement. Detector performance might be constrained by the widespread use of comparatively low-resolution clocks (typically 10–15 milliseconds).

This paper investigates the effect of clock resolution on detector performance. Using a high-resolution clock, we collected keystroke timestamps from 51 subjects typing 400 passwords each. We derived the timestamps that would have been generated by lower-resolution clocks. Using these data, we evaluated three types of detectors from the keystroke-dynamics literature, finding that detector performance is slightly worse at typical clock resolutions than at higher ones (e.g., a 4.2% increase in equal-error rate). None of the detectors achieved a practically useful level of performance, but we suggest opportunities for progress through additional, controlled experimentation.

Keywords: Anomaly detection; Insider-attack detection; Keystroke dynamics; Digital biometrics.

1 Introduction

Compromised passwords, shared accounts, and backdoors are exploited both by external attackers and insiders. Lists of default passwords and password-cracking programs are a staple in the toolbox of external attackers. In a study of insider attacks (i.e., those conducted by people with legitimate access to an organization), Keeney et al. [11] found that the majority of insiders exploited shared or compromised passwords, as well as backdoor accounts. However, if we had some sort of "digital fingerprint" with which to identify exactly who is logging into an account, and to discriminate between the *legitimate user* of an account and an *impostor*, we could significantly curb the threats represented by both insiders and external attackers. Of the various potential solutions to this problem, one technique that has been popular within the research community is keystroke dynamics—the analysis of individual typing rhythms for use as a biometric identifier. Compared to other biometric data, typing times are relatively easy to collect. When a user logs into a computer by typing his or her password, the program authenticating the user could save not just the characters of the password, but the time at which each key was pressed and released. One could imagine a keystroke-dynamics detection algorithm that analyzes these typing times, compares them to a known profile of the legitimate user of the account, and makes a decision about whether or not the new typist is an impostor. In fact, detectors have been designed to use typing rhythms as a biometric, not just during password entry (which is our focus in this work), but also for free-text typing [17].

In terms of accuracy, the European standard for access-control systems (EN-50133-1) specifies a false-alarm rate of less than 1%, with a miss rate of no more than 0.001% [3]. In other words, in order for a keystroke-dynamics detector to be practical, it must correctly identify a legitimate user 99% of the time, and it must correctly identify an impostor 99.999% of the time. At this point, no proposed detector has obtained such numbers in repeated evaluations. When a detector comes up short in evaluation, the common strategy is to go back to the drawing board and try a new detector. However, it may be possible to boost the performance of an existing detector by giving it better data.

Imagine the effect that timing noise might have on a detector. With enough noise, subtle differences between typists will be masked, and even a good detector will be ineffective. One obvious source of noise comes from the resolution of the clock supplying timestamps for each keystroke. For instance, our testing shows that the clock used by Microsoft Windows XP to timestamp keystroke-event messages [15] (which we call the *Windows-event clock*) has a resolution of 15.625 milliseconds (ms), corresponding to 64 updates per second. Figure 1 shows how the clock resolution affects the calculation of keydown–keydown digram latencies. Specifically, if every time reported by the clock is a multiple of 15.625 ms (truncated to the nearest millisecond), then all latencies will appear to fall in bands separated by 15 ms. The calculated latencies could differ from the true latencies by as much as the resolution of the clock (approximately ±15 ms). If two typists differ in their typing times by less than 15 ms, then the difference could be lost. This investigation empirically measures the effect of clock resolution on the performance of a detector. Specifically, we look at whether the performance of a detector can be boosted by increasing the resolution of the clock, and whether or not detectors are robust to low-resolution clocks.

2 Background and Related Work

Detectors for discriminating between users' and impostors' keystroke dynamics have been investigated for over 30 years. They were first considered in 1977 by Forsen et al. [6], who distinguished a legitimate user from an impostor on the basis of how each one typed the user's name. In 1980, Gaines et al. [7] compared

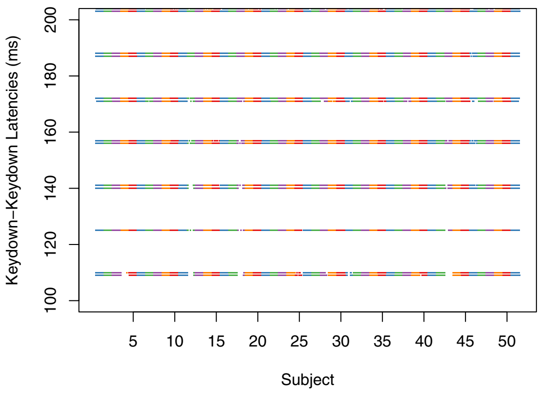


Fig. 1. The spacing between each horizontal band of keystroke latencies reveals that the Windows-event clock has a resolution of 15.625 milliseconds. Any fine-grained differences between the subjects are masked when all latencies are coarsened to a nearby multiple of the clock resolution. Data are keydown–keydown digram latencies (between 100 and 200 ms) recorded by the 15.625 ms resolution clock when each of 51 subjects typed a password 400 times. Double bands occur because Windows reports the timestamps as a whole number of milliseconds, sometimes rounding up and sometimes rounding down.

several users' keystrokes on a transcription task. Both studies presented positive findings, but cautioned that their results were only preliminary.

Joyce and Gupta [10] were some of the earliest researchers to study the keystroke dynamics of passwords. They developed a detector that compared typing characteristics of a new presentation of a password against the average (mean) typing characteristics of the legitimate user. Cho et al. [4] developed and compared two new detectors, inspired by techniques from machine learning. One was based on the nearest-neighbor algorithm, and the other used a multilayer perceptron. A full survey of keystroke-dynamics detectors has been conducted by Peacock et al. [17], but we focus here on the work of Joyce and Gupta, and Cho et al. Their work shows a diversity among available detection techniques, and our investigation uses detectors similar to theirs.

In terms of timing considerations, Forsen et al. and Gaines et al. collected data on a PDP-11. Forsen et al. reported times in 5-millisecond intervals, while Gaines et al. reported millisecond accuracy. Both Joyce and Gupta, and Cho et al. collected data on a Sun workstation. Cho et al. specified that they used X11, which provides keystroke timestamps with a 10 ms resolution. The X11 clock is typically used by researchers on UNIX-based platforms, while Windows users typically use the Windows-event clock (e.g., Sheng et al. [19]). Our testing shows that the timestamps reported through this clock have a 15.625 ms resolution (see Figure 1).

3 Problem and Approach

Keystroke-dynamics detectors—programs designed to distinguish between a legitimate user and an impostor on the basis of typing rhythms—will almost certainly be affected by the resolution of the clock that is used for timing the keystrokes. However, the extent of this effect has never been quantified or measured. In this work, we investigate the effect that clock resolution has on the performance of keystroke-dynamics detectors. We hope to boost detector performance by using better clocks, and to quantify the error introduced by typical clocks.

3.1 Investigative Approach

Our approach is outlined in the following four steps:

1. **Password-data collection:** We choose a password, and we implement a data-collection apparatus that records high-resolution timestamps. We recruit subjects to type the password. We collect keystroke timestamps simultaneously with a high-resolution clock and with a typical low-resolution clock.
2. **Derived clock resolutions:** We coarsen the high-resolution timestamps, in order to calculate the timestamps that would be generated by a range of lower-resolution clocks; we derive password-timing data at a range of resolutions.
3. **Detector implementation:** We develop three types of keystroke-dynamics detectors similar to those reported in the literature: a mean-based detector, a nearest-neighbor detector, and a multilayer perceptron.
4. **Performance-assessment method:** We construct evaluation data sets from our password-timing data, and we use them to measure the performance of the three detectors. We verify the correctness of our derivations (in step 2) by comparing a detector's performance on derived low-resolution data to its performance on data from a real clock operating at that resolution. Finally, we examine how the performance changes as a function of clock resolution.

In the end, we are able to quantify the effect that clock resolution has on several diverse detectors. We show a small but significant improvement from using high-resolution clocks. We describe the four steps of our investigation in Sections 4–7.

3.2 Controlling for Potential Confounding Factors

Our approach departs from typical keystroke-dynamics evaluations, where realism is considered to have higher importance than control. A reason for designing a controlled experiment is to remove *confounding factors*—variables that may distort the effect of the variable of interest on the experimental outcome [5].

In our investigation, the variable of interest is the clock resolution, and the experimental outcome is the performance of a detector. The clock might affect detector performance because it subtly changes the keystroke times analyzed by the detectors. All other factors that change these keystroke times are potential confounding factors that might obscure or distort this effect. They might change a detector's performance, or even change *how clock resolution affects the detector's performance*. The presence of such a factor would compromise our investigation by offering an alternative explanation for our results.

Ideally, we would test all potential confounding factors, to see whether they actually do confound the experiment. However, to do so would require an exponential amount of data (in the number of factors). Practically, we control for potential confounding factors by keeping them constant.

4 Password-Data Collection

The first step in our investigation was to collect a sample of keystroke-timing data using a high-resolution clock. We chose a single password to use as a typing sample. Then we designed a data-collection apparatus for collecting subjects' keystrokes and timestamps. Finally, we recruited 51 subjects, and collected the timing information for 400 passwords from each one (over 8 sessions).

4.1 Choosing a Password

Password selection is the first potential confounding factor we identified. Some passwords can be typed more quickly than others. The choice of password may affect a subject's keystroke times, distorting the effect of clock resolution. To control for the potential confounding factor, we chose a single fixed but representative password to use throughout the experiment.

To make the password representative of a typical, strong password, we employed a publicly available password generator [21] and password-strength checker [13]. We generated a 10-character password containing letters, numbers, and punctuation and then modified it slightly, interchanging some punctuation and casing to better conform with the general perception of a strong password. The result of this procedure was the following password:

.tie5Roanl

The password-strength checker rates this password as strong because it contains at least 8 characters, a capital letter, a number, and punctuation. The best rating is reserved for passwords with at least 14 characters, but we decided to maintain a 10-character limit on our password so as not to exhaust our subjects' patience. (Other researchers used passwords as short as 7 characters [4].)

4.2 Data-Collection Apparatus

We wrote a Windows application that prompts a subject to type the password 50 times. Of course, in the real world, users do not type their password 50 times in a row; they might only type it a few times each day. However, the amount of practice a subject has at typing a particular password represents another potential confounding factor (see Section 3.2). Practiced typists are usually faster, and the amount of practice a subject has may affect his or her keystroke times.

By having our subjects type the password in fixed-length sessions, we controlled how much (and under what circumstances) our subjects became practiced at typing the password.

The application displays the password in a screen along with a text-entry field. In order to advance to the next screen, the subject must type the 10 characters of the password correctly in sequence and then type Return. If the subject makes a mistake, the application immediately detects the error, clears the text-entry field, and after a short pause, it prompts the subject to type the password again. For instance, if a subject typed the first three characters of the password correctly (.ti) but mistyped the fourth (w instead of e), the application would make the subject type the whole password over again. In this way, we ensure that the subject correctly types the entire password as a sequence of exactly 11 keystrokes (corresponding to the 10 characters of the password and the Return key). Forcing subjects to type the password without error is a typical constraint when analyzing keystroke dynamics [4, 19].

When a subject presses or releases a key, the application records the event (i.e., whether a key was pressed or released, and what key was involved), and also the time at which the event occurred. Two timestamps are recorded: one is the timestamp reported by the 15.625 ms resolution Windows-event clock; the other is the timestamp reported by a high-resolution external reference clock. The resolution of the reference clock was measured to be 200 microseconds by using a function generator to simulate key presses at fixed intervals. This clock reported the timestamps accurately to within ±200 microseconds. We used an external reference instead of the high-precision performance counter available through Windows [16] because of concerns that factors such as system load might decrease the accuracy of the timestamps.

The data-collection application was installed on a single laptop with no network connection and with an external keyboard. We identified keyboard selection as another potential confounding factor (see Section 3.2). If subjects used different keyboards, the difference might affect their keystroke times. We control for the potential confounding factor by using one keyboard throughout the experiment.

4.3 Running Subjects

We recruited 51 subjects, many from within the Carnegie Mellon Computer Science Department, but some from the university at large. We required that subjects wait at least 24 hours between each of their 8 sessions, so each session was recorded on a separate day (ensuring that some day-to-day variation existed within our sample). All 51 subjects remained in the study, contributing 400 passwords over the 8 sessions.

Our sample of subjects consisted of 30 males and 21 females. We had 8 left-handed and 43 right-handed subjects. We grouped ages by 10-year intervals. The median group was 31–40, the youngest group was 11–20, and the oldest group was 61–70. The subjects' sessions took between 1.25 minutes and 11 minutes, with the median session taking 3 minutes. Subjects took between 9 days and 35

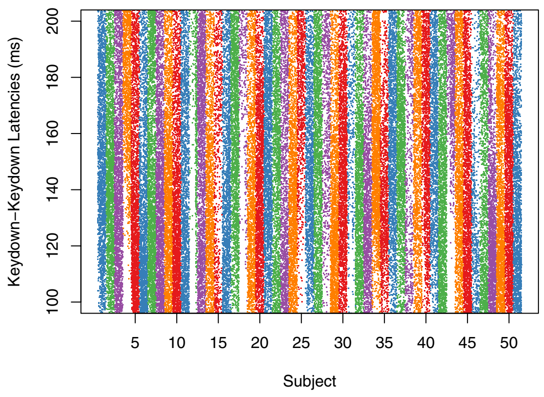


Fig. 2. The absence of horizontal bands demonstrates that the high-resolution clock has a resolution of less than 1 millisecond (200 microseconds, specifically). The keystrokes are the same as in Figure 1, but the latencies in this figure are based on the high-resolution clock.

days to complete all 8 sessions. The median length of time between the first and last session was 23 days.

5 Derived Clock Resolutions

The second step in our investigation was to use the high-resolution data to reconstruct the data that would have been collected with lower-resolution clocks. We developed a procedure to derive the timestamp of a low-resolution clock from the corresponding timestamp of a high-resolution clock.

First, we examine the keydown–keydown latencies based on the high-resolution timestamps. The latencies are shown in Figure 2. Compare these latencies to the equivalent latencies from Figure 1. Whereas the horizontal bands in Figure 1 reveal that the Windows-event clock cannot capture any timing variation smaller than 15.625 milliseconds, the absence of such bands in Figure 2 demonstrates that very subtle variations (smaller than 1 millisecond) can be captured by the high-resolution clock.

Next, to determine what would have happened if the data had been collected with a lower-resolution clock, we need to artificially decrease the resolution of this clock. Consider how timestamps are normally assigned to keystroke events:

1. The operating system is notified of the pending key event by an interrupt from the keyboard controller.
2. The operating system reads the key event from the keyboard device into memory.
3. During the handling of the key event, the operating system queries a clock for the current time.
4. The timestamp returned by the clock is included in the description of the keystroke event and is delivered to any applications waiting on the event.

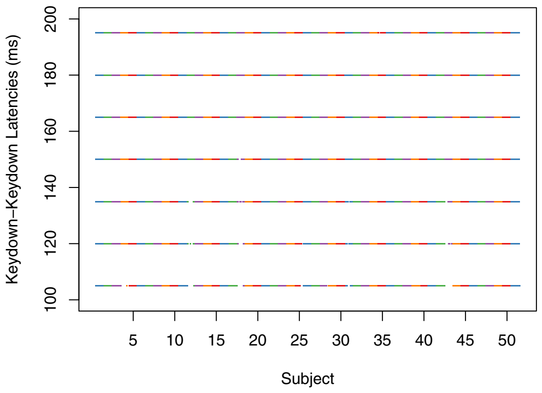


Fig. 3. The presence of horizontal bands 15 ms apart suggests that the derived 15 ms clock exhibits the same behavior as a real clock with a 15 ms resolution. The keystrokes are the same as in Figures 1 and 2, but the derived 15 ms clock was used to calculate the latencies. The bands resemble those of the real 15.625 ms clock in Figure 1, but without double bands because the 15 ms clock resolution has no fractional part being rounded to a whole millisecond.

For example, if we have a clock with a resolution of 15 ms (i.e., it is updated every 15 ms), then the timestamp returned by the clock will be divisible by 15 ms. Specifically, it will be the largest multiple of 15 ms smaller than the actual time at which the clock was queried. In general, if the clock was queried at time $t_{\text{hi-res}}$, and we want to reproduce the behavior of a lower-resolution clock (with a resolution of r), the low-resolution timestamp would be

$$t_{\text{lo-res}} \leftarrow \lfloor t_{\text{hi-res}}/r \rfloor \times r$$

where $\lfloor x \rfloor$ is the largest integer smaller than x (floor function).

Finally, with this formula and the high-resolution data, we can derive the timestamps that would have been collected with lower-resolution clocks. For instance, Figure 3 shows keystroke latencies calculated from a clock with a derived 15 ms resolution. Note the similarity to Figure 1, which shows latencies calculated from a real Windows-event clock with a 15.625 ms resolution. (The fractional part of the real clock's resolution accounts for the slight differences.)

One limitation of this procedure is that we can only derive clock resolutions that are evenly divisible by that of our high-resolution clock. This criteria allows the small but non-zero inaccuracy of our high-resolution clock to be absorbed into the inaccuracy of the lower-resolution clock. For instance, we should be able to accurately derive a 1 ms clock resolution since 1 ms is evenly divisible by 200 microseconds (the resolution of the high-resolution clock). However, we could not accurately derive a 1.5 ms clock resolution (or a 15.625 ms resolution) because it is not evenly divided. Regardless of this limitation, the accuracy of results obtained with these derived clock resolutions will be established by comparing detector performance on derived 15 ms resolution data to that on the 15.625 ms resolution Windows-event clock data. We derive data at the following 20 clock resolutions:

Milliseconds: 1 2 5 10 15 20 30 50 75 100 150 200 500 750
Seconds: 1 2 5 10 15 30

The specific resolutions were chosen arbitrarily, but with the intent of including a range of typical values (on the first line), and a range of extremely low-resolution values (on the second line) in order to identify the point at which detector performance degrades completely. In total, we have data at 22 different clock resolutions: the 20 derived clocks, the high-resolution clock, and the 15.625 ms resolution Windows-event clock.

6 Detector Implementation

The third step in our investigation was to create detectors to test using our data. We identified three different types of detector from the literature, and implemented a detector of each type:

1. a mean-based detector,
2. a nearest-neighbor detector, and
3. a multilayer-perceptron detector.

By ensuring that we have diversity in the set of detectors we evaluate, we can examine whether or not an observed effect is specific to one type of detector or more generally true for a range of detectors.

6.1 Detector Overview

We constrained our attention to detectors that behave similarly in terms of their input and output. For instance, each of our detectors must analyze password-timing data, and aims to discriminate between a legitimate user and an impostor. Each of the detectors expects the password data to be encoded in what is called a *password-timing vector*. A password-timing vector is a vector of hold times and intervals. A hold time is the difference between the key-press timestamp and the key-release timestamp for the same key. An interval time is the (signed) difference between the key-release timestamp of the first key in a digram and the key-press timestamp of the second key.

The password-timing vector is 21 elements long for the password we chose (.tie5Roanl). Each element is either a hold time for one of the 11 keys in the password (including the Return key), or the interval between one of the 10 digrams, arranged as follows:

Index	Element name
1	Hold(period)
2	Interval(period-t)
3	Hold(t)
4	Interval(t-i)
5	Hold(i)
⋮	⋮
19	Hold(l)
20	Interval(l-Return)
21	Hold(Return)

where Hold(period) is the hold time of the period key, and Interval(period-t) is the interval between the period key-release and the t key-press.

Each detector has two phases: training and testing. During training, a set of password vectors from a legitimate user is used to build a *profile* of that user. Different detectors build this profile in different ways, but the objective of a successful detector is to build a profile that uniquely distinguishes the user from all other typists (like a fingerprint). During testing, a new password-timing vector (from an unknown typist) is provided, and the detector compares the new vector against the profile. The detector produces an *anomaly score* that indicates whether the way the new password was typed is similar to the profile (low score) or different from the profile (high score). The procedure by which this score is calculated depends on the detector.

In practice, the anomaly score would be compared against some pre-determined threshold to decide whether or not to raise an alarm (i.e., whether or not the password-typing rhythms belong to an impostor). However, in our evaluation, we will use these scores directly to assess the detector's performance.

The three detectors are implemented using the R statistical programming environment (version 2.4.0) [18]. The nearest-neighbor detector leverages an implementation of Bentley's *kd*-trees [1] by Mount and Arya [14]. The multilayer perceptron uses the neural-network package AMORE [12].

6.2 Mean-Based Detector

A mean-based detector models a user's password-timing vectors as coming from some known distribution (e.g., a multidimensional normal distribution) with an unknown mean. During training, the mean is estimated, and during testing, a new password-timing vector is assigned an anomaly score based on its distance from this mean. Joyce and Gupta [10] used a detector that fits this description, and the detector we implemented is similar to theirs, but not precisely the same.[1]

During training, our mean-based detector estimates the mean vector and the covariance matrix of the training password-timing vectors. The mean vector is a 21-element vector, whose first element is the mean of the first elements of the training vectors, whose second element is the mean of the second elements of the training vectors, and so on. Similarly, the covariance matrix is the 21-by-21-element matrix containing the covariance of each pair of elements in the 21-element training vectors. These mean and covariance estimates comprise the user's profile.

During testing, the detector estimates the *Mahalanobis distance* of the new password-timing vector from the mean vector of the training data. The

[1] Our detector differs from that proposed by Joyce and Gupta in both its mean-vector calculation and the distance measure used. We calculated the mean vector using all the training data while Joyce and Gupta preprocessed the data to remove outliers. We used the Mahalanobis distance while Joyce and Gupta used the Manhattan distance. Our mean-based detector was intended to be simple (with no preprocessing) while still accommodating natural variances in the data (with the Mahalanobis distance).

Mahalanobis distance is a measure of multidimensional distance that takes into account the fact that a sample may vary more in one dimension than another, and that there may be correlations between pairs of dimensions. These variations and correlations are estimated using the correlation matrix of the training data. More formally, using the matrix notation of linear algebra, if **x** is the mean of the training data, **S** is the covariance matrix, and **y** is the new password-timing vector, the Mahalanobis distance (d) is:

$$d \leftarrow (\mathbf{x} - \mathbf{y})^{\mathsf{T}} \mathbf{S}^{-1} (\mathbf{x} - \mathbf{y})$$

The anomaly score of a new password-timing vector is simply this distance.

6.3 Nearest-Neighbor Detector

Whereas the mean-based detector makes the assumption that the distribution of a user's passwords is known, the nearest-neighbor detector makes no such assumption. Its primary assumption is that new password-timing vectors from the user will resemble one or more of those in the training data. Cho et al. [4] explored the use of a nearest-neighbor detector in their work, and we attempted to re-implement their detector for our investigation.

During training, the nearest-neighbor detector estimates the covariance matrix of the training password-timing vectors (in the same way as the mean-based detector). However, instead of estimating the mean of the training data, the nearest-neighbor detector simply saves each password-timing vector.

During testing, the nearest-neighbor detector calculates Mahalanobis distances (using the covariance matrix of the training data). However, instead of calculating the distance from the new password-timing vector to the mean of the training data, the distance is calculated from the new password-timing vector to each of the vectors in the training data. The distance from the new vector to the nearest vector from the training data (i.e., its nearest neighbor) is used as the anomaly score.

6.4 Multilayer-Perceptron Detector

Whereas the behaviors of the mean-based and nearest-neighbor detectors allow for an intuitive explanation, the multilayer perceptron is comparatively opaque. A multilayer perceptron is a kind of artificial neural network that can be trained to behave like an arbitrary function (i.e., when given inputs, its outputs will approximate the function's output). Hwang and Cho [8] showed how a multilayer perceptron could be used as an anomaly detector by training it to *auto-associate*—that is, to behave like a function that reproduces its input as the output. In theory, new input that is like the input used to train the network will also produce similar output, while input that is different from the training input will produce wildly different output. By comparing the input to the output, one can detect anomalies. Cho et al. [4] used an auto-associative multilayer perceptron to discriminate between users and impostors on the basis of password-timing vectors. We attempted to re-implement that detector.

During training, the password-timing vectors are used to create an auto-associative multilayer perceptron. This process is a standard machine-learning procedure, but it is fairly involved. We present an overview here, but we must direct a reader to the works by Hwang, Cho, and their colleagues for a comprehensive treatment [4, 8]. A skeleton of a multilayer perceptron is first created. The skeleton has 21 input nodes, corresponding to the 21 elements of the password-timing vector, and 21 output nodes. In general, a multilayer-perceptron network can have a variety of structures (called hidden nodes) between the input and the output nodes. In keeping with earlier designs, we had a single layer of 21 hidden nodes. This skeleton was trained using a technique called back-propagation to auto-associate the user's password-timing vectors. We used the recommended learning parameters: training for 500 epochs with a 1×10^{-4} learning rate and a 3×10^{-4} momentum term.[2]

During testing, the new password-timing vector is used as input to the trained multilayer perceptron, and the output is calculated. The Euclidean distance of the input to the output is computed and used as the anomaly score.

7 Performance-Assessment Method

Now that we have three detectors and data at a variety of clock resolutions, the final step is to evaluate the detectors' performance. First, we convert the data to password-timing tables. Then we devise a procedure for training and testing the detectors. Last, we aggregate the test results into overall measures of each detector's performance at each clock resolution.

7.1 Creating Password-Timing Tables

As mentioned in Section 5, we have 22 data sets that differ only in the resolution of the clock used to timestamp the keystroke events: the high-resolution clock, the 15.625 ms Windows-event clock, and the 20 derived clocks. For each clock, we have timing information for 51 subjects, each of whom typed the password (.tie5Roanl) 400 times.

We extract password-timing tables from the raw data. Hold times and digram intervals are calculated. We confirm that 50 password-timing vectors are extracted from each one of a subject's 8 sessions, and that a total of 20,400 password-timing vectors are extracted (50 passwords $\times$ 8 sessions $\times$ 51 subjects).

7.2 Training and Testing the Detectors

Consider a scenario in which a user's long-time password has been compromised by an impostor. The user is assumed to be practiced in typing her password,

[2] Note that our learning rate and momentum are 1000 times smaller than those reported by Cho et al. This change accounts for a difference in units between their password-timing vectors and ours. (We record in seconds; they used milliseconds.)

while the impostor is unfamiliar with it (e.g., typing it for the first time). We measure how well each of our three detectors is able to detect the impostor, discriminating the impostor's typing from the user's typing in this scenario.

We start by designating one of our subjects as the legitimate user, and the rest as impostors. We train and test each of the three detectors as follows:

1. We train the detector on the first 200 passwords typed by the legitimate user. The detector builds a profile of that user.
2. We test the ability of the detector to recognize the user herself by generating anomaly scores for the remaining 200 passwords typed by the user. We record these as *user scores*.
3. We test the ability of the detector to recognize impostors by generating anomaly scores for the first 5 passwords typed by each of the 50 impostors. We record these as *impostor scores*.

This process is then repeated, designating each of the other subjects as the legitimate user in turn. After training and testing a detector for each combination of subject, detector, and clock-resolution data set, we have a total of 3,366 sets of user and impostor scores (51 subjects $\times$ 3 detectors $\times$ 22 data sets).

It may seem that 200 passwords is an unrealistically large amount of training data. However, we used 200 passwords to train because we were concerned that fewer passwords might unfairly cause one or more detectors to under-perform (e.g., Cho et al. [4] trained the multilayer perceptron on up to 325 passwords). Likewise, an unpracticed impostor might seem unrealistic. If he knew that his keystroke dynamics would be scrutinized, he might practice first. However, as we argued in Section 4.2, the amount of practice a subject has had represents a potential confounding factor. Consequently, all impostors in our experiment were allowed the same level of practice. Our intuition was that the effect of clock resolution on detector performance might be seen most clearly with unpracticed impostors, and so we used their data (with plans to use practiced impostors' data in future investigations).

7.3 Calculating Detector Performance

To convert these sets of user and impostor scores into aggregate measures of detector performance, we used the scores to generate a graphical summary called an ROC curve [20], an example of which is shown in Figure 4. The hit rate is the frequency with which impostors' passwords generate an alarm (a desirable response), and the false-alarm rate is the frequency with which the legitimate user's passwords generate an alarm (an undesirable response). Whether or not a password generates an alarm depends on how the threshold for the anomaly scores is chosen. Over the continuum of possible thresholds to choose, the ROC curve illustrates how each one would change hit and false-alarm rates. Each point on the curve indicates the hit and false-alarm rates at a particular threshold.

The ROC curve is a common visualization of a detector's performance, and on the basis of the ROC curve, various cost measures can be calculated. Two common measures are the *equal-error rate* and the *zero-miss false-alarm rate*.

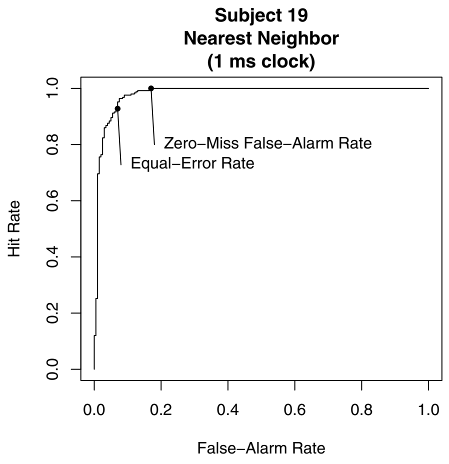


Fig. 4. An example ROC curve depicts the performance of the nearest-neighbor detector with subject 19 as the legitimate user and data from the derived 1 ms resolution clock. The curve shows the trade-off between the hit rate and false-alarm rate. The proximity of the curve to the top-left corner of the graph is a visual measure of performance.

The equal-error rate is the place on the curve where the false-alarm rate is equal to the miss rate (note that miss rate = 1 − hit rate). Geometrically, the equal-error rate is the false-alarm rate where the ROC curve intersects a line from the top-left corner of the plot to the bottom right corner. This cost measure was advocated by Peacock et al. [17] as a desirable single-number summary of detector performance. The zero-miss false-alarm rate is the smallest false-alarm rate for which the miss rate is zero (or, alternatively, the hit rate is 100%). Geometrically, the zero-miss false-alarm rate is the leftmost point on the curve where it is still flat against the top of the plot. This cost measure is used by Cho et al. [4] to compare detectors.

For each combination of subject, detector, and clock resolution, we generated an ROC curve, and we calculated these two cost measures. Then, to obtain an overall summary of a detector's performance at a particular clock resolution, we calculated the average equal-error rate and the average zero-miss false-alarm rate across all 51 subjects. These two measures of average cost were used to assess detector performance.

8 Results and Analysis

A preliminary look at the results reveals that—while the equal-error rate and the zero-miss false-alarm rate differ from one another—they show the same trends with respect to different detectors and clock resolutions. Consequently, we focus on the equal-error-rate results and acknowledge similar findings for the zero-miss false-alarm rate.

Table 1. The average equal-error rates for the three detectors are compared when using (1) the high-resolution clock, (2) the derived 15 ms resolution clock, and (3) the 15.625 ms Windows-event clock. The numbers in parentheses indicate the percent increase in the equal-error rate over that of the high-resolution timer. The results from the 15 ms derived clock very closely match the results with the actual 15.625 ms clock.

	Detectors		
Clock	Mean-based	Nearest Neighbor	Multilayer Perceptron
(1) High-resolution	0.1100	0.0996	0.1624
(2) Derived 15 ms resolution	0.1153 (+4.8%)	0.1071 (+7.5%)	0.1631 (+0.4%)
(3) 15.625 ms Windows-event	0.1152 (+4.7%)	0.1044 (+4.8%)	0.1634 (+0.6%)

The accuracy of our results depends on our derived low-resolution timestamps behaving like real low-resolution timestamps. Our first step is to establish the validity of the derived clock data by comparing a detector's performance on derived low-resolution data to its performance on data from a real clock operating at that resolution. Then we proceed to examine our primary results concerning the effect of clock resolution on detector performance.

8.1 Accuracy of the Derived Clock

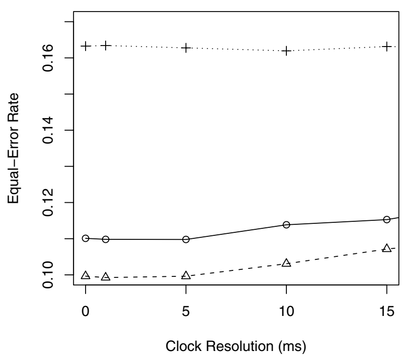
Table 1 shows the average equal-error rate for each of the three detectors, using the high-resolution clock, the derived 15 ms resolution clock, and the real 15.625 ms resolution Windows-event clock. In addition to the equal-error rates, the table includes a percentage in parentheses for the derived clock and the Windows-event clock. This percentage indicates the percent increase in the equal-error rate over that from the high-resolution clock.

To verify the correctness of the results using the derived low-resolution clocks, we compare the second and third rows of Table 1. The results are almost exactly the same except for the nearest-neighbor detector. Since the nearest-neighbor detector is not robust to small changes in the training data, it is not surprising to see a comparatively large difference between the derived 15 ms clock and the real 15.625 ms clock. The similarity in the results of the other two detectors indicate that the derived clock results are accurate.

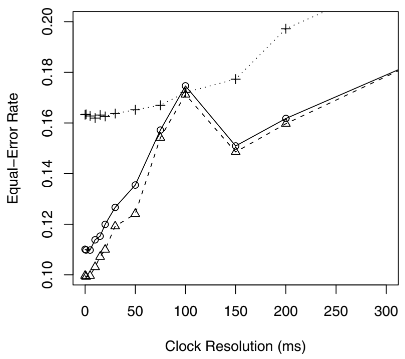
Even if we had been able to directly derive a 15.625 ms clock (impossible because of the limitations of the derivation procedure described in Section 5), small differences between the derived and real timestamps would still cause small differences in detector performance (e.g., differences resulting from small delays in how quickly the real clock is queried).

8.2 Effects of Clock Resolution on Detector Performance

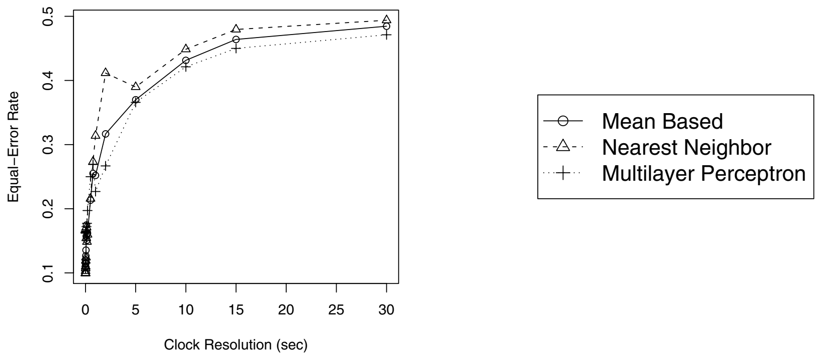
Figure 5 depicts the effect of clock resolution on the average equal-error rate of the three detectors. Each panel displays a curve for each of the three detectors, but at different scales, highlighting a different result.



(a) Resolutions of 0–15 ms



(b) Resolutions of 0–300 ms



(c) All resolutions (0–30 sec)

Fig. 5. The equal-error rates of the three detectors increase as clock-resolution goes from fine to coarse. Panel (a) depicts the minor but significant change in performance resulting from a transition from the high-resolution clock to typical 15 ms clocks. Panel (b) shows how the error jumps significantly when the clock resolution is between 50 ms and 300 ms. Panel (c) characterizes the variation in detector performance over the full range of derived clock resolutions from 1 ms to 30 seconds (where the detector does no better than randomly guessing).

Panel (a) shows the effect of clock resolutions in the range of 0–15 ms on the equal-error rate. These are resolutions that we see in practice (e.g., in Windows and X11 event timestamps). We observe some increase in the equal-error rate for the mean-based and nearest-neighbor detectors, even from the 1 ms clock to the 15 ms clock. The change from the 1 ms clock to the 15 ms clock does not seem to have much effect on the multilayer perceptron (which could be because that detector's performance is already comparatively poor, rather than because the multilayer perceptron is more robust to lower-resolution clocks). The parenthetical percentages in Table 1 quantify the change from high resolution to typical resolutions. When the detectors use the 15 ms clock, their equal-error rate is an average of 4.2% higher than with the high-resolution clock. While this loss may not seem significant, keystroke dynamics needs near-perfect accuracy to be practical (1% false-alarm rate and 0.001% miss rate according to the European standard for access control [3]), so every possible boost in performance will help.

Panel (b) examines the effect of clock resolution beyond the 15 ms range. The graph reveals that the equal-error rates of the mean-based and nearest-neighbor detectors increase sharply after a resolution of 50 ms, and all three detectors' equal-error rates increase together after a resolution of 150 ms. While such low-resolution clocks are not used for keystroke dynamics, we can consider clock resolution to be one of many factors that might affect a detector. (Other factors include bus contention, system load, and even networking delays.) This panel suggests that these detectors are not particularly robust to noise in the form of low clock resolution. By extrapolation, it suggests that tens of milliseconds of noise from any of these sources (or any combination thereof) could be a problem.

Panel (b) also reveals a peak in the equal-error rate of the mean-based and nearest-neighbor detectors at a resolution of 100 ms. The cause of the peak is not obvious; it could be an artifact of our particular subjects' typing characteristics and would disappear with more or different subjects. More typing data and analysis would be necessary to determine whether such peaks appear consistently for a particular detector and clock resolution, but the existence of a peak does suggest that the effects of factors like clock resolution are not always easy to predict.

Panel (c) demonstrates the effect of very-low-resolution clocks on the equal-error rate of a detector. All three detectors' equal-error rates tend to 0.5, which is the theoretically worst possible equal-error rate (akin to random guessing). That the equal-error rate goes to 0.5 is not surprising, but it is surprising that the equal-error rate converges so slowly to 0.5. With a 1-second resolution, the three detectors all have equal-error rates of about 0.3. While not great, it is certainly better than randomly guessing. It is surprising that key-hold times and digram intervals retain some (weakly) discriminative information even when expressed as a whole number of seconds. It may be that the features being used to discriminate users from impostors are present only because our impostors are unpracticed; they type the password a few seconds more slowly than a practiced user would. It is possible that a curve for practiced impostors would be steeper, more quickly ascending to 0.5 (to be investigated in future work).

9 Discussion

Based on these findings, we take away two messages from this investigation, each of which suggests a trajectory for the future. First, we have demonstrated that clock resolution does have an effect on the performance of keystroke-dynamics detectors, and as a result, we should consider the potential deleterious effects of timing noise. Fortunately, the effect appears to be small for the typical clock resolutions we see in practice, but we do get a small boost in performance by using a high-resolution clock. However, clock-resolution granularity is not the only factor that affects keystroke timestamps. Given these results, it seems almost certain that other forms of noise (e.g., system load) will cause similar problems. In the long term, we should try to eliminate noise from our timestamps, but in the short term we should at least acknowledge and account for its presence by carefully evaluating our timing mechanisms (e.g., by measuring and reporting clock resolution).

Second, even with the high-resolution timestamps, our detectors' performance is less than ideal. The best performance we obtained was a 9.96% equal-error rate for the nearest-neighbor detector, which is a long way from a 1% false-alarm rate and a 0.001% miss rate. We were surprised, since the detectors we used are similar to those that have performed well in the literature (e.g., by Joyce and Gupta [10], and by Cho et al. [4]). However, it would be improper to compare our results directly to those in the literature, because there are significant differences between our experimental method and theirs. The most obvious difference is our control of potential confounding factors (e.g., password selection and practice effect).

We speculate that experimental control is indeed responsible for the poorer performance of our detectors. Furthermore, we advocate the control of potential confounding factors in future experiments. Why? While realistic but uncontrolled experiments can demonstrate that a detector does well (or poorly), controlled experiments are necessary to reveal a causal connection between experimental factors (e.g., password choice or practice) and detector performance. If we are to use keystroke dynamics as a biometric, causal factors must be identified—*why it works* is as important as *whether it works*. For instance, it would be significant to discover that, regardless of other factors, every typist has an immutable, intrinsically identifiable quality to his or her typing. It would also be significant (but unfortunate) to find that a detector's performance depends primarily on the number of times an impostor practiced a password, and that with enough practice, any impostor could pass for a legitimate user.

We intend to conduct a survey of other detectors proposed in the literature to see whether performance remains poor on our data. We also observe that these detection algorithms tend to treat typing data as arbitrary points in a high-dimensional space, ignoring the fact that the data are observations about fingers typing. Perhaps better results can be obtained by building a detector that relies upon a model of user typing (such as those proposed by Card et al. [2] or John [9]).

10 Summary and Conclusion

The goal of this work is to investigate the effect that clock resolution has on the performance of keystroke-dynamics detectors, in part to determine if a high-resolution clock would boost performance. We collected data at a high resolution, and derived data at lower resolutions. We implemented three detectors and evaluated their performances over a range of clock resolutions. We found that a high-resolution clock does provide a slight performance boost, and conversely, clocks with a typical 15 ms resolution increase the equal-error rate by an average of 4.2%. Based on results using very-low-resolution clocks, we found that detectors are not particularly robust to timing noise. Finally, we discovered that none of the detectors achieved a practically useful level of performance, and identified significant opportunities for progress through controlled experimentation.

Acknowledgements

The authors are grateful to Rachel Krishnaswami for her insightful comments and helpful advice, and to Patricia Loring for running the experiments that provided the data for this paper. Fahd Arshad and Rob Reeder were responsible for the instrumentation that presented stimuli to participants. Thanks also to several anonymous reviewers for their comments.

This work was supported by National Science Foundation grant numbers CNS-0430474 and CNS-0716677, and by the Army Research Office through grant number DAAD19-02-1-0389 (Perpetually Available and Secure Information Systems) to Carnegie Mellon University's CyLab. The views and conclusions contained in this document are those of the authors, and should not be interpreted as representing the official policies, either expressed or implied, of any sponsoring institution, the U.S. government, or any other entity.

References

[1] Bentley, J.L.: Multidimensional binary search trees used for associative searching. Communications of the ACM 18(9), 509–517 (1975)

[2] Card, S.K., Moran, T.P., Newell, A.: The keystroke-level model for user performance time with interactive systems. Communications of the ACM 23(7), 396–410 (1980)

[3] CENELEC. European Standard EN 50133-1: Alarm systems. Access control systems for use in security applications. Part 1: System requirements, Standard Number EN 50133-1:1996/A1:2002, Technical Body CLC/TC 79, European Committee for Electrotechnical Standardization (CENELEC) (2002)

[4] Cho, S., Han, C., Han, D.H., Kim, H.-I.: Web-based keystroke dynamics identity verification using neural network. Journal of Organizational Computing and Electronic Commerce 10(4), 295–307 (2000)

[5] Dodge, Y.: Oxford Dictionary of Statistical Terms. Oxford University Press, New York (2003)

[6] Forsen, G., Nelson, M., Staron Jr., R.: Personal attributes authentication techniques. Technical Report RADC-TR-77-333, Rome Air Development Center (October 1977)

[7] Gaines, R.S., Lisowski, W., Press, S.J., Shapiro, N.: Authentication by keystroke timing: Some preliminary results. Technical Report R-2526-NSF, RAND Corporation (May 1980)

[8] Hwang, B., Cho, S.: Characteristics of auto-associative MLP as a novelty detector. In: Proceedings of the IEEE International Joint Conference on Neural Networks, Washington, DC, July 10–16, 1999, vol. 5, pp. 3086–3091 (1999)

[9] John, B.E.: TYPIST: A theory of performance in skilled typing. Human-Computer Interaction 11(4), 321–355 (1996)

[10] Joyce, R., Gupta, G.: Identity authentication based on keystroke latencies. Communications of the ACM 33(2), 168–176 (1990)

[11] Keeney, M., Kowalski, E., Cappelli, D., Moore, A., Shimeall, T., Rogers, S.: Insider threat study: Computer system sabotage in critical infrastructure sectors. Technical report, U.S. Secret Service and CERT Coordination Center/SEI (May 2005), http://www.cert.org/archive/pdf/insidercross051105.pdf

[12] Limas, M.C., Meré, J.O., Gonzáles, E.V., Martinez de Pisón Ascacibar, F.J., Espinoza, A.P., Elias, F.A.: AMORE: A MORE Flexible Neural Network Package (October 2007), http://cran.r-project.org/web/packages/AMORE/index.html

[13] Microsoft. Password checker (2008), http://www.microsoft.com/protect/yourself/password/checker.mspx

[14] Mount, D., Arya, S.: ANN: A Library for Approximate Nearest Neighbor Searching (2006), http://www.cs.umd.edu/~mount/ANN/

[15] Microsoft Developer Network. EVENTMSG structure (2008), http://msdn2.microsoft.com/en-us/library/ms644966(VS.85).aspx

[16] Microsoft Developer Network. QueryPerformanceCounter function (2008), http://msdn2.microsoft.com/en-us/library/ms644904(VS.85).aspx

[17] Peacock, A., Ke, X., Wilkerson, M.: Typing patterns: A key to user identification. IEEE Security and Privacy 2(5), 40–47 (2004)

[18] R Development Core Team. R: A Language and Environment for Statistical Computing. R Foundation for Statistical Computing, Vienna, Austria (2008)

[19] Sheng, Y., Phoha, V., Rovnyak, S.: A parallel decision tree-based method for user authentication based on keystroke patterns. IEEE Transactions on Systems, Man, and Cybernetics 35(4), 826–833 (2005)

[20] Swets, J.A., Pickett, R.M.: Evaluation of Diagnostic Systems: Methods from Signal Detection Theory. Academic Press, New York (1982)

[21] PC Tools. Security guide for windows—random password generator (2008), http://www.pctools.com/guides/password/

A Comparative Evaluation of Anomaly Detectors under Portscan Attacks

Ayesha Binte Ashfaq, Maria Joseph Robert, Asma Mumtaz, Muhammad Qasim Ali, Ali Sajjad, and Syed Ali Khayam

School of Electrical Engineering & Computer Science
National University of Sciences & Technology (NUST)
Rawalpindi, Pakistan
{ayesha.ashfaq,47maria,45asma,mqasim.ali,ali,khayam}@niit.edu.pk

Abstract. Since the seminal 1998/1999 DARPA evaluations of intrusion detection systems, network attacks have evolved considerably. In particular, after the CodeRed worm of 2001, the volume and sophistication of self-propagating malicious code threats have been increasing at an alarming rate. Many anomaly detectors have been proposed, especially in the past few years, to combat these new and emerging network attacks. At this time, it is important to evaluate existing anomaly detectors to determine and learn from their strengths and shortcomings. In this paper, we evaluate the performance of eight prominent network-based anomaly detectors under malicious portscan attacks. These ADSs are evaluated on four criteria: accuracy (ROC curves), scalability (with respect to varying normal and attack traffic rates, and deployment points), complexity (CPU and memory requirements during training and classification,) and detection delay. These criteria are evaluated using two independently collected datasets with complementary strengths. Our results show that a few of the anomaly detectors provide high accuracy on one of the two datasets, but are unable to scale their accuracy across the datasets. Based on our experiments, we identify promising guidelines to improve the accuracy and scalability of existing and future anomaly detectors.

1 Introduction

With an increasing penetration of broadband Internet connectivity and an exponential growth in the worldwide IT infrastructure, individuals and organizations now rely heavily on the Internet for their communication and business needs. While such readily-available network connectivity facilitates operational efficiency and networking, systems connected to the Internet are inherently vulnerable to network attacks. These attacks have been growing in their number and sophistication over the last few years [1]. Malware, botnets, spam, phishing, and denial of service attacks have become continuous and imminent threats for today's networks and hosts [1], [2]. Financial losses due to these attacks are in the orders of billions of dollars[1]. In addition to the short-term revenue losses for

[1] Economic losses to recover from the CodeRed worm alone are estimated at \$2.6 billion [3].

businesses and enterprises, network attacks also compromise information confidentiality/integrity and cause disruption of service, thus resulting in a long-term loss of credibility.

Since the CodeRed worm of 2001, malware attacks have emerged as one of the most prevalent and potent threats to network and host security[2]. Many network-based anomaly detection systems (ADSs) have been proposed in the past few years to detect novel network attacks [4]–[23]. Since malicious portscans are the vehicle used by malware and other automated tools to locate and compromise vulnerable hosts, some of these anomaly detectors are designed specifically for portscan detection [4]–[11], [19], while other detectors are more general-purpose and detect any anomalous traffic trend [12]–[18], [20]. Most of the network-based anomaly detectors model and leverage deep-rooted statistical properties of benign traffic to detect anomalous behavior. A variety of theoretical frameworks–including stochastic, machine learning, information-theoretic and signal processing frameworks–have been used to develop robust models of normal behavior and/or to detect/flag deviations from that model. However, very little effort has been expended into comparative evaluation of these recent ADSs for the portscan detection problem.

In this paper, we evaluate and compare eight prominent network-based anomaly detectors on two public portscan datasets. The objectives of this study are: 1) to quantify and compare the accuracies of these detectors under varying rates of attack and normal traffic and at different points of deployment; 2) to identify promising traffic features and theoretical frameworks for portscan anomaly detection; 3) to investigate the accuracy of contemporary anomaly detectors with respect to their complexity and detection delay; 4) to identify a set of promising portscan detection guidelines that build on the strengths and avoid the weaknesses of the evaluated anomaly detectors; and finally 5) to provide an open-source library of anomaly detection tools that operate on public and labeled datasets, and can be used for repeatable performance benchmarking by future detectors[3].

The anomaly detectors compared in this work were proposed in [4], [7], [8], [12], [15], [18], [20] and [21]. These ADSs are chosen because they employ very different traffic features and theoretical frameworks for anomaly detection. Moreover, most of these detectors are frequently used for performance benchmarking in the intrusion detection research literature [6], [9]–[11], [13], [14], [16], [17], and [19]. Some of these ADSs have been designed for and evaluated at endpoints while others have been tailored towards organization/ISP gateways. Similarly, some detector are designed for portscan detection, while others are general-purpose ADSs. This diversity allows us to determine how much, if any, performance improvement is provided by portscan ADSs over general-purpose ADSs.

[2] From 2006 to 2007, the total number of malicious code attacks reported by Symantec DeepSight^{TM}showed a phenomenal increase of 468% [1].

[3] Background and attack datasets are available at [24] and [25]. ADS implementations are also available at [25].

For performance evaluation of the anomaly detectors, we use two independently-collected datasets with complementary strengths. The first dataset is an enterprise traffic dataset collected at the edge router of the Lawrence Berkeley National Lab (LBNL) [24]. Attack traffic in this dataset mostly comprises high-rate background traffic and low-rate outgoing scans. The second dataset comprises traffic data collected at network endpoints in home, university and office settings. Background traffic rates of these endpoints are relatively low as compared to the LBNL dataset, but the endpoint attack data contains relatively high-rate outgoing scan traffic.

We evaluate these ADSs on four criteria: accuracy, scalability, complexity and detection delay. Accuracy is evaluated by comparing ROC (false alarms per day versus detection rate) characteristics of the ADSs. Scalability is evaluated with respect to different background and attack traffic rates. Since the two datasets used in this study are collected at different network entities and contain attacks with different characteristics, evaluation over these datasets allows us to compare the scalability of the proposed ADSs under varying traffic volumes. Complexity is evaluated in terms of time and memory required during training and classification steps of each ADS. Detection delay is evaluated separately for high- and low-rate attacks.

Our results show that some of the evaluated anomaly detectors provide reasonable accuracy with low detection delay. However, these detectors do not provide sustained accuracy on both the datasets. For instance, the Maximum Entropy detector [20] provides very high accuracy at the endpoints, but cannot provide the same level of accuracy at the edge router. Similarlyl, the credit-based TRW algorithm [8] provides reasonably high accuracy at endpoints, while its original counterpart, the TRW algorithm [7], outperforms all other algorithms at the edge router. The rate limiting detector [4], [5] that has been designed for malware detection fails to provide high accuracy at endpoints or routers. In summary, the detectors are unable to scale their accuracies for different points of network deployment.

To improve scalability and detection accuracy, we further evaluate two anomaly detectors, namely the Maximum Entropy and the PHAD detectors. These two detectors are somewhat unique because they allow more degrees of freedom in their feature spaces; i.e. they have higher dimensional feature spaces than the other detectors. Using these two detectors, we show that a promising approach to improve the accuracy of a detector is to operate it across a high dimensional feature space and/or over multiple time windows. We refer to these accuracy improving extensions as Space-Time (ST) variants of the original detectors.

2 Related Work

In this section, we focus on prior IDS/ADS evaluation studies. Details of anomaly detectors used in this work are deferred to subsequent sections.

Performance evaluation of IDSs received significant attention from the industry and academia in the late 1990's [30]–[45]. However, in the past few years, only four studies have performed comparative comparison of anomaly detectors

[26]–[29]. Similarly, very few prior studies have performed ROC analysis of the evaluated IDSs. Still fewer studies have made their evaluation datasets available online.

DARPA-funded IDS evaluation studies by the MIT Lincoln Lab in 1998 and 1999 represent a shift in the IDS evaluation methodology [33], [38]. Datasets used in these studies were made publicly available [39] and the ROC method used in these studies has since become the de facto standard for IDS accuracy evaluation. While some shortcomings of the DARPA evaluation have been highlighted [47], [48], in the absence of other benchmarks, the results and datasets of this study have been used extensively in subsequent works. In the present paper's context, the DARPA dataset is somewhat dated.

The four recent ADS evaluation studies focus on specific types of detectors and attacks [26]–[29]. The study by Wong et al. [26] is most relevant in the present context. Wong et al. [26] evaluated four variants of the rate limiting detector under portscan attacks at two different network points [4]–[11]. Two findings of this study are pertinent to the present work: 1) classical rate limiting is not an effective technique for portscan detection, and 2) rate limiting can operate on aggregate-level DNS traffic and hence can potentially scale to core-level deployments. Attack and background traffic data used in this study are not publicly available.

A comparative evaluation of bio-inspired anomaly detection algorithms was performed recently [27]. This work proposed to improve the accuracy of bio-inspired anomaly detectors by providing intelligent and discriminant features as inputs to the detectors' classification algorithms. The experimental results indicated that the use of intelligent features significantly improves the true positive and false positive rates of bio-inspired classifiers.

Ingham and Inoue [28] compared seven HTTP anomaly detection techniques under real-world attacks reported at public databases. These authors report the same evaluation difficulties that were faced by us: 1) Some anomaly detectors are not described completely; 2) Implementation source code is not available; and 3) labeled data used for algorithm evaluation are not publicly available. Consequently, the authors in [28] make their implementation and attack data publicly available "to encourage further experimentation". We subscribe to the same viewpoint and therefore all data and implementation used in this project are available online [25]. Lazarevic et al. performed a comparative analysis of four data mining based anomaly detection techniques in [29]. The live network traffic data used by this study is not publicly available.

3 Evaluation Datasets

We wanted to use real, labeled and public background and attack datasets to measure the accuracy of the evaluated anomaly detectors. Real and labeled data allow realistic and repeatable quantification of an anomaly detector's accuracy, which is a main objective of this work. Moreover, as defined in the introduction, another objective is to evaluate the accuracy or scalability of the anomaly detectors under different normal and attack traffic rates and at different points of deployment in the network. This evaluation objective is somewhat unique to this effort, with [26] being the only other study that provides some insight into host versus edge deployments.

Different network deployment points are responsible for handling traffic from varying number of nodes. For instance, an endpoint requires to cater for only its own traffic, while an edge router needs to monitor and analyze traffic from a variety of hosts in its subnet. In general, as one moves away from the endpoints towards the network core, the number of nodes, and consequently the traffic volume, that a network entity is responsible for increase considerably. We argue that if an algorithm that is designed to detect high- or low-rate attacks at a particular point of deployment, say an edge router, scales to and provides high accuracy at other traffic rates and deployment points, say at endpoints, then such an algorithm is quite valuable because it provides an off-the-shelf deployment option for different network entities. (We show later in this paper that some existing algorithms are able to achieve this objective.)

To test the anomaly detectors for scalability, we use two real traffic datasets that have been independently-collected at different deployment points. The first dataset is collected at the edge router of the Lawrence Berkeley National Laboratory (LBNL), while the second dataset is collected at network endpoints by our research lab[4]. In this section, we describe the data collection setups and the attack and background traffic characteristics of the LBNL and the endpoint datasets.

3.1 The LBNL Dataset

LBNL Background Traffic: This dataset was obtained from two international network locations at the Lawrence Berkeley National Laboratory (LBNL) in USA. Traffic in this dataset comprises packet-level incoming, outgoing and internally-routed traffic streams at the LBNL edge routers. Traffic was anonymized using the `tcpmkpub` tool; refer to [49] for details of anonymization.

LBNL data used in this study is collected during three distinct time periods. Some pertinent statistics of the background traffic are given in Table 1. The average remote session rate (i.e., sessions from distinct non-LBNL hosts) is approximately 4 sessions per second. The total TCP and UDP background traffic rate in packets per second is shown in column 5 of the table. A large variance can be observed in the background traffic rate at different dates. This variance will have an impact on the performance of volumetric anomaly detectors that rely on detecting bursts of normal and malicious traffic.

The main applications observed in internal and external traffic are Web (HTTP), Email and Name Services. Some other applications like Windows Services, Network File Services and Backup were being used by internal hosts;

[4] We also wanted to use a traffic dataset collected at a backbone ISP network; such datasets have been used in some prior studies [15]–[17]. However, we could not find a publicly available ISP traffic dataset.

Table 1. Background Traffic Information for the LBNL Dataset

Date	Duration(mins)	LBNL Hosts	Remote Hosts	Backgnd Rate(pkt/sec)	Attack Rate(pkt/sec)
10/4/04	10min	4,767	4,342	8.47	0.41
12/15/04	60min	5,761	10,478	3.5	0.061
12/16/04	60min	5,210	7,138	243.83	72

details of each service, information of each service's packets and other relevant description are provided in [50].

LBNL Attack Traffic: Attack traffic was isolated by identifying scans in the aggregate traffic traces. Scans were identified by flagging those hosts which unsuccessfully probed more than 20 hosts, out of which 16 hosts were probed in ascending or descending order [49]. Malicious traffic mostly comprises failed incoming TCP SYN requests; i.e., TCP portscans targeted towards LBNL hosts. However, there are also some outgoing TCP scans in the dataset. Most of the UDP traffic observed in the data (incoming and outgoing) comprises successful connections; i.e., host replies are received for the UDP flows. Table 1 [column 6] shows the attack rate observed in the LBNL dataset. Clearly, the attack rate is significantly lower than the background traffic rate. Thus these attacks can be considered low rate relative to the background traffic rate. (We show later that background and attack traffic at endpoints exhibit the opposite characteristics.)

Since most of the anomaly detectors used in this study operate on TCP, UDP and/or IP packet features, to maintain fairness we filtered the background data to retain only TCP and UDP traffic. Moreover, since most of the scanners were located outside the LBNL network, to remove any bias we filter out internally-routed traffic. After filtering the datasets, we merged all the background traffic data at different days and ports. Synchronized malicious data chunks were then inserted in the merged background traffic.

Since no publicly-available endpoint traffic set was available, we spent up to 14 months in collecting our own dataset on a diverse set of 13 endpoints. Complexity and privacy were two main reservations of the participants of the endpoint data collection study. To address these reservations, we developed a custom tool for endpoint data collection. This tool was a multi-threaded MS Windows application developed using the `Winpcap` API [51]. (Implementation of the tool is available at [25].) To reduce the packet logging complexity at the endpoints, we only logged some very elementary session-level information of TCP and UDP packets. Here a *session* corresponds to a bidirectional communication between two IP addresses; communication between the same IP address on different ports is considered part of the same network session. To ensure user privacy, the source IP address (which was fixed/static for a given host) is not logged, and each session entry is indexed by a one-way hash of the destination IP with the hostname. Most of the detectors evaluated in this work can operate with this level of data granularity.

Table 2. Background Traffic Information for Four Endpoints with High and Low Rates

Endpoint ID	Endpoint Type	Duration(months)	Total Sessions	Mean Session Rate(/sec)
3	Home	3	373,009	1.92
4	Home	2	444,345	5.28
6	Univ	9	60,979	0.19
10	Univ	13	152,048	0.21

3.2 Endpoint Dataset

Statistics of the two highest rate and the two lowest rate endpoints are listed in Table 2[5]. As can be intuitively argued, the traffic rates observed at the endpoints are much lower than those at the LBNL router. In the endpoint context, we observed that home computers generate significantly higher traffic volumes than office and university computers because: 1) they are generally shared between multiple users, and 2) they run peer-to-peer and multimedia applications. The large traffic volumes of home computers are also evident from their high mean number of sessions per second. For this study, we use 6 weeks of endpoint traffic data for training and testing. Results for longer time periods were qualitatively similar.

To generate attack traffic, we infected VMs on the endpoints by the following malware: `Zotob.G`, `Forbot-FU`, `Sdbot-AFR`, `Dloader-NY`, `SoBig.E@mm`, `MyDoom.A@mm`, `Blaster`, `Rbot-AQJ`, and `RBOT.CCC`; details of the malware can be found at [52]. These malware have diverse scanning rates and attack ports/applications. Table 3 shows statistics of the highest and lowest scan rate worms; `Dloader-NY` has the highest scan rate of 46.84 scans per second (sps), while `MyDoom-A` has the lowest scan rate of 0.14 sps, respectively. For completeness, we also simulated three additional worms that are somewhat different from the ones described above, namely `Witty`, `CodeRedv2` and a fictitious TCP worm with a fixed and unusual source port. `Witty` and `CodeRedv2` were simulated using the scan rates, pseudocode and parameters given in research and commercial literature [52], [53].

Endpoint Background Traffic: The users of these endpoints included home users, research students, and technical/administrative staff. Some endpoints, in particular home computers, were shared among multiple users. The endpoints used in this study were running different types of applications, including peer-to-peer file sharing software, online multimedia applications, network games, SQL/SAS clients etc.

Endpoint Attack Traffic: The attack traffic logged at the endpoints mostly comprises outgoing portscans. Note that this is the opposite of the LBNL dataset, in which most of the attack traffic is inbound. Moreover, the attack traffic rates

[5] The mean session rates in Table 2 are computed using time-windows containing one or more new sessions. Therefore, dividing total sessions by the duration does not yield the session rate of column 5.

Table 3. Endpoint Attack Traffic for Two High- and Two Low-rate Worms

Malware	Release Date	Avg. Scan Rate(/sec)	Port(s) Used
`Dloader-NY`	Jul 2005	46.84 sps	TCP 135,139
`Forbot-FU`	Sept 2005	32.53 sps	TCP 445
`MyDoom-A`	Jan 2006	0.14 sps	TCP 3127 – 3198
`Rbot-AQJ`	Oct 2005	0.68 sps	TCP 139,769

(Table 3) in the endpoint case are generally much higher than the background traffic rates (Table 2). This characteristic is also the opposite of what was observed in the LBNL dataset. This diversity in attack direction and rates provides us a sound basis for performance comparison of the anomaly detectors evaluated in this study [7], [8].

For each malware, attack traffic of 15 minutes duration was inserted in the background traffic of each endpoint at a random time instance. This operation was repeated to insert 100 non-overlapping attacks of each worm inside each endpoint's background traffic.

4 Anomaly Detection Algorithms

In this section, we focus on network-based anomaly detectors and compare the anomaly detectors proposed in [4], [7], [8], [12], [15], [18], [20], and [21]. Most of these detectors are quite popular and used frequently for performance comparison and benchmarking in the ID research community. Improvements to these algorithms have also been proposed in [6], [9]–[11], [13], [14], [16], [17], [19], and [26][6].

Before briefly describing these detectors, we highlight that some of these detectors are designed specifically for portscan detection, while others are general-purpose network anomaly detectors. More generically, based on the taxonomy of [54], the algorithms evaluated in this study can be subdivided into the ADS categories shown in Fig. 1. Clearly, the evaluated ADSs are quite diverse in their traffic features as well as their detection frameworks. These ADSs range from very simple rule modelling systems like PHAD [12] to very complex and theoretically-inclined self-Learning systems like the PCA-based subspace method [15] and the Sequential Hypothesis Testing technique [7]. This diversity is introduced to achieve the following objectives: a) to identify promising traffic features and theoretical frameworks for portscan anomaly detection; b) to investigate the accuracy, complexity and delays of these anomaly detectors under different attack and normal traffic scenarios and at different points of deployment in the network; and c) to identify a set of promising portscan detection guidelines that build on the strengths and avoid the weaknesses of the evaluated anomaly detectors.

[6] Some promising commercial ADSs are also available in the market now [22], [23]. We did not have access to these ADSs, and therefore these commercial products are not evaluated in this study.

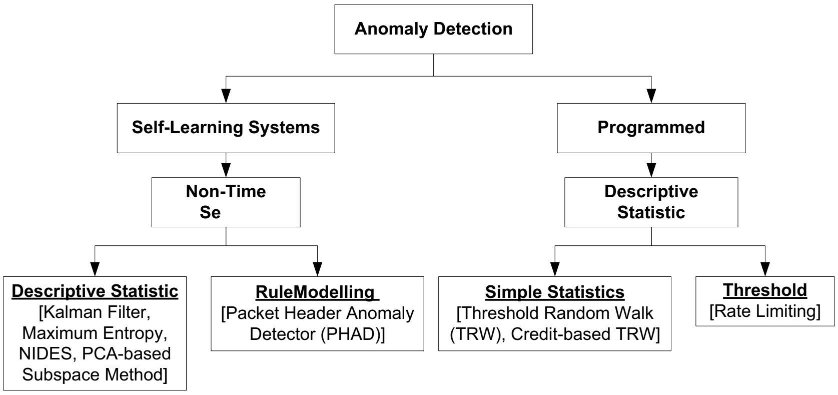


Fig. 1. Taxonomy of the anomaly detectors evaluated in this work [54]

Due to space constraints, we do not provide detailed descriptions of the evaluated algorithms. We instead focus on the algorithm adaptation and parameter tuning for the datasets under consideration. Readers are referred to [4], [7], [8], [12], [15], [18], [20], and [21] for details of the algorithms. For techniques operating on fixed-sized time windows, we use a window of 20 seconds. All other parameters not mentioned in this section are the same as those described in the algorithms' respective papers.

4.1 Rate Limiting

Rate limiting [4], [5] detects anomalous connection behavior by relying on the premise that an infected host will try to connect to many different machines in a short period of time. Rate limiting detects portscans by putting new connections exceeding a certain threshold in a queue. An alarm is raised when the queue length, η_q, exceeds a threshold. ROCs for endpoints are generated by varying $\eta_q = \mu + k\sigma$, where μ and σ represent the sample mean and sample standard deviation of the connection rates in the training set, and $k = 0, 1, 2, \ldots$ is a positive integer. Large values of k will provide low false alarm and detection rates, while small values will render high false alarm and detection rates. In the LBNL dataset, connection rate variance in the background traffic is more than the variance in the attack traffic. Therefore, to obtain a range of detection and false alarm rates for the LBNL dataset, we use a threshold of $\eta_q = w\mu$, with a varying parameter $0 \geq w \leq 1$, and the queue is varied between 5 and 100 sessions.

4.2 Threshold Random Walk (TRW) Algorithm

The TRW algorithm [7] detects incoming portscans by noting that the probability of a connection attempt being a success should be much higher for a benign host than for a scanner. To leverage this observation, TRW uses sequential hypothesis testing (i.e., a likelihood ratio test) to classify whether or not a remote host is a scanner. We plot ROCs for this algorithm by setting different values of false alarm and detection rates and computing the likelihood ratio thresholds, η_0 and η_1, using the method described in [7].

4.3 TRW with Credit-Based Rate Limiting (TRW-CB)

A hybrid solution to leverage the complementary strengths of Rate Limiting and TRW was proposed by Schechter et al. [8]. Reverse TRW is an anomaly detector that limits the rate at which new connections are initiated by applying the sequential hypothesis testing in a reverse chronological order. A credit increase/decrease algorithm is used to slow down hosts that are experiencing unsuccessful connections. We plot ROCs for this technique for varying η_0 and η_1 as in the TRW case.

4.4 Maximum Entropy Method

This detector estimates the benign traffic distribution using maximum entropy estimation [20]. Training traffic is divided into $2,348$ packet classes and maximum entropy estimation is then used to develop a baseline benign distribution for each packet class. Packet class distributions observed in real-time windows are then compared with the baseline distribution using the Kullback-Leibler (K-L) divergence measure. An alarm is raised if a packet class' K-L divergence exceeds a threshold, η_k, more than h times in the last W windows of t seconds each. Thus the Maximum Entropy method incurs a detection delay of at least $h \times t$ seconds. ROCs are generated by varying η_k.

4.5 Packet Header Anomaly Detection (PHAD)

PHAD learns the normal range of values for all 33 fields in the Ethernet, IP, TCP, UDP and ICMP headers [12]. A score is assigned to each packet header field in the testing phase and the fields' scores are summed to obtain a packet's aggregate anomaly score. We evaluate PHAD-C32 [12] using the following packet header fields: source IP, destination IP, source port, destination port, protocol type and TCP flags. Normal intervals for the six fields are learned from 5 days of training data. In the test data, fields' values not falling in the learned intervals are flagged as suspect. Then the top n packet score values are termed as anomalous. The value of n is varied over a range to obtain ROC curves.

4.6 PCA-Based Subspace Method

The subspace method uses Principal Component Analysis (PCA) to separate a link's traffic measurement space into useful subspaces for analysis, with each subspace representing either benign or anomalous traffic behavior [15]. The authors proposed to apply PCA for domain reduction of the Origin-Destination (OD) flows in three dimensions: number of bytes, packets, IP-level OD flows. The top k eigenvectors represent normal subspaces. It has been shown that most of the variance in a link's traffic is generally captured by 5 principal components [15]. A recent study showed that the detection rate of PCA varies with the level and method of aggregation [55]. It was also concluded in [55] that it may be impractical to run a PCA-based anomaly detector over data aggregated at the level of OD flows. We evaluate the subspace method using the number of TCP flows aggregated in 10 minutes intervals. To generate ROC results, we changed the number of normal subspace as $k = 1, 2, \ldots, 15$. Since the principal components capture maximum variance of the data, as we increase k, the dimension of the residual subspace reduces and fewer observations are available for detection. In other words, as more and more principal components are selected as normal subspaces, the detection and false alarm rates decrease proportionally. Since there is no clear detection threshold, we could not obtain the whole range of ROC values for the subspace method. Nevertheless, we evaluate and report the subspace method's accuracy results for varying number of principal components.

4.7 Kalman Filter Based Detection

The Kalman filter based detector of [18] first filters out the normal traffic from the aggregate traffic, and then examines the residue for anomalies. In [18], the Kalman Filter operated on SNMP data to detect anomalies traversing multiple links. Since SNMP data was not available to us in either dataset, we model the traffic as a 2-D vector X_t. The first element of X_t is the total number of sessions (in the endpoint dataset) or packets (in the LBNL dataset), while the second element is the total number of distinct remote ports observed in the traffic. We defined a threshold, η_f on the residue value r to obtain ROC curves. Thresholding of r is identical to the rate limiting case. An alarm is raised, if $r < -\eta_f$ or $r > \eta_f$.

4.8 Next-Generation Intrusion Detection Expert System (NIDES)

NIDES [21] is a statistical anomaly detector that detects anomalies by comparing a long-term traffic rate profile against a short-term, real-time profile. An anomaly is reported if the Q distribution of the real-time profile deviates considerably from the long-term values. After specific intervals, new value of Q are generated by monitoring the new rates and compared against a predefined threshold, η_s. If $\Pr(Q > q) < \eta_s$, an alarm is raised. We vary η_s over a range of values for ROC evaluation.

5 Performance Evaluation

In this section, we evaluate the accuracy, scalability, complexity and delay of the anomaly detectors described in the last section on the endpoint and router datasets.

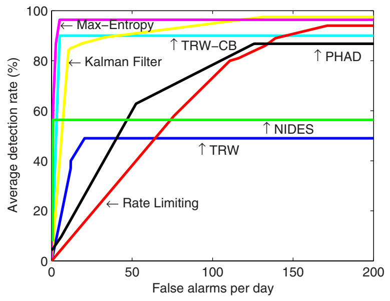
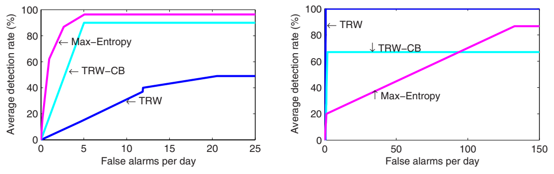


Fig. 2. ROC analysis on the endpoint dataset; each ROC is averaged over 13 endpoints with 12 attacks per endpoint and 100 instances per attack



(a) Endpoint dataset (b) LBNL dataset

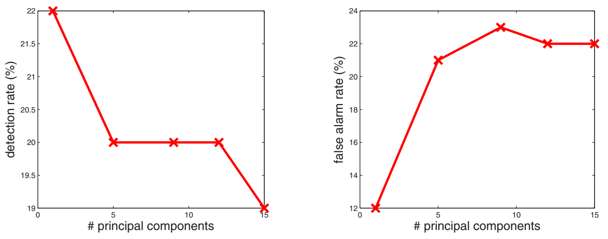
Fig. 3. Comparison of the Maximum Entropy, TRW and TRW-CB algorithms

5.1 Accuracy and Scalability Comparison

In this section, we present ROC analysis on the endpoint dataset. The following section explains the scalability experiments in which ROC analysis is performed on the LBNL dataset and the results are compared with the endpoint experiments.

Averaged ROCs for the Endpoint Dataset: Fig. 2 provides the averaged ROC analysis of the anomaly detection schemes under consideration. Clearly, the Maximum Entropy detector provides the highest accuracy by achieving near 100% detection rate at a very low false alarm rate of approximately 5 alarms/day. The Maximum Entropy detector is followed closely by the credit-based TRW approach. TRW-CB achieves nearly 90% detection rate at a reasonable false alarm rate of approximately 5 alarms/day. The original TRW algorithm, however, provides very low detection rates for the endpoint dataset. Results of these three schemes are shown more clearly in Fig. 3(a). Based on these results, the Maximum Entropy algorithm provides the best accuracy on endpoints, while TRW provides the best detection on LBNL dataset.

The Kalman Filter approach is also quite accurate as it provides up to 85% detection rates at a reasonably low false alarm cost. Rate Limiting, although designed to detect outgoing scanning attacks, provides very poor performance. This result substantiates the results of [26] where very high false positive rates



(a) Detection rate (b) false alarm rate

Fig. 4. Detection and false alarm rates for the subspace method [15]

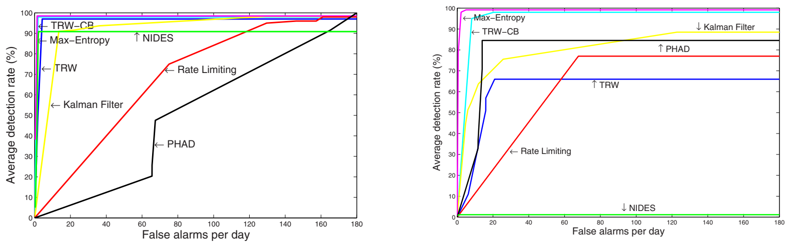
for high detection rates were reported for classical rate limiting. Hence, we also deduce that rate limiting is ineffective for portscan detection at endpoints.

PHAD does not perform well on the endpoint data set. The detection is accompanied with very high false alarm rates. NIDES achieve reasonable detection rates at very low false alarm rates, but is unable to substantially improve its detection rates afterwards. PHAD relies on previously seen values in the training dataset for anomaly detection. Therefore, if a scanner attacks a commonly-used port/IP then PHAD is unable to detect it. On similar grounds, if the malicious traffic is not bursty enough as compared to background traffic then NIDES will not detect it, irrespective of how much the detection threshold is tuned.

Due to the thresholding difficulties for the subspace method explained in Section 4, in Fig. 4 we report results for this technique for varying values of selected principal components. The highest detection rate of 22% is observed at $k = 2$ principal components. This already low detection rate decreases further at $k = 5$ and drops to 0% at $k = 15$. False alarm rates show the opposite trend. Thus the subspace method fails to give acceptable accuracy on the endpoint dataset.

The ROC results for the endpoint dataset are somewhat surprising because two of the top three detectors are general-purpose anomaly detectors (Maximum Entropy and Kalman Filter), but still outperform other detectors designed specifically for portscan detection, such as the TRW and the Rate Limiting detectors. We, however, note that this analysis is not entirely fair to the TRW algorithm because TRW was designed to detect incoming portscans, whereas our endpoint attack traffic contains mostly outgoing scan packets. The credit-based variant of TRW achieves high accuracy because it leverages outgoing scans for portscan detection. Thus TRW-CB combines the complementary strengths of rate limiting and TRW to provide a practical and accurate portscan detector for endpoints. This result agrees with earlier results in [26].

ROCs for Low- and High-Rate Endpoint Attacks: To evaluate the scalability of the ADSs under high- and low-rate attack scenarios, Fig. 5 plots the ROCs for the highest rate (`Dloader-NY`) and lowest rate (`MyDoom-A`) attacks in the endpoint dataset. It can be observed that for the high-rate attack [Fig. 5(a)] Maximum Entropy, TRW, TRW-CB and Kalman Filter techniques



(a) `Dloader-NY`, high scan rate (b) `MyDoom-A`, low scan rate

Fig. 5. ROC curves for the lowest and highest rate attack in the endpoint dataset; results averaged over 12 endpoints with 100 instances of each attack

provide excellent accuracy by achieving 100% or near-100% detection rates with few false alarms. NIDES' performance also improves as it achieves approximately 90% detection rate at very low false alarm rates. This is because the high-rate attack packets form bursts of malicious traffic that NIDES is tuned to detect. Rate Limiting and PHAD do not perform well even under high attack rate scenarios.

Fig. 5(b) shows that the accuracies of all detectors except PHAD and Maximum Entropy degrade under a low-rate attack scenario. Maximum Entropy achieves 100% detection rate with false alarm rate of 4-5 alarms/day. TRW-CB recovers quickly and achieves a near-100% detection rate for a daily false alarm rate around 10 alarms/day. NIDES, however, shows the biggest degradation in accuracy as its detection rate drops by approximately 90%. This is because low-rate attack traffic when mixed with normal traffic does not result in long attack bursts. TRW's accuracy is also affected significantly as its detection rate drops by about 35% as compared to the high-rate attack. PHAD does not rely on traffic rate for detection, and hence its accuracy is only dependent on the header values observed during training.

Averaged ROCs for the LBNL Dataset: Fig. 6 shows the ROCs for the LBNL dataset. Comparison with Fig. 3 (a) and (b) reveals that the Maximum Entropy detector is unable to maintain its high accuracy on the LBNL dataset; i.e., the Maximum Entropy algorithm cannot scale to different points of network deployment. TRW's performance improves significantly as it provides a 100% detection rate at a negligible false alarm cost. TRW-CB, on the other hand, achieves a detection rate of approximately 70%. Thus contrary to the endpoint dataset, the original TRW algorithm easily outperforms the TRW-CB algorithm on LBNL traces. As explained in Section 3.1, the LBNL attack traffic mostly comprises failed incoming TCP connection requests. TRW's forward sequential hypothesis based portscan detection algorithm is designed to detect such failed incoming connections, and therefore it provides high detection rates. Thus on an edge router, TRW represents a viable deployment option.

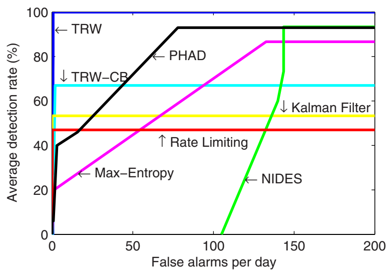


Fig. 6. ROC analysis on the LBNL dataset

Table 4. Complexity Comparison of the Anomaly Detectors

	Rate Limiting	TRW	TRW-CB	Max Entropy	NIDES	PHAD	Subspace Method	Kalman Filter
Training Time(sec)	48.58	37948.42	23185.11	50.06	2.01	17650.52	34111.89	16.56
Detection Time(sec)	15961.75	31339.43	18226.38	22.79	5.4	25082.45	15939.27	12.58
Training Memory(KB)	22789.04	21110.86	25104.67	57223.39	3590.95	48100.35	68851.71	34515.98
Detection Memory(KB)	14667.4	49087.78	67545.53	66930.06	4013.08	42262.46	16685.93	60314.65

Kalman Filter detector's accuracy drops as it is unable to achieve a detection rate above 60%. PHAD provides very high detection rates, albeit at an unacceptable false alarm rate. Other detectors' results are similar to the endpoint case. (Results for the subspace method were similar to those reported earlier and are skipped for brevity.) It can be observed from Fig. 6 that all algorithms except TRW fail to achieve 100% detection rates on the LBNL dataset. This is because these algorithms inherently rely on the high burstiness and volumes of attack traffic. In the LBNL dataset, the attack traffic rate is much lower than the background traffic rate. Consequently, the attack traffic is distributed across multiple time windows, with each window containing very few attack packets. Such low density of attack traffic in the evaluated time-windows remains undetected regardless of how much the detection thresholds are decreased.

5.2 Complexity and Delay Comparison

Table 4 lists the training and classification time taken by the anomaly detectors as well as their training and run-time memory requirements. These numbers are computed using the `hprof` tool. The first observation from the table is that, contrary to common intuition, complexity does not translate directly into accuracy of an anomaly detector. For instance, the Maximum Entropy detector, while providing the highest accuracy on endpoints, has orders of magnitude lower training and run-time complexity than many other detectors. NIDES is by far the least complex algorithm requiring only a few seconds for training and execution. However, NIDES' low accuracy makes it an unsuitable choice for practical

Table 5. Detection Delay of the Anomaly Detectors

	Rate Limiting	TRW	TRW-CB	Max Entropy	NIDES	PHAD	Subspace Method	Kalman Filter
`MyDoom` (msec)	310	510	40	215000	∞	900	79	377
`Dloader-NY` (msec)	140	320	20	56000	0.086	990	23	417
LBNL (msec)	660	660	290	86000	330	330	∞	800

deployments. The Kalman Filter based detector is also extremely low complexity. Thus despite its low accuracy at edge routers, it is still a viable deployment option at network endpoints.

The credit-based TRW algorithm has slightly lower complexity than the originally-proposed TRW algorithm. Since TRW-CB also provides better accuracy than TRW at endpoints, it is a good deployment option for systems without significant complexity constraints. Rate Limiting, while having low complexity, is not a practical deployment option because of its poor accuracy. The subspace method and PHAD have high complexity requirements, but still render very poor accuracy.

Table 5 provides the detection delay for each anomaly detector. On the endpoint dataset, delay is reported for the highest and the lowest rate attacks, while on the LBNL dataset this delay is computed for the first attack that is detected by an anomaly detector. A delay value of ∞ is listed if an attack is not detected altogether. It can be observed that detection delay is reasonable (less than 1 second) for all the anomaly detectors except the Maximum Entropy detector which incurs very high detection delays. High delays are observed for the Maximum Entropy detector because it waits for perturbations in multiple time windows before raising an alarm. Among other viable alternatives, TRW-CB provides the lowest detection delays for all three experiments. Detection delay for the TRW is also reasonably low.

6 Summary, Discussion and Future Work

6.1 Summary

In this paper, we evaluated eight prominent network-based anomaly detectors using two portscan traffic datasets having complementary characteristics. These detectors were evaluated on accuracy, scalability, complexity and delay criteria. Based on the results of this paper, we now rephrase and summarize our deductions pertaining to the main objectives of this study:

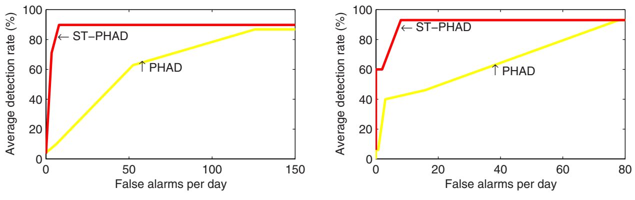
– Which algorithms provide the best accuracy under varying rates of attack and normal traffic and at different points of deployment? Under the varying attack and background traffic rates observed in the two datasets, a general-purpose Maximum Entropy Detector [20] and variants of the Threshold Random Walk (TRW) algorithm [7], [8] provided the best overall performance under most evaluation criteria. In this context, TRW is suitable for deployment at routers, while TRW-CB and Maximum Entropy are suitable for deployment at endpoints.

– What are the promising traffic features and theoretical frameworks for portscan anomaly detection? The Maximum Entropy and TRW detectors use statistical distributions of failed connections, ports and IP addresses. Furthermore, based on the results of the Maximum Entropy detector on endpoints, a histogram-based detection approach, in which baseline frequency profiles of a set of features is compared with real-time feature frequencies, appears very promising.
– Does complexity improve accuracy? Complexity has no relation with the accuracy of an algorithm. Intelligence of traffic features and detection frameworks determine the accuracy of an algorithm.
– What detection delays are incurred by the anomaly detectors? If an attack is detected, detection delay is less than 1 second for all anomaly detectors, except the Maximum Entropy Estimation method which incurs very large delays.
– What are promising portscan detection guidelines that build on the strengths and avoid the weaknesses of the evaluated anomaly detectors? From the high detection rates of the Maximum Entropy and PHAD detectors, it appears that using a higher dimensional feature space facilitates detection, without compromising complexity. On the other hand, relying on specific traffic features (e.g., rate, connection failures, etc.) can degrade accuracy as the attack and background traffic characteristics change. In summary, a number of statistical features used in an intelligent histogram-based classification framework appear promising for portscan anomaly detection.

6.2 Discussion

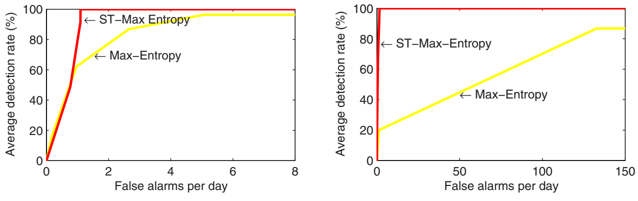
Based on our comparative analysis and deductions, we further evaluate Maximum Entropy and PHAD detectors. Both these detectors have a high dimensional feature space, thereby allowing us to improve their accuracy by evaluating them in feature space and/or time. For instance, recall that PHAD operates on a high dimensional feature space and detects an anomaly when a certain number of features in a packet are perturbed. Thus PHAD leverages its large feature space to achieve high detection rates, but the high false alarm rates render the detector useless. To reduce PHAD's false alarms, in addition to operating the PHAD detector in its feature space, we also operate it in time and an alarm is raised only when an anomaly is observed in multiple time windows/packets. This Space-Time (ST) strategy should reduce the false alarm rate of PHAD.

Maximum Entropy, on the other hand, operates across multiple time windows before raising an alarm. Such a strategy results in very low false alarm rates, but compromises the detection rate and delay, as seen in Fig. 6 and Table 5. Extending this algorithm across its high dimensional feature space should improve its detection rate and delay. Thus, instead of waiting for h windows before making



(a) `Endpoint dataset` (b) `LBNL dataset`

Fig. 7. ROC comparison of PHAD and its Space-Time variant (ST-PHAD)



(a) `Endpoint dataset` (b) `LBNL dataset`

Fig. 8. ROC comparison of Maximum Entropy and its Space-Time variant (ST-Max-Entropy)

Table 6. Detection Delay of the ST variants of Maximum Entropy and PHAD

	ST-Max Entropy	ST-PHAD
`MyDoom` (msec)	157	900
`Dloader-NY` (msec)	100	990
LBNL (msec)	333	330

a decision, we modified the Maximum Entropy method to raise an alarm if the K-L divergence of l packet classes in a given time-window exceed η_k.

Figs. 7 and 8 show a comparative analysis of the original and the Space-Time variants of PHAD and Maximum Entropy. It can be seen that evaluating PHAD across multiple time windows using a high dimensional feature space clearly improves the accuracy of the detector. Similarly, evaluating Maximum Entropy across its feature space instead of its original time domain design considerably improves the accuracy and detection delay of the detector.

Table 6 provides the detection delay for the Space-Time variants of Maximum Entropy detector and PHAD. It can be observed that the detection delay for ST variant of the Maximum Entropy detector is dramatically lower than the original algorithm. This is because the ST variant of the Maximum Entropy detector does not wait for multiple anomalous windows before raising an alarm. For PHAD, the detection delay remains unaltered because the ST-PHAD variants simultaneously operates in space and time.

6.3 Future Work

The preliminary results of this section show that operating an anomaly detector in space and time can improve the detector's accuracy and delay. In a future work, we will report detailed results on this finding.

Acknowledgements

This work is supported by the National ICT R&D Fund, Ministry of Information Technology, Government of Pakistan. The information, data, comments, and views detailed herein may not necessarily reflect the endorsements of views of the National ICT R&D Fund.

The authors would like to thank Aamir Shafi for providing unrestricted access to the Solaris cluster for our experiments.

References

1. Symantec Internet Security Threat Reports I–XI (January 2002–January 2008)
2. McAfee Corp., McAfee Virtual Criminology Report: North American Study into Organized Crime and the Internet (2005)
3. Computer Economics: Economic Impact of Malicious Code Attacks (2001), http://www.computereconomics.com/cei/press/pr92101.html
4. Williamson, M.M.: Throttling viruses: Restricting propagation to defeat malicious mobile code. In: ACSAC (2002)
5. Twycross, J., Williamson, M.M.: Implementing and testing a virus throttle. In: Usenix Security (2003)
6. Sellke, S., Shroff, N.B., Bagchi, S.: Modeling and automated containment of worms. In: DSN (2005)
7. Jung, J., Paxson, V., Berger, A.W., Balakrishnan, H.: Fast portscan detection using sequential hypothesis testing. In: IEEE Symp. Sec. and Priv. (2004)
8. Schechter, S.E., Jung, J., Berger, A.W.: Fast detection of scanning worm infections. In: Jonsson, E., Valdes, A., Almgren, M. (eds.) RAID 2004. LNCS, vol. 3224, pp. 59–81. Springer, Heidelberg (2004)
9. Weaver, N., Staniford, S., Paxson, V.: Very fast containment of scanning worms. In: Usenix Security (2004)
10. Chen, S., Tang, Y.: Slowing Down Internet Worms. In: IEEE ICDCS (2004)
11. Ganger, G., Economou, G., Bielski, S.: Self-Securing Network Interfaces: What, Why, and How. Carnegie Mellon University Technical Report, CMU-CS-02-144 (2002)
12. Mahoney, M.V., Chan, P.K.: PHAD: Packet Header Anomaly Detection for Indentifying Hostile Network Traffic. Florida Tech. technical report CS-2001-4 (2001)
13. Mahoney, M.V., Chan, P.K.: Learning Models of Network Traffic for Detecting Novel Attacks. Florida Tech. technical report CS-2002-08 (2002)
14. Mahoney, M.V., Chan, P.K.: Network Traffic Anomaly Detection Based on Packet Bytes. In: ACM SAC (2003)
15. Lakhina, A., Crovella, M., Diot, C.: Characterization of network-wide traffic anomalies in traffic flows. In: ACM Internet Measurement Conference (IMC) (2004)

16. Lakhina, A., Crovella, M., Diot, C.: Diagnosing network-wide traffic anomalies. In: ACM SIGCOMM (2004)
17. Lakhina, A., Crovella, M., Diot, C.: Mining anomalies using traffic feature distributions. In: ACM SIGCOMM (2005)
18. Soule, A., Salamatian, K., Taft, N.: Combining Filtering and Statistical methods for anomaly detection. In: ACM/Usenix IMC (2005)
19. Zou, C.C., Gao, L., Gong, W., Towsley, D.: Monitoring and early warning of Internet worms. In: ACM CCS (2003)
20. Gu, Y., McCullum, A., Towsley, D.: Detecting anomalies in network traffic using maximum entropy estimation. In: ACM/Usenix IMC (2005)
21. Next-Generation Intrusion Detection Expert System (NIDES), http://www.csl.sri.com/projects/nides/
22. Peakflow-SP and Peakflow-X, http://www.arbornetworks.com/peakflowsp, http://www.arbornetworks.com/peakflowx
23. Cisco IOS Flexible Network Flow, http://www.cisco.com/go/netflow
24. LBNL/ICSI Enterprise Tracing Project, http://www.icir.org/enterprise-tracing/download.html
25. WisNet ADS Comparison Homepage, http://wisnet.niit.edu.pk/projects/adeval
26. Wong, C., Bielski, S., Studer, A., Wang, C.: Empirical Analysis of Rate Limiting Mechanisms. In: Valdes, A., Zamboni, D. (eds.) RAID 2005. LNCS, vol. 3858, pp. 22–42. Springer, Heidelberg (2006)
27. Shafiq, M.Z., Khayam, S.A., Farooq, M.: Improving Accuracy of Immune-inspired Malware Detectors by using Intelligent Features. In: ACM GECCO (2008)
28. 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)
29. Lazarevic, A., Ertoz, L., Kumar, V., Ozgur, A., Srivastava, J.: A Comparative Study of Anomaly Detection Schemes in Network Intrusion Detection. In: SIAM SDM (2003)
30. Mueller, P., Shipley, G.: Dragon claws its way to the top. In: Network Computing (2001), http://www.networkcomputing.com/1217/1217f2.html
31. The NSS Group: Intrusion Detection Systems Group Test (Edition 2) (2001), http://nsslabs.com/group-tests/intrusion-detection-systems-ids-group-test-edition-2.html
32. Yocom, B., Brown, K.: Intrusion battleground evolves, Network World Fusion (2001), http://www.nwfusion.com/reviews/2001/1008bg.html
33. Lippmann, R.P., Haines, J.W., Fried, D.J., Korba, J., Das, K.: The 1999 DARPA OffLine Intrusion Detection Evaluation. Comp. Networks 34(2), 579–595 (2000)
34. Durst, R., Champion, T., Witten, B., Miller, E., Spagnuolo, L.: Testing and Evaluating Computer Intrusion Detection Systems. Comm. of the ACM 42(7), 53–61 (1999)
35. Shipley, G.: ISS RealSecure Pushes Past Newer IDS Players. In: Network Computing (1999), http://www.networkcomputing.com/1010/1010r1.html
36. Shipley, G.: Intrusion Detection, Take Two. In: Network Computing (1999), http://www.nwc.com/1023/1023f1.html
37. Roesch, M.: Snort – Lightweight Intrusion Detection for Networks. In: USENIX LISA (1999)

38. Lippmann, R.P., Fried, D.J., Graf, I., Haines, J.W., Kendall, K.R., McClung, D., Weber, D., Webster, S.E., Wyschogrod, D., Cunningham, R.K., Zissman, M.A.: Evaluating Intrusion Detection Systems: The 1998 DARPA Off-Line Intrusion Detection Evaluation. In: DISCEX, vol. (2), pp. 12–26 (2000)
39. DARPA-sponsored IDS Evaluation (1998 and 1999). MIT Lincoln Lab, Cambridge, www.ll.mit.edu/IST/ideval/data/data_index.html
40. Debar, H., Dacier, M., Wespi, A., Lampart, S.: A workbench for intrusion detection systems. IBM Zurich Research Laboratory (1998)
41. Denmac Systems, Inc.: Network Based Intrusion Detection: A Review of Technologies (1999)
42. Ptacek, T.H., Newsham, T.N.: Insertion, Evasion, and Denial of Service: Eluding Network Intrusion Detection. Secure Networks, Inc. (1998)
43. Aguirre, S.J., Hill, W.H.: Intrusion Detection Fly-Off: Implications for the United States Navy. MITRE Technical Report MTR 97W096 (1997)
44. Puketza, N., Chung, M., Olsson, R.A., Mukherjee, B.: A Software Platform for Testing Intrusion Detection Systems. IEEE Software 14(5), 43–51 (1997)
45. Puketza, N.F., Zhang, K., Chung, M., Mukherjee, B., Olsson, R.A.: A Methodology for Testing Intrusion Detection Systems. IEEE Trans. Soft. Eng. 10(22), 719–729 (1996)
46. Mell, P., Hu, V., Lippmann, R., Haines, J., Zissman, M.: An Overview of Issues in Testing Intrusion Detection Systems. NIST IR 7007 (2003)
47. McHugh, J.: The 1998 Lincoln Laboratory IDS Evaluation (A Critique). In: Debar, H., Mé, L., Wu, S.F. (eds.) RAID 2000. LNCS, vol. 1907, Springer, Heidelberg (2000)
48. Mahoney, M.V., Chan, P.K.: An Analysis of the 1999 DARPA/Lincoln Laboratory Evaluation Data for Network Anomaly Detection. In: Vigna, G., Krügel, C., Jonsson, E. (eds.) RAID 2003. LNCS, vol. 2820, pp. 220–237. Springer, Heidelberg (2003)
49. Pang, R., Allman, M., Paxson, V., Lee, J.: The Devil and Packet Trace Anonymization. In: ACM CCR, vol. 36(1) (2006)
50. Pang, R., Allman, M., Bennett, M., Lee, J., Paxson, V., Tierney, B.: A First Look at Modern Enterprise Traffic. In: ACM/USENIX IMC (2005)
51. Winpcap homepage, http://www.winpcap.org/
52. Symantec Security Response, http://securityresponse.symantec.com/avcenter
53. Shannon, C., Moore, D.: The spread of the Witty worm. IEEE Sec & Priv 2(4), 46–50 (2004)
54. Axelsson, S.: Intrusion Detection Systems: A Survey and Taxonomy. Technical Report 99-15, Chalmers University (2000)
55. Ringberg, H., Rexford, J., Soule, A., Diot, C.: Sensitivity of PCA for Traffic Anomaly Detection. In: ACM SIGMETRICS (2007)

Advanced Network Fingerprinting

Humberto J. Abdelnur, Radu State, and Olivier Festor

Centre de Recherche INRIA Nancy - Grand Est
615, rue du jardin botanique
Villers-les-Nancy, France
{Humberto.Abdelnur,Radu.State,Olivier.Festor}@loria.fr
http://madynes.loria.fr

Abstract. Security assessment tasks and intrusion detection systems do rely on automated fingerprinting of devices and services. Most current fingerprinting approaches use a signature matching scheme, where a set of signatures are compared with traffic issued by an unknown entity. The entity is identified by finding the closest match with the stored signatures. These fingerprinting signatures are found mostly manually, requiring a laborious activity and needing advanced domain specific expertise. In this paper we describe a novel approach to automate this process and build flexible and efficient fingerprinting systems able to identify the source entity of messages in the network. We follow a passive approach without need to interact with the tested device. Application level traffic is captured passively and inherent structural features are used for the classification process. We describe and assess a new technique for the automated extraction of protocol fingerprints based on arborescent features extracted from the underlying grammar. We have successfully applied our technique to the Session Initiation Protocol (SIP) used in Voice over IP signalling.

Keywords: Passive Fingerprinting, Feature extraction, Structural syntax inference.

1 Introduction

Many security operations rely on the precise identification of a remote device or a subset of it (e.g. network protocol stacks, services). In security assessment tasks, this fingerprinting step is essential for evaluating the security of a remote and unknown system; especially network intrusion detection systems might use this knowledge to detect rogue systems and stealth intruders. Another important applicability resides in blackbox devices/application testing for potential copyright infringements. In the latter case, when no access to source code is provided, the only hints that might detect a copyright infringement can be obtained by observing the network level traces and determine if a given copyright protected software/source code is used unlawfully.

The work described in this paper was motivated by one major challenge that we had to face when building a Voice over IP (VoIP) specific intrusion detection system. We had to fingerprint VoIP devices and stacks in order to detect the presence of a rogue system on the network. Typically, only some vendor specific devices should be able to connect, while others and potentially malicious intended systems had to be detected and blocked. We decided that an automated system, capable to self-tune and self-deploy was the only viable solution on the long run. Therefore, we considered that the ideal system has to be able to process captured and labeled network traffic and detect the structural features that serve as potential differentiators. When searching for such potential features, there are some natural candidates: the type of individual fields and their length or the order in which they appear. For instance, the presence of login headers, the quantities of spaces after commas or the order presented in the handshake of capabilities. Most existing systems use such features, but individual signatures are built manually requiring a tedious and time consuming process.

Our approach consists in an automated solution for this task, assuming a known syntax specification of the protocol data units. We have considered only the signalling traffic - all devices were using Session Initiation Protocol [1] (SIP) - and our key contribution is to differentiate stack implementations by looking at some specific patterns in how the message processing code has been designed and developed. This is done in two main steps. In the first step, we extract features that can serve to differentiate among several stack implementations. These features are used in a second phase in order to implement a decisional process. This approach and the supporting algorithms are presented in this paper.

This paper is organized as follow. Section 2 illustrates the overall architecture and operational framework of our fingerprinting system. Section 3 shows how structural inference, comparison and identification of differences can be done based on the underlying grammar of a given specified protocol. Section 4 introduces the training, calibration and classification process. We provide an overview of experimental results in Sect. 5 using the signalling protocol (SIP) as an application case. Section 6 describes the related work in the area of fingerprinting as well as the more general work on structural similarity. Finally, Sect. 7 points out future works and concludes this paper.

2 Structural Protocol Fingerprinting

Most known application level and network protocols use a syntax specification based on formal grammars. The essential issue is that each individual message can be represented by a tree like structure. We have observed that stack implementers can be tracked by some specific subtrees and/or collection of subtrees appearing in the parse trees. The key idea is that structural differences between two devices can be detected by comparing the underlying parse trees generated for several messages. A structural signature is given by features that are extracted from these tree structures. Such distinctive features are called fingerprints. We will address in the following the automated identification of them.

If we focus for the moment one individual productions (in a grammar rule), the types of signatures might be given by:

– Different **contents** for one field. This is in fact a sequence of characters which can determinate a signature. (e.g. a prompt or an initialization message).
– Different **lengths** for one field. The grammar allows the production of a repetition of items (e.g. quantity of spaces after a symbol, capabilities supported). In this case, the length of the field is a good signature candidate.
– Different **orders** in one field. This is possible, when no explicit order is specified in a set of items. A typical case is how capabilities are advertised in current protocols.

We propose a learning method to automatically identify distinctive structural signatures. This is done by analyzing and comparing captured messages traces from the different devices. The overview of the learning and classification process is illustrated in Fig.1.

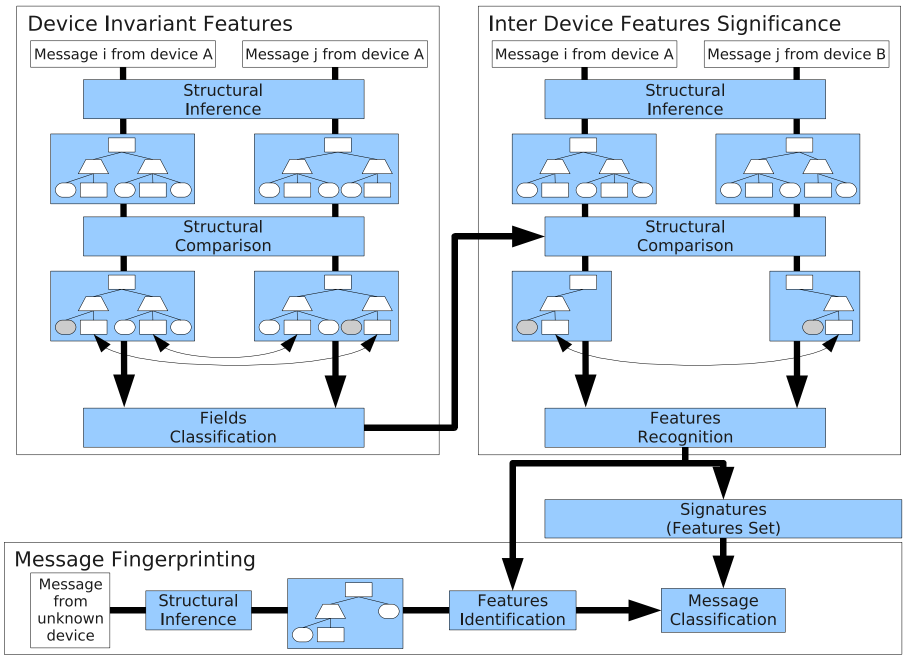


Fig. 1. Fingerprinting training and classification

The upper boxes in Fig.1 constitute the training period of the system. The output is a set of signatures for each device presented in the training set. The lowest box represents the fingerprinting process. The training is divided in two phases:

Phase 1 (*Device Invariant Features*). In this phase, the system automatically classifies each field in the grammar. This classification is needed to identify which fields may change between messages coming from the same device.

Phase 2 (*Inter Device Features Significance*) identifies among the Invariant fields of each implementation, those having different values for at least two group of devices. These fields will constitute part of the signatures set.

When one message has to be classified, the values of each invariant field are extracted and compared to the signature values learned in the training phase.

3 Structural Inference

3.1 Formal Grammars and Protocol Fingerprinting

The key assumption made in our approach is that an Augmented BackusNaur Form (ABNF) grammar [2] specification is a priori known for a given protocol. Such a specification is made of some basic elements as shown in Fig.2.

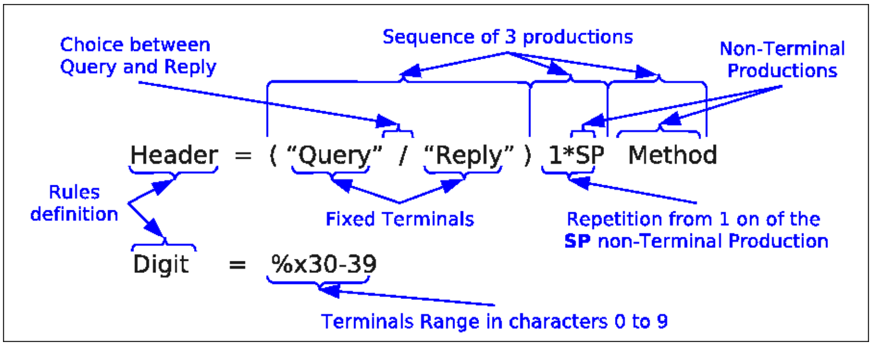

Choice between Query and Reply
Sequence of 3 productions
Non-Terminal Productions
Header = ("Query" / "Reply") 1*SP Method
Rules definition
Fixed Terminals
Repetition from 1 on of the SP non-Terminal Production
Digit = %x30-39
Terminals Range in characters 0 to 9

Fig. 2. Basic elements of a grammar

– A **Terminal** can represent a fixed string or a character to be chosen from a range of legitimate characters.
– A **Non-Terminal** is reduced using some rules to either a Terminal or a Non-Terminal.
– A **Choice** defines an arbitrary selection among several items.
– A **Sequence** involves a fixed number of items, where the order is specified.
– A **Repetition** involves a sequence of one item/group of items, where some additional constraints might be specified.

A given message is parsed according to the fields defined in the grammar. Each element of the grammar is placed in an n-ary tree which obeys the following rules:

- A **Terminal** becomes a leaf node with a name associated (i.e. the terminal that it represents) which is associated to the encountered value in the message.
- A **Non-Terminal** is an internal node associated to a name (i.e. the non-terminal rule) and it has a unique child which can be any of the types defined here (e.g. Terminal, non-Terminal, Sequence or Repetition).
- A **Sequence** is an internal node that has a fixed number of children. This number is in-line with the rules of the syntax specification.
- A **Repetition** is also an internal node, but having a number of children that may vary based on the number of items which appear in the message.
- A **Choice** does not create any node in the tree. However, it just marks the node that has been elected from a choice item.

It is important to note that even if sequences and repetitions do not have a defined name in the grammar rules, an implicit name is assigned to them that uniquely distinguishes each instance of these items at the current rule.

Figure 3 shows a Toy ABNF grammar defined in (a), messages from different implementation compliant with the grammar in (b/c) and (d) the inferred structure representing one of the messages in (d).

With respect to the usage, fields can be classified in three categories:

- **Cosmetics Fields:** these fields are mandatory and do not really provide a value added interest for fingerprinting purposes. The associated values do not change in different implementations.
- **Static Fields:** are the fields which values never change in a same implementation. These values do however change between different implementations. Obviously, these are the type of fields which may represent a signature for one implementation.
- **Dynamic Fields:** these fields are the opposite of static fields and do change their values in relation to semantic aspects of the message even in a single implementation.

An additional sub-classification can be defined for Dynamic and Static fields:

- **Value Type** relates to the String reduction of the node (i.e. the text information of that node),
- **Choice Type** relates to the selected choice from the grammar,
- **Length Type** corresponds to the number of items in a Repetition reduction,
- **Order Type** corresponds to the order in which items of a Repetition reduction appear.

Even if one implementation may generate different kind of values for the same field, such values could be related by a function and then serve as a feature. Therefore, a **Function Type** can be also defined to be used to compute the value from a node of the tree and return an output useful for the fingerprinting. Essentially, this type is used for manually tuning the training process.

```
Message    =   Header 1*SP 1*( "(" Opt-Value ")" )
Header     =   ("Query" / "Reply") 1*SP Method
Opt-Value  =   (Ack / Value / Version)
Method     =   1*ALPHA
Ack        =   "Ack" HCOLON 1*DIGIT
Value      =   "Value" HCOLON 1*ALPHA
Version    =   "Version" 1*SP DIGIT "." DIGIT
ALPHA      =   %x41-5A / %x61-7A              ; A-Z / a-z
DIGIT      =   %x30-39                        ; 0-9
HCOLON     =   *SP ":" *SP
SP         =   %x20                           ; space
```

(a) Toy Grammar

```
1        Query USER (Version 1.0)(Ack: 1)
2        Reply  USER (Version  1.1)(Ack : 1)(Value : Alex)

3        Query NAME (Version 1.0)(Ack: 2)
4        Reply  NAME (Version  1.1)(Ack  : 2)(Value : Alexander)

4        Query PASS (Version 1.0)(Ack: 3)
5        Reply  PASS (Version  1.1)(Ack  : 3)(Value : admin)
```

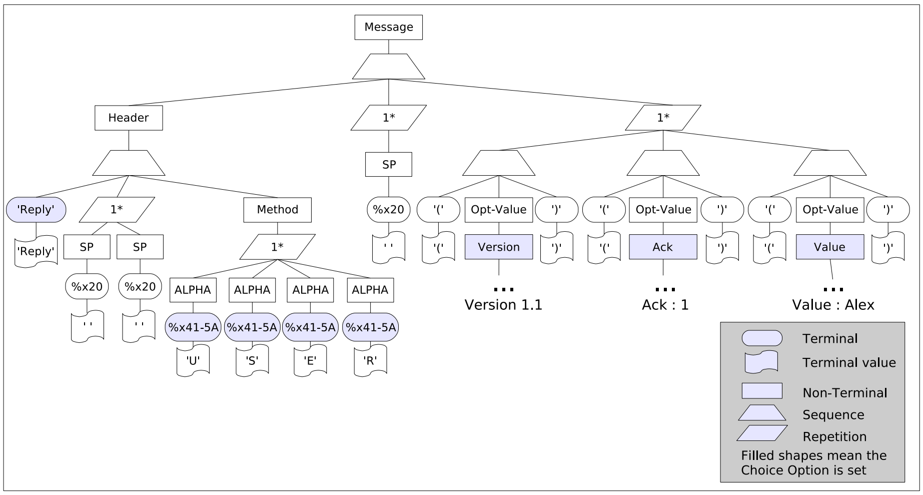
(b) Dialog between device A and B

```
1        Query USER (Version 1.1)(Ack  : 10)
2        Reply  USER (Ack:10)(Value: alex)(Value: Alex)

3         Query ITEMS (Version 1.0)(Ack  : 15)
4         Reply ITEMS (Ack:15)(Version 1.0)(Value : 512)(Value: 23)(Value: 36)

6         Query PASS (Ack  : 19)
5         Reply PASS (Ack:19)(Value:password)
```

(c) Dialog between device B and D



(d) Parsed Structure from Message Number 2 of Box (b)

Fig. 3. Parsed Structure Grammar

3.2 Node Signatures and Resemblance

Guidelines for designing a set of tree signatures (for a tree or a sub-tree) should follow some general common sense principles like:

- As more items are shared between trees, the more similar their signatures must be.
- Nodes that have different tags or ancestors must be considered different.
- In cases where the parent node is a Sequence, the location order in the Sequence should be part of the tree signature.
- If the parent node is a repetition, the location order should not be part of the tree signature, order will be captured later on in the fingerprinting features.

The closest known approach is published by D. Buttler in [3]. This method starts by encoding the tree in a set. Each element in the set represents a partial path from the root to any of the nodes in the tree. A resemblance method defined by A. Broder [4] uses the elements of the set as tokens. This resemblance is based on shingles, where a shingle is a contiguous sequence of tokens from the document. Between documents D_i and D_j the resemblance is defined as:

$$r(D_i, D_j) = \frac{|S(D_i, w) \bigcap S(D_j, w)|}{|S(D_i, w) \bigcup S(D_j, w)|} \tag{1}$$

where $S(D_i, w)$ creates the shingles of length w for the document D_i.

Definition 1. *The* ***Node Signature*** *function is defined to be a Multi-Set of all partial paths belonging to the sub-branch of the node.*

The *partial paths* start from the current node rather than from the root of the tree, but still goes through all the nodes of the subtree which has the current element as root like it was in the original approach. However, partial paths obtained from fields classified as *Cosmetics* are excluded from this Multi-Set. The structure used is a Multi-Set rather than a Set in order to store the quantity of occurrences for specific nodes in the sub-branch. For instance, the number of spaces after a specific field can determinate a signature in an implementation.

Siblings nodes in a Sequence items are fixed and representative. Sibling nodes in a Repetition can be made representative creating the partial paths of the Multi-Set and using the respective position of a child.

Table 1 shows the *Node Signature* obtained from the node *Header* at the tree of Fig. 3 (d).

Definition 2. *The* ***Ressemblance*** *function used to measure the degree of similarity between two nodes is based on the (1). The $S(N_i, w)$ function applies the Node Signature function over the node N_i.*

Using $w = 1$ allows to compare the number of items these nodes have in common though ignoring their position for a repetition.

3.3 Structural Difference Identification

Algorithm 1 is used to identify differences between two nodes which share the same ancestor path in the two trees,

where the functions *Tag*, *Value*, *Type* return the name, value and respectively the type of the current node. Note that $Tag(node_a) = Tag(node_b) \Rightarrow Type(node_a) = Type(node_b)$.

The function **Report_Difference** takes the type of difference to report and the corresponding two nodes. Each time the function is called, it creates one structure that stores the type of difference, the partial path from the root of the tree to the current nodes (which is the same for both nodes) and a corresponding value. For differences of type 'Value' it will store the two terminal values, for 'Choice' the two different Tags names, for 'Length' the two lengths and for 'Order' the matches.

The function **Identify_Children_Matches** identifies a match between children of different repetition nodes. The similitude between each child from $node_a$ and $node_b$ (with n and m children respectively) is represented as a matrix, M, of size n x m where:

Table 1. Partial paths obtained from Fig.3 (d)

Partial Paths	Occurrences
Header.0.'Reply'	1
~~Header.0.'Reply'.'Reply'~~	1
Header.1.?	2
~~Header.1.?.SP~~	2
~~Header.1.?.SP.%x20~~	2
~~Header.1.?.SP.%x20.' '~~	2
Header.2.Method.?	4
~~Header.2.Method.?.ALPHA.~~	4
Header.2.Method.?.ALPHA.%x41-5A	4
Header.2.Method.?.ALPHA.%x41-5A.'U'	1
Header.2.Method.?.ALPHA.%x41-5A.'S'	1
Header.2.Method.?.ALPHA.%x41-5A.'E'	1
Header.2.Method.?.ALPHA.%x41-5A.'R'	1

(~~strikethrough~~) Strikethrough paths are the ones considered as cosmetics.
(?) Quotes define that the current path may be any of the repetition items.

Algorithm 1. Node differences Location

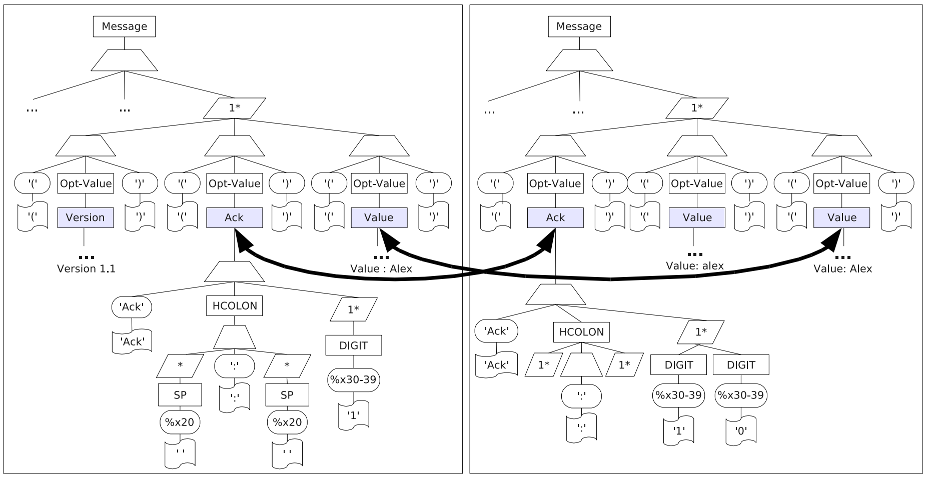
```
procedure NODEDIFF($node_a, node_b$)
    if $Tag(node_a) = Tag(node_b)$ then
        if $Type(node_a) = TERMINAL$ then
            if $Value(node_a) \; != \; Value(node_b)$ then
                $Report\_Difference('Value', node_a, node_b)$
            end if
        else if $Type(node_a) = NON-TERMINAL$ then
            $NODEDIFF(node_a.child_0, node_b.child_0))$        ▷Non_Terminals have
                                                              ▷an unique child
        else if $Type(node_a) = SEQUENCE$ then
            for $i = 1..\#node_a$ do                          ▷In a Sequence
                $NODEDIFF(node_a.child_i, node_b.child_i)$    ▷$\#node_a = \#node_b$
            end for
        else if $Type(node_a) = REPETITION$ then
            if not $(\#node_a = \#node_b)$ then
                $Report\_Difference('Length', node_a, node_b)$
            end if
            $matches := Identify\_Children\_Matches(node_a, node_b)$
            if $\exists \; (i, j) \in matches : \; i \; != \; j$ then
                $Report\_Difference('Order', node_a, node_b)$
            end if
            forall $(i, j) \in matches$ do
                $NODEDIFF(node_a.child_i, node_b.child_j)$
            end for
        end if
    else
        $Report\_Difference('Choice', node_a, node_b)$
    end if
end procedure
```

$$M_{i,j} = resemblance(node_a.child_i, node_b.child_j)$$

To find the most adequate match, a greedy matching assignment based on the concept of Nash Equilibrium [5] is used. Children with the biggest similarity are bound. If a child from $node_a$ shares the same similarity score with more than one child from $node_b$, some considerations have to be added respecting their position in the repetitions.

Figure 4 illustrates an example match, assuming that the following matrix was obtained using the *Resemblance* method with the path "`Message.2.?`". The rows in the matrix represent the children from the subtree in (a) and the columns the children from subtree (b).

$$M = \begin{pmatrix} .00 & .00 & .00 \\ .33 & .00 & .00 \\ .00 & .61 & .90 \end{pmatrix}$$



(a) Message Number 2 of Figure 1 (b)

(b) Message Number 2 of Figure 1 (c)

Partial Path	Type	Node$_a$	Node$_b$
ACK.1.HCOLON.0	Static Length	1	0
ACK.1.HCOLON.2	Static Length	1	0
ACK.2	Static Length	1	2
ACK.2.(?).DIGIT.%x30-39	Static Value	'1'	'1'
ACK.2.(?).DIGIT.%x30-39	Static Value		'0'

(c) Node Signatures from the **Ack** nodes

$S_A \cup S_B$	$S_A \cap S_B$	$\frac{S_A \cap S_B}{S_A \cup S_B}$
1	0	0
1	0	0
2	1	0.5
1	1	1
1	0	0
6	2	0.33

(d) Shingle information from the **Ack** nodes where $S_a = S(N_a, 1)$ and $S_b = (N_b, 1)$

Fig. 4. Performed match between sub-branches of the tree

All the compared children share some common items besides the choice nodes (colored). Those common items are *Cosmetics* nodes, which are required in the message in order to be compliant with the grammar. Note that, besides the *Cosmetic* fields, the first item of the subtree (a) does not share any similarity with any of the other nodes. It should therefore not match any other node.

4 Structural Features Extraction

4.1 Fields Classification

One major activity that was not yet described is how non-invariant fields are identified. The process is done by using all the messages coming from one device and finding the differences between each two messages using Algorithm 1. For each result, a secondary algorithm (Algorithm 2) is run in order to fine tune the extracted classification.

Algorithm 2. Fields Classification Algorithm

```
procedure FieldClassification($differences_{a,b}$)
    forall $diff \in differences_{a,b}$ do
        if $diff.type ==' Value')$ then
            $Classify\_as\_Dynamic('Value', diff.path)$
        else if $diff.type ==' Choice'$ then
            $Classify\_as\_Dynamic('Choice', diff.path)$
        else if $diff.type ==' Length'$ then
            $Classify\_as\_Dynamic('Length', diff.path)$
        else if $diff.type ==' Order'$ then
            if not $(\forall\ (i,j),(x,z) \in diff.matches :$
                    $(i < x \wedge j < z) \vee (i > x \wedge j > z))$ then
                ▷Check if a permutation exists between the matched items.
                $Classify\_as\_Dynamic('Order', diff.path)$
            end if
        end if
    end for
end procedure
```

The **Classify_as_Dynamic** functions store in the global list, **fieldClassifications**, a tuple with the type of the found difference (e.g. 'Value', 'Choice', 'Length' or 'Order') and the partial path in the tree structure that represents the node in the message.

This algorithm recognizes only the fields that are *Dynamic*. The set of *Static* fields will be represented by the union of all the fields not recognized as *Dynamic*.

Assuming a training set *Msg_set*, of messages compliant with the grammar as

$$Msg = \bigcup_{i=0}^{n} msg\_set_i$$

where n is quantity of devices and msg_set_i is the set of messages generated by device i, the total number of comparisons computed in this process is

$$cmps_1 = \sum_{i=0}^{n} \frac{|msg\_set_i| * (|msg\_set_i| - 1)}{2} \qquad (2)$$

4.2 Features Recognition

Some features are essential for an inter-device classification. In contrast to the Fields Classification, this process compares all the messages from the training set sourced from different devices. All the *Invariant* Fields -for which different implementations have different values- are identified. Algorithm 3 recognizes these features. Its inputs are the *fieldClassifications* computed by the Algorithm 2, the Devices Identifier to which the compared message belongs as well as the set of differences found by Algorithm 1 between the messages.

Algorithm 3. Features Recognition Algorithm

```
procedure featuresRecognition($fieldClassifications,\ DevID_{a,b},\ differences_{a,b}$)
    forall $diff \in differences_{a,b}$ do
        if not $(diff.type, diff.path) \in fieldClassifications$ then
            if $diff.type == 'Value'$ then
                $addFeature('Value',\ diff.path,\ DevID_{a,b},\ diff.value_{a,b})$
            else if $diff.type == 'Choice'$ then
                $addFeature('Choice',\ diff.path,\ DevID_{a,b},\ diff.name_{a,b})$
            else if $diff.type == 'Length'$ then
                $addFeature('Length',\ diff.path,\ DevID_{a,b},\ diff.length_{a,b})$
            else if $diff.type == 'Order'$ then
                if $(\exists\ (x,z) \in diff.matches : x \neq z)$ then
                    $addFeature('Order',\ diff.path,\ DevID_{a,b},$
                                    $diff.match,\ diff.children\_nodes_{a,b})$
                end if
            end if
        end if
    end for
end procedure
```

The **add_Feature** function stores in a global variable, **recognizedFeatures**, the partial path of the node associated with the type of difference (i.e. Value, Name, Order or Length) and a list of devices with their encountered value. However, the 'Order' feature presents a more complex approach, requiring minor improvements.

Assuming the earlier *Msg_set* set, this process will do the following number of comparisons:

$$cmps_2 = \sum_{i=0}^{n} |msg\_set_i| * \sum_{j=i+1}^{n} |msg\_set_j| \tag{3}$$

From the *recognizedFeatures* only the *Static* fields are used. The recognized features define a sequence of items, where each one represents the field location path in the tree representation and a list of Device ID with their associated value.

The Recognized Features can be classified in:

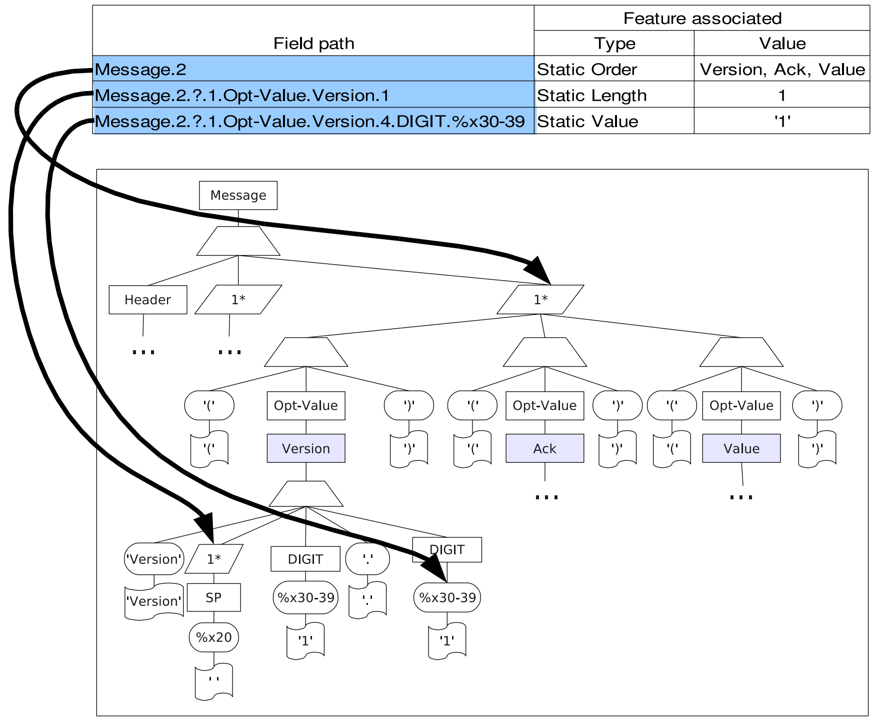
– Features that were found with each device and at least two distinct values are observed for a pair of devices,
– Features that were found in some of the devices for which such a location path does not exists in messages from other implementations.

4.3 Fingerprinting

The classification of a message uses the tree structure representation introduced in section 3.1. The set of recognized features obtained in section 4 represents all the partial paths in a tree structure that are used for the classification process.

In some cases, the features are of type *'Value'*, *'Choice'* or *'Length'*. Their corresponding value is easily obtained. However, the case of an *'Order'* represents a more complex approach, requiring some minor improvements

Figure 5 illustrates some identified features for an incoming message.



Field path	Feature associated	
	Type	Value
Message.2	Static Order	Version, Ack, Value
Message.2.?.1.Opt-Value.Version.1	Static Length	1
Message.2.?.1.Opt-Value.Version.4.DIGIT.%x30-39	Static Value	'1'

Fig. 5. Features Identification

Once a set of distinctive features is obtained, some well known classification techniques can be leveraged to implement a classifier. In our work, we have leveraged the machine learning technique described in [6].

5 Experimental Results

We have implemented the Fingerprinting Framework approach in Python. A scannerless Generalized Left-to-right Rightmost (GLR) derivation parser has been used (Dparser[7]) in order to solve ambiguities in the definition of the grammar. The training function could easily be parallelized.

We have instantiated the fingerprinting approach on the SIP protocol. The SIP messages are sent in clear text (ASCII) and their structure is inspired from HTTP. Several primitives - REGISTER, INVITE, CANCEL, BYE and OPTIONS - allow session management for a voice/multimedia call. Some additional extensions do also exist -INFO, NOTIFY, REFER, PRACK- which allow the support of presence management, customization, vendor extensions etc.

We have captured 21827 SIP messages from a real network, summarized in Table 2.

Table 2. Tested equipment

Device	**Software/Firmware version**
Asterisk	v1.4.4
Cisco CallManager	v5.1.1
Cisco 7940/7960	vP0S3-08-7-00
	vP0S3-08-8-00
Grandstream Budge Tone-200	v1.1.1.14
Linksys SPA941	v5.1.5
Thomson ST2030	v1.52.1
Thomson ST2020	v2.0.4.22
SJPhone	v1.60.289
	v1.60.320
	v1.65
Twinkle	v0.8.1
	v0.9
Snom	v5.3
Kapanga	v0.98
X-Lite	v3.0
Kphone	v4.2
3CX	v1.0
Express Talk	v2.02
Linphone	v1.5.0
Ekiga	v2.0.3

The system was trained with only 12% of the 21827 messages. These messages were randomly sampled. However, a proportion between the number of collected messages and the number used for the training was kept; they ranged from 50 to 350 messages per device. Table 3 shows the average and total time obtained for the comparisons of each training phase and for the message classification process (i.e. message fingerprinting). During both Phase 1 and 2, the comparisons were distributed over 10 computers ranging from Pentium IV to Core Duo. As it was expected, the average comparison time per message in Phase 2 was lower than in the previous phase, since only the invariant fields are compared. To evaluate the training, the system classified all the sampled messages (i.e. 21927 messages) in only in one computer (Core Duo @ 2.93GHz).

Table 3. Performance results obtained with the system

Type of Action	Average time per action	Number of actions computed	Total computed time
Msg. comparisons for Phase 1	632 milisec	296616	5 hours(1)
Msg. comparisons for Phase 2	592 milisec	3425740	56 hours(1)
Msg. classification	100 milisec.	21827	40 minutes

(1) Computed time using 10 computers.

172 features were discovered among all the different types of messages. These features represent items order, different lengths and values of fields where non protocol knowledge except its syntax grammar had been used. Between two different devices the distance of different features ranges between 26 to 95 features, where most of the lower values correspond to different versions of the same device. Usually, up to 46 features are identified in one message.

Table 4 summarizes the sensitivity, specificity and accuracy. The results were obtained using the test data set.

Table 4. Accuracy results obtained with the system

Classification	**True Positive** 18881	**False Positive** 20	**Positive Predictive Value** 0.998
	False Negative 2909	**True Negative** 435780	**Negative Predictive Value** 0.993
	Sensitivity 0.866	**Specificity** 0.999	**Accuracy** 0.993

In this table we can observe that the results are very encouraging due to the high specificity and accuracy. However, some observations can be made about the quantity of false negatives. About 2/5 of them belong to only one implementation (percentage that represents 50% of its messages), 2/5 belongs to 3 more device classes (representing 18% of their messages), the final 1/5 belongs to 8 classes (representing 10% of their messages) and the 7 classes left do not have false negatives. This issue can be a consequence of the irregularity in the quantity from the set of messages in each device. Three of the higher mentioned classes had been used in our test-bed to acquire most features of SIP. A second explanation can be that in fact many of those messages do not contain valuable information (e.g. intermediary messages). Table 5 shows all the 38 types of messages collected in our test with information concerning their miss-classification (i.e. False Negatives).

Finally, we created a set of messages which have been manually modified. These modifications include changing the User-Agent, Server-Agent and references to device name. As a result, deleting a few such fields did not influence

Table 5. False Negative classification details

Type of Message	False Negatives	Message quantity	Miss percentage
`200, 100, ACK`	1613 (710, 561, 347)	9358 (4663, 1802, 2893)	17% (15%, 31%, 11%)
`501, 180, 101` `BYE, 486`	824 $\begin{pmatrix} 257, 215, 148 \\ 104, 100 \end{pmatrix}$	3414 $\begin{pmatrix} 385, 1841, 148 \\ 892, 176 \end{pmatrix}$	24% $\begin{pmatrix} 65\%, 11\%, 100\% \\ 11\%, 67\% \end{pmatrix}$
`489, 487, 603` `202, 480, 481` `380, 415, 400`	213 $\begin{pmatrix} 84, 57, 28 \\ 21, 13, 6 \\ 2, 1, 1 \end{pmatrix}$	636 $\begin{pmatrix} 84, 230, 118 \\ 52, 42, 18 \\ 2, 38, 51 \end{pmatrix}$	33% $\begin{pmatrix} 100\%, 24\%, 23\% \\ 40\%, 30\%, 33\% \\ 100\%, 2\%, 2\% \end{pmatrix}$
`INVITE, OPTIONS` `REGISTER, CANCEL` `SUBSCRIBE`	117 $\begin{pmatrix} 38, 34 \\ 25, 19 \\ 1 \end{pmatrix}$	5694 $\begin{pmatrix} 3037, 628 \\ 1323, 297 \\ 409 \end{pmatrix}$	2% $\begin{pmatrix} 1\%, 5\% \\ 1\%, 6\% \\ .00\% \end{pmatrix}$
`INFO, REFER` `PRACK, NOTIFY` `PUBLISH`	0	2223 $\begin{pmatrix} 1830, 163 \\ 117, 77 \\ 36 \end{pmatrix}$	0%
11 other Response Codes	0	492	0%

the decision of the system; neither did it changing their banners to another implementation name. However, as more modifications were done, less precise the system became and more mistakes were done.

6 Related Work

Fingerprinting became a popular topic in the last few years. It started with the pioneering work of Comer and Lin [8] and is currently an essential activity in security assessment tasks. Some of the most known network fingerprinting operations are done by NMAP [9], using a set of rules and active probing techniques. Passive techniques became known mostly with the P0F [10] tool, which is capable to do OS fingerprinting without requiring active probes. Many other tools like (AMAP, XProbe, Queso) did implement similar schemes.

Application layer fingerprinting techniques, specifically for SIP, were first described in [11,12]. These approaches proposed active as well as passive fingerprinting techniques. Their common baseline is the lack of an automated approach for building the fingerprints and constructing the classification process. Furthermore, the number of signatures described are minimal which leaves the systems easily exposed to approaches as the one described by D. Watson et al. [13], that can fool them by obfuscation of such observable signatures. Recently, the work by J. Caballero et al. [6] described a novel approach for the automation of Active Fingerprint generation which resulted in a vast set of possible signatures. It is one of the few automatic approaches found in the literature and it is based in finding a set of queries (automatically generated) that identify different responses in the different implementations. While our work addresses specifically the automation for passive fingerprinting, we can imagine this two complementary approaches working together.

There have been recently similar efforts done in the research community aiming however at a very different goal from ours. These activities started with practical reverse engineering of proprietary protocols [14] and [15] and a simple application of bioinformatics inspired techniques to protocol analysis [16]. These initial ideas matured and several other authors reported good results of sequence alignment techniques in [17], [18], [19] and [20]. Another major approach for the identification of the structure in protocol messages is to monitor the execution of an endpoint and identify the relevant fields using some tainted data [21], [22]. Recently, work on identifying properties of encrypted traffic has been reported in [23,24]. These two approaches used probabilistic techniques based on packet arrivals, interval, packet length and randomness in the encrypted bits to identify Skype traffic or the language of conversation. While all these complementary works addressed the identification of the protocol building blocks or properties in their packets, we assumed a known protocol and worked on identifying specific implementation stacks.

The closest approach to ours, in terms of message comparison, it is the work developed by M. Chang and C. K.Poon [25] for collection training SPAM detectors. However, in their approach as they focus in identifying human written sentences, they only consider the lexical analysis of the messages and do not exploit an underlying structure.

Finally, two other solutions have been proposed in the literature in this research landscape. Flow based identification has been reported in [26], while a grammar/ probabilistic based approach is proposed in [27] and respectively in [28].

7 Conclusions

In this article we described a novel approach for generating fingerprinting systems based on the structural analysis of protocol messages. Our solution automates the generation by using both formal grammars and collected traffic traces. It detects important and relevant complex tree like structures and leverages them for building fingerprints. The applicability of our solution lies in the field of intrusion detection and security assessment, where precise device/service/stack identification are essential. We have implemented a SIP specific fingerprinting system and evaluated its performance. The obtained results are very encouraging. Future work will consist in improving the method and applying it to other protocols and services. Our work is relevant to the tasks of identifying the precise vendor/device that has generated a captured trace. We do not address the reverse engineering of unknown protocols, but consider that we know the underlying protocol. The current approach does not cope with cryptographically protected traffic. A straightforward extension for this purpose is to assume that access to the original traffic is possible. Our main contribution consists in a novel solution to automatically discover the significant differences in the structure of protocol compliant messages. We will extend our work towards the natural evolution, where the underlying grammar is unknown.

The key idea is to use a structural approach, where formal grammars and collected network traffic are used. Features are identified by paths and their associated values in the parse tree. The obtained results of our approach are very good. This is due to the fact that a structural message analysis is performed. Most existing fingerprinting systems are built manually and require a long lasting development process.

References

1. Rosenberg, J., Schulzrinne, H., Camarillo, G., Johnston, A., Peterson, J., Sparks, R., Handley, M., Schooler, E.: SIP: Session Initiation Protocol (2002)
2. Crocker, D.H., Overell, P.: Augmented BNF for Syntax Specifications: ABNF (1997)
3. Buttler, D.: A Short Survey of Document Structure Similarity Algorithms. In: Arabnia, H.R., Droegehorn, O. (eds.) International Conference on Internet Computing, pp. 3–9. CSREA Press (2004)
4. Broder, A.Z.: On the Resemblance and Containment of Documents. In: SEQUENCES 1997: Proceedings of the Compression and Complexity of Sequences 1997, Washington, DC, USA, p. 21. IEEE Computer Society, Los Alamitos (1997)
5. Nash, J.F.: Non-Cooperative Games. The Annals of Mathematics 54(2), 286–295 (1951)
6. Caballero, J., Venkataraman, S., Poosankam, P., Kang, M.G., Song, D., Blum, A.: FiG: Automatic Fingerprint Generation. In: The 14th Annual Network & Distributed System Security Conference (NDSS 2007) (February 2007)
7. DParser, http://dparser.sourceforge.net/
8. Comer, D., Lin, J.C.: Probing TCP Implementations. In: USENIX Summer, pp. 245–255 (1994)
9. Nmap, http://www.insecure.org/nmap/
10. P0f, http://lcamtuf.coredump.cx/p0f.shtml
11. Yan, H., Sripanidkulchai, K., Zhang, H.: Incorporating Active Fingerprinting into SPIT Prevention Systems. In: Third Annual VoIP Security Workshop (June 2006)
12. Scholz, H.: SIP Stack Fingerprinting and Stack Difference Attacks. Black Hat Briefings (2006)
13. Watson, D., Smart, M., Malan, G.R., Jahanian, F.: Protocol scrubbing: network security through transparent flow modification. IEEE/ACM Trans. Netw. 12(2), 261–273 (2004)
14. Open Source FastTrack P2P Protocol (2007), http://gift-fasttrack.berlios.de/
15. Fritzler, A.: UnOfficial AIM/OSCAR Protocol Specification (2007), http://www.oilcan.org/oscar/
16. Beddoe, M.: The Protocol Informatics Project. Toorcon (2004)
17. Gopalratnam, K., Basu, S., Dunagan, J., Wang, H.J.: Automatically Extracting Fields from Unknown Network Protocols. In: Systems and Machine Learning Workshop 2006 (2006)
18. Wondracek, G., Comparetti, P.M., Kruegel, C., Kirda, E.: Automatic Network Protocol Analysis. In: Proceedings of the 15th Annual Network and Distributed System Security Symposium (NDSS 2008) (2008)

19. Newsome, J., Brumley, D., Franklin, J., Song, D.: Replayer: automatic protocol replay by binary analysis. In: CCS 2006: Proceedings of the 13th ACM conference on Computer and communications security, pp. 311–321. ACM, New York (2006)
20. Cui, W., Kannan, J., Wang, H.J.: Discoverer: automatic protocol reverse engineering from network traces. In: SS 2007: Proceedings of 16th USENIX Security Symposium on USENIX Security Symposium, Berkeley, CA, USA, pp. 1–14. USENIX Association (2007)
21. Brumley, D., Caballero, J., Liang, Z., Newsome, J., Song, D.: Towards automatic discovery of deviations in binary implementations with applications to error detection and fingerprint generation. In: SS 2007: Proceedings of 16th USENIX Security Symposium on USENIX Security Symposium, Berkeley, CA, USA, pp. 1–16. USENIX Association (2007)
22. Lin, Z., Jiang, X., Xu, D., Zhang, X.: Automatic Protocol Format Reverse Engineering through Context-Aware Monitored Execution. In: 15th Symposium on Network and Distributed System Security (2008)
23. Bonfiglio, D., Mellia, M., Meo, M., Rossi, D., Tofanelli, P.: Revealing skype traffic: when randomness plays with you. SIGCOMM Comput. Commun. Rev. 37(4), 37–48 (2007)
24. Wright, C.V., Ballard, L., Monrose, F., Masson, G.M.: Language identification of encrypted VoIP traffic: Alejandra y Roberto or Alice and Bob? In: SS 2007: Proceedings of 16th USENIX Security Symposium on USENIX Security Symposium, Berkeley, CA, USA, pp. 1–12. USENIX Association (2007)
25. Chang, M., Poon, C.K.: Catching the Picospams. In: Hacid, M.-S., Murray, N.V., Raś, Z.W., Tsumoto, S. (eds.) ISMIS 2005. LNCS (LNAI), vol. 3488, pp. 641–649. Springer, Heidelberg (2005)
26. Haffner, P., Sen, S., Spatscheck, O., Wang, D.: ACAS: automated construction of application signatures. In: MineNet 2005: Proceedings of the 2005 ACM SIGCOMM workshop on Mining network data, pp. 197–202. ACM, New York (2005)
27. Borisov, N., Brumley, D.J., Wang, H.J.: Generic Application-Level Protocol Analyzer and its Language. In: 14th Symposium on Network and Distributed System Security (2007)
28. Ma, J., Levchenko, K., Kreibich, C., Savage, S., Voelker, G.M.: Unexpected means of protocol inference. In: IMC 2006: Proceedings of the 6th ACM SIGCOMM conference on Internet measurement, pp. 313–326. ACM, New York (2006)

On Evaluation of Response Cost for Intrusion Response Systems
(Extended Abstract)

Natalia Stakhanova[2], Chris Strasburg[1], Samik Basu[1], and Johnny S. Wong[1]

[1] Department of Computer Science, Iowa State University, USA
{cstras,sbasu,wong}@cs.iastate.edu
[2] Faculty of Computer Science, University of New Brunswick, Canada
natalia@unb.ca,
nStakhanova@gmail.com

Abstract. In this work we present a structured and consistent methodology for evaluating cost of intrusion responses. The proposed approach provides consistent basis for response evaluation across different systems while incorporating security policy and properties of specific system environment. The advantages of the proposed cost model were evaluated via simulation process.

The proliferation of complex and fast-spreading intrusions against computer systems brought new requirements to intrusion detection and response, demanding the development of sophisticated and automated intrusion response systems. In this context, the cost-sensitive intrusion response models have gained the most interest mainly due to their emphasis on the balance between potential damage incurred by the intrusion and cost of the response. However, one of the challenges in applying this approach is defining consistent and adaptable measurement of these cost factors on the basis of policy of the system being protected against intrusions.

We developed a structured and consistent methodology for the evaluation of intrusion response cost based on three parameters: (a) *the impact of a response on the system* that quantifies the negative effect of the response on the system resources, (b) *the response goodness* that measures the ability of the corresponding response to mitigate damage caused by the intrusion to the system resources and (c) *the operational cost of a response in a given environment*.

Within this methodology, we assess response impact with respect to resources of the affected system. Our model takes into account the relative importance of the system resources determined through the review of the system policy goals according to the following categories: *confidentiality*, *availability* and *integrity*. One of the important steps in this process is the analysis of the system resources. Based on this analysis, the evaluation algorithm assesses the *response goodness* in terms of the resources protected by the response, the *response damage* in terms of the resources impaired by the action and the *operational cost* with respect to its environmental impact.

This methodology does not substitute the response selection process in case of detected intrusion, but rather allows to evaluate the available responses on the consistent basis. The proposed methodology includes the following steps:

1. **The system classification:** The first step in quantifying the cost of a response involves determining the characteristics of the computing environment where the response will be deployed which includes evaluating system security policy priorities, defining level of tolerable risk, etc.
2. **The system policy goals:** The next step is to determine the importance of the system policy goals, and subsequently, to assess the potential risks according to the following categories: *confidentiality*, *availability* and *integrity*.
3. **The system resources:** System resources can be broadly viewed as the system assets (e.g., host, network, etc.), services provided by the system (e.g., HTTP, file system) and users served by the system. The analysis of system resources includes the enumeration of the available resources and their classification according to the importance for the system policy goals.
4. **The intrusion responses:** The responses are deployed to either counter possible attacks and defend the system resources or regain secure system state. Thus, the selection of applicable responses primarily depends on the identified system resources.
5. **The response operational cost:** The assessment of operational cost is generally independent from the system policy. We assess the involved operational expenses on the basis of three requirements: *human resources* , i.e., administrator time, *system resources*, i.e., storage, network bandwidth, processor time, etc., and *direct expenses* i.e., data processing fees by a third party, subscription service fees, cost of components replacement, etc.
6. **The response goodness:** Often the detection mechanism of the intrusion detection system (IDS) provides administrators with a set of alerts indicating potential attacks rather than a specific intrusion. When this situation arises, the response needs to be deployed preemptively on the basis of high likelihood of possible intrusions. In these cases, the response goodness is evaluated based on the number of possible intrusions it can potentially address, and consequently, the number of resources that can be protected by the response.
7. **The response impact on the system:** The impact of a response on the system is evaluated based on the defined system goals and their importance. The impact assessment process for a specific response includes three steps: (1) identifying the system resources affected by each response, (2) for each resource determining the priority of responses based on their effect on the resource, and (3) computing the negative impact of the responses on the associated resource using the ordering obtained in step 2. Eventually, the impact of a response on the system as a whole is an aggregation of the response's impact on the system resources.

The proposed methodology for assigning response costs essentially presents the first roadmap for defining *standardized metrics* for response cost evaluation. These response metrics provide a *consistent basis* for evaluation across systems, while allowing the response cost to be *adapted with respect to the security policy and properties of specific system environment*. Importantly, this approach is *practically implementable* in a real-world environment, making response cost assessment accessible to system administrators with a range of system expertise.

WebIDS: A Cooperative Bayesian Anomaly-Based Intrusion Detection System for Web Applications (Extended Abstract)

Nathalie Dagorn

Laboratory of Algorithmics, Cryptology and Security (LACS), Luxembourg
& ICN Business School, France
nathalie.dagorn@icn-groupe.fr,
nathalie.dagorn@orange.fr
http://www.uni.lu/
http://www.icn-groupe.fr/

Abstract. This paper presents WebIDS, a learning-based anomaly detection system for Web applications aiming at improving the decision process, reducing the number of false positives, and achieving distributed detection.

Keywords: Anomaly detection, Correlation, Web application.

1 Introduction

Attacks on Web applications and services have been increasing dramatically for the last years. Related approaches in intrusion detection are still rare. The major challenges anomaly-based systems have to solve in the field are the improvement of the decision process, the reduction of the high number of (false) alarms caused by unusual activities, and the recent need of distributed intrusion detection. At the crossing of these research areas, the aim of our work is to propose an efficient distributed anomaly detection system dedicated to the security of Web applications.

2 Our Proposal: WebIDS

WebIDS analyzes HTTP GET requests as logged by Apache Web servers. The analysis process is based on a multi-model approach [5] implementing ten statistical algorithms: attribute length, attribute character distribution, structural inference, token finder, attribute presence or absence, attribute order, access frequency, inter-request delay, invocation order, and anomaly history (which allows, among others, keeping track of alarms). The system requires no special configuration (autonomous learning). A non-naive Bayesian network is used as a decision process [3], classifying the events more accurately and incorporating information about confidence in the models. At the root node, a specification of the event classification [6] distinguishes between a normal state and five Web attack states (authentication, XSS, command execution, denial of service, and other attack). The system is improved after each log analysis by filtering out false positives using an alarm clustering technique [2]. As part of the anomaly history model, a cooperation feature enables the system to achieve alarm and event correlation [4]. The Intrusion Detection Message Exchange Format (IDMEF) [1] is used for sharing alarm information between systems.

3 Experimental Results

WebIDS has been implemented in an IT company based in Luxembourg and showed good detection rates (sensitivity of 96.02 %, specificity of 99.99 %, and reliability of 99.94 %). The false positive rate (0.01422 %) is lower than the rates observed for similar systems. Nevertheless, these results must be mitigated because only a small number of anomalies could be observed by WebIDS over the experimental period, and the comparison with existing systems is not based on the same dataset.

4 Conclusion and Future Work

As a conclusion, we can state that the cooperative anomaly-based intrusion detection system proposed is both innovative and efficient. By improving the decision process, reducing the false positive rate and enabling cooperation between systems, it meets the defined challenges. As a follow-up to this research, the deployment of WebIDS in a more widely distributed environment is currently considered. Some functional and technical improvements are being carried out for that purpose.

References

1. Debar, H., Curry, D., Feinstein, B.: The Intrusion Detection Message Exchange Format. Internet Draft IETF (2005), `http://www.ietf.org/internet-drafts/draft-ietf-idwg-idmef-xml-14.txt`
2. Julisch, K.: Using Root Cause Analysis to Handle Intrusion Detection Alarms. PhD Thesis, University of Dortmund, Germany (2003)
3. Kruegel, C., Mutz, D., Robertson, W., Valeur, F.: Bayesian Event Classification for Intrusion Detection. In: 19th Annual Computer Security Applications Conference. IEEE Computer Society Press, New York (2003)
4. Kruegel, C., Valeur, F., Vigna, G.: Intrusion Detection and Correlation - Challenges and Solutions. In: Advances in Information Security, vol. 14. Springer, Heidelberg (2005)
5. Kruegel, C., Vigna, G., Robertson, W.: A Multi-Model Approach to the Detection of Web-Based Attacks. Computer Networks 48(5), 717–738 (2005)
6. Valdes, A., Skinner, K.: Adaptive, Model-Based Monitoring for Cyber Attack Detection. In: 3rd International Symposium on Recent Advances in Intrusion Detection, pp. 80–92. Springer, Heidelberg (2000)

Evading Anomaly Detection through Variance Injection Attacks on PCA (Extended Abstract)

Benjamin I.P. Rubinstein[1], Blaine Nelson[1], Ling Huang[2], Anthony D. Joseph[1,2], Shing-hon Lau[1], Nina Taft[2], and J. D. Tygar[1]

[1] UC Berkeley
{benr,nelsonb,adj}@cs.berkeley.edu, mrvulcanpaypal@gmail.com
[2] Intel Research, Berkeley
hling@cs.berkeley.edu, nina.taft@intel.com

Abstract. Whenever machine learning is applied to security problems, it is important to measure vulnerabilities to adversaries who poison the training data. We demonstrate the impact of variance injection schemes on PCA-based network-wide volume anomaly detectors, when a single compromised PoP injects chaff into the network. These schemes can increase the chance of evading detection by sixfold, for DoS attacks.

1 Motivation and Problem Statement

We are broadly interested in understanding vulnerabilities associated with using machine learning in decision-making, specifically how adversaries with even limited information and control over the learner can subvert the decision-making process [1]. An important example is the role played by machine learning in dynamic network anomography, the problem of inferring network-level Origin-Destination (OD) flow anomalies from aggregate network measurements. We ask, can an adversary generate OD flow traffic that misleads network anomography techniques into misclassifying anomalous flows? We show the answer is yes for a popular technique based on Principal Components Analysis (PCA) from [2].

The detector operates on the $T \times N$ link traffic matrix $\mathbf{Y}$, formed by measuring N link volumes between PoPs in a backbone network, over T time intervals. Figure 1 depicts an example OD flow within a PoP-to-PoP topology. Lakhina et al. observed that the rows of the normal traffic in $\mathbf{Y}$ lie close to a low-dimensional subspace captured by PCA using $K = 4$ principal components. Their detection method involves projecting the traffic onto this normal K-dimensional subspace; large (small) residuals, as compared with the Q-statistic, are called *positive* anomalies (*negative* normal traffic).

2 Results and Future Work

Consider an adversary launching a DoS attack on flow f in week w. Poisoning aims to rotate PCA's K-dimensional subspace so that a false negative (FN) occurs during the attack. Our *Week-Long* schemes achieve this goal by adding high variance chaff at the compromised origin PoP, along f, throughout week $w - 1$.

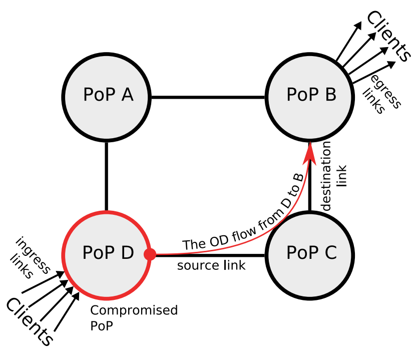


Fig. 1. Point-of-presence (PoP)-level granularity in a backbone network, and the links used for data poisoning

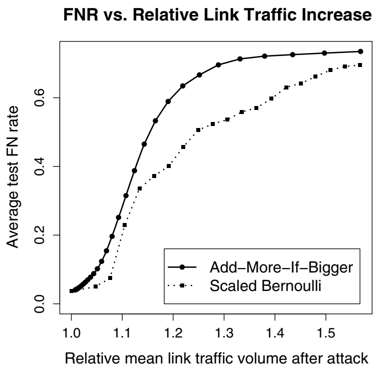


Fig. 2. *Week-Long* attacks: test FNRs are plot against the relative increase to the mean link volumes for the attacked flow

Figure 2 presents results for two chaff methods. Both methods add chaff c_t to each link in f at time t, depending on parameter θ: *Scaled Bernoulli* selects c_t uniformly from $\{0, \theta\}$; *Add-More-If-Bigger* adds $c_t = (y_o(t) - \overline{y_o})^\theta$ where $y_o(t)$ and $\overline{y_o}$ are the week $w - 1$ origin link traffic at time t and average origin link traffic, respectively. We evaluated these methods on data from Abilene's backbone network of 12 PoPs. For each week 2016 measurements were taken, each averaged over 5 minute intervals, for each of 54 virtual links—15 bi-directional inter-PoP links and the PoPs' ingress & egress links.

The attacker's chance of evasion is measured by the FN rate (FNR). We see that the *Add-More-If-Bigger* chaff method, which exploits information about origin link traffic, achieves greater FNR increases compared to *Scaled Bernoulli*. The baseline FNR of 4% for PCA on clean data, can be doubled by adding on average only 4% additional traffic to the links along the poisoned flow. The FNR can be increased sixfold to 24% via an average increase of 10% to the poisoned link traffic. In our *Boiling Frog* strategies, where poisoning is increased slowly over several weeks, a 50% chance of successful evasion can be achieved with a modest 5% volume increase from week-to-week over a 3 week period [3].

We have verified that simply increasing the number of principal components is not useful in protecting against our attacks [3]. In future work we will evaluate counter-measures based on Robust formulations of PCA, and will devise poisoning strategies for increasing PCA's false positive rate.

References

1. Barreno, M., Nelson, B., Joseph, A.D., Tygar, J.D.: The security of machine learning. Technical Report UCB/EECS-2008-43, UC Berkeley (April 2008)
2. Lakhina, A., Crovella, M., Diot, C.: Diagnosing network-wide traffic anomalies. In: Proc. SIGCOMM 2004, pp. 219–230 (2004)
3. Rubinstein, B.I.P., Nelson, B., Huang, L., Joseph, A.D., Lau, S., Taft, N., Tygar, J.D.: Compromising PCA-based anomaly detectors for network-wide traffic. Technical report UCB/EECS-2008-73, UC Berkeley (May 2008)

Anticipating Hidden Text Salting in Emails (Extended Abstract)

Christina Lioma[1], Marie-Francine Moens[1], Juan-Carlos Gomez[1], Jan De Beer[1,⋆], Andre Bergholz[2], Gerhard Paass[2], and Patrick Horkan[3]

[1] Katholieke Universiteit Leuven, Belgium
{christina.lioma,sien.moens,juancarlos.gomez}@cs.kuleuven.be,
jan.debeer@be.ibm.com
[2] Fraunhofer IAIS, Germany
{andre.bergholz,gerhard.paass}@ais.fraunhofer.de
[3] Symantec, Ireland
patrick_horkan@symantec.com

Abstract. Salting is the intentional addition or distortion of content, aimed to evade automatic filtering. Salting is usually found in spam emails. Salting can also be hidden in phishing emails, which aim to steal personal information from users. We present a novel method that detects hidden salting tricks as visual anomalies in text. We solely use these salting tricks to successfully classify emails as phishing (F-measure >90%).

1 Introduction

Given a text and a user who reads this text, *hidden text salting* is any modification of text content that cannot be seen by the user, e.g., text written in invisible colour, or in zero font size. Hidden text salting can be applied to any medium and content genre, e.g. emails or MMS messages, and can be common in fraudulent *phishing* emails [3]. We present a novel method for detecting hidden text salting and using it to recognise phishing emails.

Related research has focused on filtering email spam. Early spam filters used human-coded ad-hoc rules, often optimized by machine learning, e.g. spamassassin. Such filters were easy to fool, hard to maintain, and outperformed by filters using visual features, e.g. embedded text in images [4]. Recently, statistical data compression has been used to build separate models for compressed ham and spam, and then classify emails according to which model they fit better when compressed [2]. None of these studies addresses hidden salting directly.

2 Hidden Text Salting Detection

Given an email as input, a text production process, e.g. a Web browser, creates an internal parsed representation of the email text and drives the rendering of that representation onto some output medium, e.g. a browser window. We tap into this rendering process to detect hidden content (= manifestations of salting).

⋆ Jan De Beer is no longer at K.U.Leuven.

Methodology. We intercept requests for drawing text primitives, and build an internal representation of the characters that appear on the screen. This representation is a list of attributed *glyphs* (positioned shapes of individual characters). Then, we test for *glyph visibility* (are glyphs seen by the user?) according to these conditions: (1) *clipping:* glyph drawn within the bounds of the drawing clip, which is a type of 'spatial mask'; (2) *concealment:* glyph not concealed by other shapes; (3) *font colour:* glyph's colour contrasts with the background colour; (4) *glyph size:* large enough glyph size and shape. We compute a visibility score for each feature and consolidate their product into a single confidence score, parameterised by an empirically-tuned penalty factor. The lower the final glyph visibility score, the stronger the indication of hidden text salting.

Evaluation. We use the above salting tricks as features for classifying emails as ham or phishing in a real-life corpus that contains 16,364 ham and 3,636 phishing emails from 04/2007-11/2007. The corpus is protected by non-disclosure privacy-preserving terms. We use a standard Support Vector Machine (SVM) classifier with 10-fold cross validation. We obtain 96.46% precision, 86.26% recall, and 91.07% F-measure. The best classification feature, found in 86% of all phishing emails, is font colour. State-of-the-art phishing classification reaches F-measures of 97.6% (with random forests [3]) up to 99.4% (with SVMs [1]) when using known discriminative features, such as url length & longevity, HTML & Javascript information on the 2002-2003 spamassassin corpus & a public phishing corpus (these corpora are described in [3]). We use **only salting features**. If we also combine these known discriminative features, performance may improve.

3 Conclusions

We detect hidden text salting in emails as hidden visual anomalies in text, unlike existing methods which target spam in general. We show that hidden text salting is used in phishing emails, and that phishing emails can be identified based on hidden text salting features alone. Our method can be used as improved content representation in filtering, retrieval or mining.

Acknowledgments. Partly funded by project EU-FP6-IST-027600-ANTI-PHISH.

References

1. Bergholz, A., Paass, G., Reichartz, F., Strobel, S., Chang, J.-H.: Improved phishing detection using model-based features. In: Conf. on Email and Anti-Spam (CEAS) (2008)
2. Bratko, A., Cormack, G., Filipic, B., Lynam, T., Zupan, B.: Spam filtering using statistical data compression models. J. of Mach. Learn. Res. 7, 2673–2698 (2006)
3. Fette, I., Sadeh, N., Tomasic, A.: Learning to detect phishing emails. In: International World Wide Web Conference (WWW), pp. 649–656 (2007)
4. Fumera, G., Pillai, I., Roli, F.: Spam filtering based on the analysis of text information embedded into images. J. of Mach. Learn. Res. 7, 2699–2720 (2006)
5. Kirda, E., Kruegel, C.: Protecting users against phishing attacks. The Computer Journal 49(5), 554–561 (2006)

Improving Anomaly Detection Error Rate by Collective Trust Modeling
(Extended Abstract)

Martin Rehák[1], Michal Pěchouček[1], Karel Bartoš[2,1], Martin Grill[2,1], Pavel Čeleda[3], and Vojtěch Krmíček[3]

[1] Department of Cybernetics, Czech Technical University in Prague
mrehak@labe.felk.cvut.cz, pechouc@labe.felk.cvut.cz
[2] CESNET, z. s. p. o.
bartosk@labe.felk.cvut.cz, grillm@labe.felk.cvut.cz
[3] Institute of Computer Science, Masaryk University
celeda@ics.muni.cz, vojtec@ics.muni.cz

Abstract. Current Network Behavior Analysis (NBA) techniques are based on anomaly detection principles and therefore subject to high error rates. We propose a mechanism that deploys trust modeling, a technique for cooperator modeling from the multi-agent research, to improve the quality of NBA results. Our system is designed as a set of agents, each of them based on an existing anomaly detection algorithm coupled with a trust model based on the same traffic representation. These agents minimize the error rate by unsupervised, multi-layer integration of traffic classification. The system has been evaluated on real traffic in Czech academic networks.[1]

Network Behavior Analysis attempts to detect the attacks against computer systems by analyzing the network traffic (flow/session) statistics. We present a mechanism that efficiently combines several anomaly detection algorithms in order to significantly reduce their error rate, especially in terms of false positives. The mechanism is based on **extended trust modeling**, a method from the multi-agent field [1], which generalizes traditional trust modeling by introduction of generalized identities and situation representation. The traditional trust models are principally used to identify dishonest partners engaged in repetitive interactions, such as supply chain management.

Traditionally, the alerts from multiple sources are grouped to improve the quality of classification and reduce the number of events presented to the user [2]. Other approaches concentrate on the improvement of several distinct intrusion detection methods, differentiated by the set of traffic features these methods work on [3]. In our work [4], we extend the latter by introducing the collective extended trust modeling as a supplementary layer which further improves the quality of classification.

The system (Fig. 1) receives the flow data in batches, typically covering between 2-5 minutes of network traffic. The data is processed by several anomaly

[1] This material is based upon work supported by the International Technology Center - Atlantic of the US Army under Contract No. W911NF-08-1-0250. Also supported by Czech Min. of Education grants 6840770038 (CTU) and 6383917201 (CESNET).

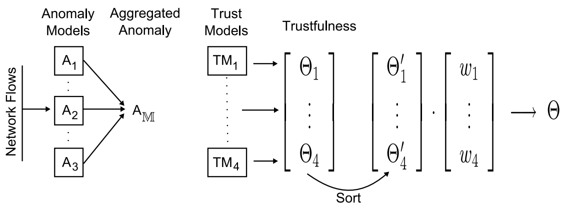


Fig. 1. Detection process overview

detection algorithms, and each algorithm determines an **anomaly value** for each of the flows. The anomaly value is a real number in the $[0, 1]$ interval, with the values close to 0 corresponding to normal flows and the values around 1 being reserved for anomalous flows.

The anomalies provided by individual anomaly detectors are averaged to obtain a single **joint anomaly** for each flows. At this stage, our algorithm differentiates from the existing approaches by introducing another processing layer, based on extended trust modeling. Flow description and joint anomaly value is processed by several trust models. Each of these models represents the flows in a distinct feature space, aggregates them into clusters, and assigns **trustfulness** to these clusters. The trustfulness of the cluster (again in the $[0, 1]$ interval) is aggregated from the joint anomaly of the flows (from all past data sets) that were previously assigned to the cluster.

The system than uses the trustfulness provided for each flow (aggregated from the trustfulness of the close clusters) as its output. Use of trustfulness in lieu of single-file dependent anomaly, together with the order-weighted combination of the results between the models, filters most of the false positives, and significantly reduces the error rate of the mechanism.

We validate [4] our technique on the NetFlow data from the university network (with botnet/P2P traffic, buffer overflow attempts) and empirically show that its use successfully reduces the rate of false positives, while not impacting the false negatives ratio. The false positives are reduced by the factor of 10-20 when compared to the individual anomaly detection methods, and by the factor of 2-4 when compared to joint anomalies.

References

1. Rehak, M., Pechoucek, M.: Trust modeling with context representation and generalized identities. In: Klusch, M., Hindriks, K.V., Papazoglou, M.P., Sterling, L. (eds.) CIA 2007. LNCS (LNAI), vol. 4676, pp. 298–312. Springer, Heidelberg (2007)
2. Valeur, F., Vigna, G., Kruegel, C., Kemmerer, R.A.: A comprehensive approach to intrusion detection alert correlation. IEEE Transactions on Dependable and Secure Computing 01, 146–169 (2004)
3. Giacinto, G., Perdisci, R., Rio, M.D., Roli, F.: Intrusion detection in computer networks by a modular ensemble of one-class classifiers. Information Fusion 9, 69–82 (2008)
4. Rehak, M., Pechoucek, M., Grill, M., Bartos, K.: Trust-based classifier combination for network anomaly detection. In: Cooperative Information Agents XII. LNCS(LNAI), Springer, Heidelberg (to appear, 2008)

Database Intrusion Detection and Response
(Extended Abstract)*

Ashish Kamra and Elisa Bertino

Purdue University
akamra@ecn.purdue.edu, bertino@cs.purdue.edu

Why is it important to have an intrusion detection (ID) mechanism tailored for a database management system (DBMS)? There are three main reasons for this. First, actions deemed malicious for a DBMS are not necessarily malicious for the underlying operating system or the network; thus ID systems designed for the latter may not be effective against database attacks. Second, organizations have stepped up data vigilance driven by various government regulations concerning data management such as SOX, GLBA, HIPAA and so forth. Third, and this is probably the most relevant reason, the problem of *insider threats* is being recognized as a major security threat; its solution requires among other techniques the adoption of mechanisms able to detect access anomalies by users internal to the organization owning the data.

Our approach to an ID mechanism tailored for a DBMS consists of two main elements: an anomaly detection (AD) system and an anomaly response system. We have developed algorithms for detecting anomalous user/role accesses to a DBMS [2]. Our approach considers *two* different scenarios. In the first scenario, it is assumed that the DBMS has a Role Based Access Control (RBAC) model in place. Our AD system is able to determine role intruders, that is, individuals that while holding a specific role, behave differently than expected. The problem in this case is treated as a supervised learning problem. The roles are used as classes for the classification purpose. For every user request under observation, its role is predicted by a classifier. If the predicted role is different from the role associated with the query, an anomaly is detected. In the second case, the same problem is addressed in the context of a DBMS without any role definitions. In such setting, every request is associated with the user that issued it. We build user-group profiles based on the SQL commands users submit to the database. The specific methodology used for anomaly detection is as follows. The training data is partitioned into clusters using standard clustering techniques. A mapping is maintained for every user to its representative cluster (RC). For a new query under observation, its RC is determined by examining the user-cluster mapping. For the detection phase, two different approaches are followed. In the first approach, the classifier is applied in a manner similar to the supervised case with the RCs as classes. In the second approach, a statistical test is used to identify if the query is an outlier in its RC. If the result of the statistical test is positive, the query is marked as an anomaly.

* The work reported here has been partially supported by the NSF grant 0712846 "IPS: Security Services for Healthcare Applications".

In order to build profiles, the log-file entries need to be pre-processed and converted into a format that can be analyzed by the detection algorithms. Therefore, each entry in the log file is represented by a basic data unit that contains five fields, and thus it is called a *quiplet*. The abstract form of a quiplet consists of five fields (*SQL Command, Projection Relation Information, Projection Attribute Information, Selection Relation Information and Selection Attribute Information*). Depending on the level of details required, the quiplets are captured from the log file entries using *three* different representation levels. Each level is characterized by a different amount of recorded information. For details, we refer the reader to [2]. Our approach towards a DBMS specific AD mechanism has several advantages. By modeling the access patterns of users based on the SQL command syntax, the insider threat scenario is directly addressed. Our approach is able to capture users/roles that access relations not conforming to their normal access pattern. Second, the three different granularity levels of representation proposed in the scheme offer alternatives for space/time/accuracy overhead. Third, the profiles themselves can be used by the security administrators to refine existing access control policies of the DBMS or define new ones.

The second element of our approach addresses a common shortcoming of all other DBMS-specific ID mechanisms, that is, a limited number of possible anomaly response options. In all such mechanisms, the response is either *aggressive*, thus dropping the malicious request, or *conservative*, thus simply raising an alarm while letting the malicious request go through. So what more can a DBMS do to respond to a database access anomaly signaling a possible intrusion? Consider an AD system in place for a DBMS. AD systems are useful for detecting novel zero-day attacks, but they are also notorious for generating a large number of false alarms. Taking an aggressive action on every alarm can result in potential denial of service to legitimate requests, while only logging the alarms will nullify the advantages of the AD mechanism. We address these problems using a two-pronged approach [1]. First we propose the notion of *database response policies* that specify appropriate response actions depending on the details of the anomalous request. Second we propose more fine-grain response actions by introducing the concept of *privilege states* in the access control system. For example, as we discuss in [1], the privilege corresponding to an anomalous action may be moved into a *suspended* state until a remedial action, such as a 2^{nd}-factor authentication, is executed by the user. We have implemented a policy language and extended the PostgreSQL DBMS with an engine supporting the enforcement of the response policies. We have also extended the access control mechanism of PostgreSQL to support privilege states. Initial performance evaluation shows that our approach is very efficient.

References

1. Kamra, A., Bertino, E., Nehme, R.: Responding to anomalous database requests. In: Proceedings of Secure Data Management (SDM) (to appear, 2008)
2. Kamra, A., Bertino, E., Terzi, E.: Detecting anomalous access patterns in relational databases. VLDB Journal (2008)

An Empirical Approach to Identify Information Misuse by Insiders
(Extended Abstract)

Deanna D. Caputo, Greg Stephens, Brad Stephenson, Megan Cormier, and Minna Kim

The MITRE Corporation[*]
{dcaputo,gstephens,stephenson}@mitre.org

Abstract. Rogue employees with access to sensitive information can easily abuse their access to engage in information theft. To help differentiate malicious from benign behavior, this study measures how participants, given a common search topic, seek information. This study uses double-blind procedures, a stratified sample, and carefully designed control and experimental conditions. We seek to validate previously identified network indicators (ELICIT), find new host-based behaviors, and consider other human attributes that affect the information-use of malicious insiders by comparing their behavior to equivalent non-malicious users.

Keywords: insider threat, detection, malicious users, misuse.

1 Introduction

Malicious insiders who abuse their privileges to steal valuable information remain largely invisible to current detection methods that rely on rule-breaking behavior. To effectively detect this misuse, one must observe how trusted insiders interact with information and differentiate innocuous from malicious patterns of information-use.

In prior work[1], we developed ELICIT, a network-based system designed to help analysts detect insiders who operate outside the scope of their duties but within their privileges. The current research uses the same approach, observing information-use and applying user and information context. This study will evaluate ELICIT's detectors across a different participant pool while adding host-based monitoring, and

[*] This material is based upon work supported by the U.S. Department of Homeland Security under Grant Award Number 2006-CS-001-000001, under the auspices of the Institute for Information Infrastructure Protection (I3P) research program. The I3P is managed by Dartmouth College. The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the U.S. Department of Homeland Security, the I3P, or Dartmouth College.

[1] Maloof, M.A., and Stephens, G.D. "ELICIT: A system for detecting insiders who violate need-to-know." Recent Advances in Intrusion: 146-166.

considering baseline human behavior as well as individual differences in a controlled environment.

Envision a labyrinth where people enter the maze (an information landscape) at different locations (intentions) yet seek the same prize (information). Can one's path to the prize tell us where they started from? We hypothesize that the information gathering patterns of maliciously motivated users will differ in predictable ways from those of benignly motivated users. For example, malicious insiders may attempt to hide their bad behavior by interleaving it with separate innocuous information gathering sessions whereas benign users may focus on a single information gathering session.

2 Methods

There will be a minimum of 50 participants in this study. They will all be MITRE employees and the sample will be stratified by seniority in the company.

The experimental procedure is double-blind to guard against bias and placebo effects. Participants are randomly assigned to one of two conditions: Benign User (control) or Malicious User. Participants are recruited under the cover story that we are monitoring computer use while testing the latest anti-keylogging software. Deception is necessary so they are all unaware that we are studying insider threat behaviors. Participants complete a pre-questionnaire asking for biographical data and other behavioral questions of interest.

Each participant receives a study laptop running software that monitors their information-use behavior. They are randomly assigned one of two scenarios, based on their condition, explaining a role and task. Both conditions are tasked to search the MITRE intranet and deliver the most valuable information found, on an identical topic, onto a CD and are informed that it will be evaluated by subject matter experts (creating a performance demand). Each participant is given up to 10 hours to play the role and complete the task over a 7 day period. Participants complete a post-questionnaire about their experience, the role, the task, and other behaviors of interest.

The two scenarios were designed to be completely balanced, except for the experimental variable—*user intent*. Both roles describe a person who has fallen on hard financial times and must complete the task in order to improve their financial situation. In the benign condition, the person joins a high profile team and good performance on that team will lead to a promotion and pay increase. In the malicious condition, the person accepts a new, higher paying job. The offer is conditional on bringing inside information from his old employer that would provide the new employer a competitive advantage.

Monitoring is done using the network-based ELICIT sensors and the host-based product Verdasys' Digital Guardian. Together, the sensors monitor information-use events including file/directory reads, writes, moves, and deletes. They also monitor search engine queries, cut-and-pastes, application launches, and URLs visited. Events will be analyzed to measure statistical differences in information usage for each condition. We will apply previously determined ELICIT indicators where appropriate, and look for new behavior patterns in network and host activity. Participant responses to pre-/post-experiment questionnaires will be analyzed across conditions.

Page-Based Anomaly Detection in Large Scale Web Clusters Using Adaptive MapReduce (Extended Abstract)

Junsup Lee[1] and Sungdeok Cha[2]

[1] The Attached Institute of ETRI, Daejeon, Republic of Korea
jslee@dependable.kaist.ac.kr
[2] Department of CSE, Korea University, Seoul, 136-701, Republic of Korea
scha@korea.ac.kr

Abstract. While anomaly detection systems typically work on single server, most commercial web sites operate cluster environments, and user queries trigger transactions scattered through multiple servers. For this reason, anomaly detectors in a same server farm should communicate with each other to integrate their partial profile. In this paper, we describe a real-time distributed anomaly detection system that can deal with over one billion transactions per day. In our system, base on Google MapReduce algorithm, an anomaly detector in each node shares profiles of user behaviors and propagates intruder information to reduce false alarms. We evaluated our system using web log data from www.microsoft.com. The web log data, about 250GB in size, contains over one billion transactions recorded in a day.

Anomaly detection systems are often considered impractical solutions because of two major limitations. One is a difficulty of collaboration among anomaly detectors from web servers. The other limitation is real-time consideration. Existing systems fail to deliver real-time performance and often require expensive computational cost during training and evaluation. Conventional ADSs implicitly assume that all activities related to an event have been completed before the event may be inspected. To satisfy such assumption, the systems usually have timing windows to ensure that an event is not inspected until complete information is available. Consequently, the systems fail to satisfy real-time constraints.

To overcome these issues, we developed a page-based anomaly detection system (PADS). The PADS keeps track of access patterns on each service object such as web page and generate models per pages. In this page-profile based ADS, compare with other user-based ADS, timing windows are unnecessary during operation. An anomaly detector in each node, base on Google MapReduce algorithm [1], shares self-learned profiles and propagates intruder information to reduce false alarms.

The PADS architecture employs a combination of self-learning and profile-based anomaly detection techniques. Self-learning methodology enables the PADS to study the usage and traffic patterns of web service objects over time. In

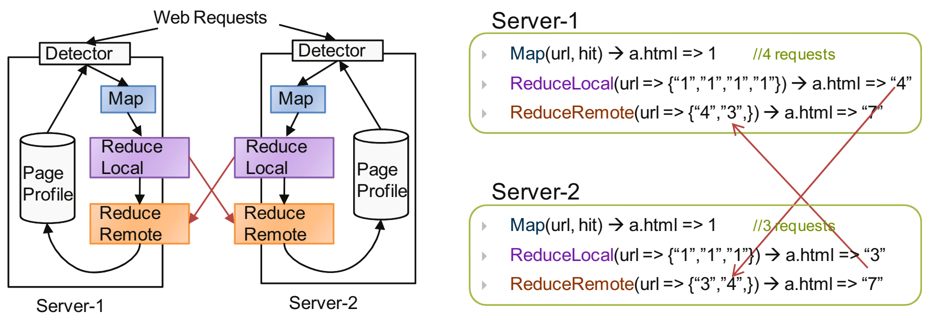

Web Requests
Detector
Map
Reduce Local
Reduce Remote
Page Profile
Server-1
Server-2
Server-1
Map(url, hit) → a.html => 1 //4 requests
ReduceLocal(url => {"1","1","1","1"}) → a.html => "4"
ReduceRemote(url => {"4","3",}) → a.html => "7"
Server-2
Map(url, hit) → a.html => 1 //3 requests
ReduceLocal(url => {"1","1","1"}) → a.html => "3"
ReduceRemote(url => {"3","4",}) → a.html => "7"

Fig. 1. Overview of PADS, Example of MapReduce in PADS (Page hit)

our model, profiles that summarize statistical features (such as exchanged bytes, query related behavior and HTTP traffic features using Chebyshev inequality, relative frequency and clustering method) per web service objects (e.q., html, asp, aspx, mspx and so on). Because popular web sites do experience legitimate and sometimes unexpected surge on particular web pages, the PADS automatically specifies access thresholds per web pages to understand unusual patterns that may occur during legitimate web query operations.

To keep pace with the latest usage and traffic pattern per web pages, each PADS node in a server farm propagates the profiles to other nodes using adaptive MapReduce technique[2]. We redesigned the technique which aware memory and network traffic. Using MapReduce algorithm, user requests are summarized into page profiles and share with other server simultaneously (Fig 1). According to our experiment on Microsoft web log data, the PADS only generate 4.44% of network traffic compare with sharing all requests among servers. In a half hour, a server receives 0.42 million requests on average. In the meantime, only 7.2 thousand page profiles are produced by LocalReduce. While total size of these profiles is 3.6MB, all requests are 81MB relatively.

In our system, each web page profile is generated dynamically by the system initially and subsequently updated and shared with other servers in real time by Google MapReduce algorithm. Currently, we are evaluating PADS using a web log data from 'www.microsoft.com'. While every web servers maintain same latest web page profiles which is about million, communication overhead between nodes is dramatically low.

References

1. Dean, J., Ghemawat, S.: Mapreduce: Simplified data processing on large clusters. Operating Systems Design and Implementation, 137–149 (2004)
2. Ranger, C., Raghuraman, R., Penmetsa, A., Bradski, G., Kozyrakis, C.: Evaluating MapReduce for Multi-core and Multiprocessor Systems. In: Proceedings of the 13th Intl. Symposium on HPCA, Phoenix, AZ (February 2007)

Automating the Analysis of Honeypot Data (Extended Abstract)

Olivier Thonnard[1], Jouni Viinikka[2], Corrado Leita[1], and Marc Dacier[3]

[1] Institut Eurecom
olivier.thonnard@rma.ac.be, Corrado.Leita@eurecom.fr
[2] France Telecom R&D
jouni.viinikka@orange-ftgroup.com
[3] Symantec Research Labs, France
dacier@eurecom.fr

Abstract. We describe the on-going work towards further automating the analysis of data generated by a large honeynet architecture called Leurre.com and SGNET. The underlying motivation is helping us to integrate the use of honeypot data into daily network security monitoring. We propose a system based on two automated steps: *i)* the detection of relevant attack events within a large honeynet traffic data set, and *ii)* the extraction of highly similar events based on temporal correlation.

Keywords: Honeypots, Internet threats analysis, malicious behavior characterization.

1 Introduction

We look to identify and characterize certain large-scale phenomena that are active on the Internet by detecting similarities across network attack traces in an automated manner. The analyzed data is extracted from datasets collected through Leurre.com [4] and SGNET honeypot deployments [5]. By automating our analysis, it should help us to integrate the use of honeypot data into daily network security monitoring. To achieve this we need to identify relevant periods of activity on the sensors shortly after they occurred. These periods of activity are analyzed to detect temporal and spatial similarities within the observed attack processes.

In [2] the authors highlighted the usefulness of analyzing temporal correlations between different attacks collected through a honeynet, e.g. to highlight synchronized attack patterns which are part of a very same phenomenon, or to discover stealthier attack phenomena related to botnet propagation schemes or "multi-headed" attack tools [3].

Once groups of similar attack events are revealed, we can perform a more in-depth analysis of those specific groups so as to characterize them with respect to other relevant attack features, e.g. by analyzing the spatial and temporal characteristics of the attackers, or by looking at other meta-information obtained from the SGNET sensors, such as shellcode or malware characteristics when the attacks have led to a successful upload of shellcode commands and malicious binaries.

Thanks to the extensive characterization of those similar attack events, we seek to discover other types of similarities across attack events, even when they occurred at different periods of time. As a result, this process should facilitate the identification of possible root causes for new attack phenomena. The characterization and correlation steps may currently require some manual or semi-automated work.

2 Analysis Process

The main idea of our ongoing effort consists to automatically i) detect relevant attack events within a large honeynet traffic data set shortly after it has been collected, and ii) group highly similar temporal events by relying on *clique* algorithms and appropriate similarity metrics.

We propose to use in step i) an approach based on non-stationary autoregressive (NAR) modeling using Kalman fixed-lag smoothers [1] and in step ii) we use clique algorithms described in [2,6].

The strengths of the detection algorithm are its capability to flag the beginnings of activity periods and isolated activity peaks. Our initial results show that the algorithm is effective when applied to the three different types of time series identified in [2], i.e. ephemeral spikes, sustained bursts and continous patterns. The shortcomings are related to detection of the end of an activity period, the association of the end to the beginning, and the risk of an activity peak begin masked by closely preceding peak. We look to improve these aspects of the detection algorithm.

Then, to correlate the identified attack events, we use an approach based on *maximal cliques* [6], which are able to group all events having important similarities in an unsupervised manner. The main advantage of this approach is that the number of groups (or cliques) does not need to be specified before executing the clustering, and many different feature vectors and similarity distances can be used transparently. We currently use two different techniques: *i)* the dominant sets approach developed by Pavan and Pelillo[7], and *ii)* the quality-based clustering developed in [2]. While the first approach provides a real approximation of the maximum clique problem (known to be NP-hard), the second approach is more pragmatic and is mainly focused on finding cliques having a high quality garantee with a low computational overhead. The choice of one or another clique algorithm depends on the intrinsic characteristics of the data set, as well as the *feature vectors* used in the data mining process.

References

1. Viinikka, J., Debar, H., Mé, L., Lehikoinen, A., Tarvainen, M.: Processing intrusion detection alert aggregates with time series modeling. Information Fusion Journal (2008); Special Issue on Computer Security (to appear)
2. Thonnard, O., Dacier, M.: A Framework for Attack Patterns Discovery in Honeynet Data. In: Digital Forensic Research Workshop (DFRWS) (2008)
3. Pouget, F., Urvoy-Keller, G., Dacier, M.: Time signatures to detect multi-headed stealthy attack tools. In: 18th Annual FIRST Conference, Baltimore, USA (2006)
4. The Leurre.com Project, http://www.leurrecom.org
5. Leita, C., Dacier, M.: SGNET: a worldwide deployable framework to support the analysis of malware threat models. In: Proceedings of EDCC 2008, 7th European Dependable Computing Conference, Kaunas, Lithuania, May 7-9 (2008)
6. Pouget, F., Dacier, M., Zimmerman, J., Clark, A., Mohay, G.: Internet attack knowledge discovery via clusters and cliques of attack traces. Journal of Information Assurance and Security 1(1) (March 2006)
7. Pavan, M., Pelillo, M.: A new graph-theoretic approach to clustering and segmentation. In: Proceedings of IEEE Conference on Computer Vision and Pattern Recognition (2003)

Anomaly and Specification Based Cognitive Approach for Mission-Level Detection and Response[*]

(Extended Abstract)

Paul Rubel[1], Partha Pal[1], Michael Atighetchi[1], D. Paul Benjamin[2], and Franklin Webber[1]

[1] BBN Technologies, Cambridge MA 21038, USA
prubel@bbn.com, ppal@bbn.com, matighet@bbn.com, franklin@eutaxy.net
[2] Pace University, 1 Pace Plaza, New York NY 10038, USA
benjamin@pace.edu

Abstract. In 2005 a survivable system we built was subjected to red-team evaluation. Analyzing, interpreting, and responding to the defense mechanism reports took a room of developers. In May 2008 we took part in another red-team exercise. During this exercise an autonomous reasoning engine took the place of the room of developers. Our reasoning engine uses anomaly and specification-based approaches to autonomously decide if system and mission availability is in jeopardy, and take necessary corrective actions. This extended abstract presents a brief summary of the reasoning capability we developed: how it categorizes the data into an internal representation and how it uses deductive and coherence based reasoning to decide whether a response is warranted.

1 The Basic Idea

Requiring experts to manage a system's defenses is an expensive undertaking, even assuming that such operators can be found. With faster CPUs, more RAM, faster and higher capacity networks we can transfer this tedious work to a reasoning engine. Our reasoning engine uses the mission concept (a model of how the system functions in a particular context) and sensor inputs, generated while the mission runs, to autonomously defend the system.

1.1 Challenges and Solution Approach

The main challenge is making sense of the low level observables reported by the survivability architecture in the context of the current system and mission, and then deciding what and when remedial actions should be taken. Additionally, we want the reasoning to accommodate new systems and missions.

At the center of our reasoning engine is a general reasoner, bracketed by system specific adapter logic. The input adapter takes alerts and turns them into **accusations**. The general reasoner then uses these accusations to make

[*] This research was funded by DARPA under Navy Contract No. N00178-07-C-2003.

hypotheses and passes **claims** and hypotheses to the output adapter where they are evaluated in the mission context and may be acted upon.

Accusations are abstract alerts, expressive enough to enable a wide range of responses while not overwhelming the reasoner with unnecessary distinctions. Accusations come in five types: value (wrong data), omission, flood, timing (right message at the wrong time), and policy (not following a specification). From these accusations the reasoning engine generates four types of hypotheses, which are potential explanations of the accusation: dead host, corrupt host, flooded host, or communication is broken between hosts. A single accusation may create multiple hypotheses. For example, we assume that the sender may be corrupt so accusing a host of not sending a reply creates a dead hypotheses about the accused as well as a corrupt hypothesis about the accuser.

In order for a hypothesis to be acted upon, there needs to be sufficient support to turn that hypothesis into a claim. Claim selection relies upon four main techniques: deductive reasoning, coherence search[1], mission knowledge, and heuristic techniques. Deductive reasoning takes the current hypotheses and system knowledge and attempts to logically prove hypotheses. Coherence search takes multiple accusations, each supporting hypotheses, and aggregates the support. In this way a single source will likely not turn a hypothesis into a claim but a collection of accusations may. Mission knowledge is used to include or exclude some options. For example, if a host is corrupt but is critical to the mission a reboot may initially be preferred to permanently blocking its network traffic. Finally, we use heuristics to choose claims when the other techniques have failed to come up with any workable claims but yet actions still need to be taken.

2 Evaluation

In May of 2008 our system was subjected to an external red-team evaluation. One goal was to effectively respond to 50% of attacks. Preliminary results delivered immediately after the exercise showed 89% of the attacks were detected, and of those detected, 69% were responded to effectively. Additional analysis is ongoing.

3 Conclusion

Application of cognitive/knowledge based tools, especially in the area of specification and anomaly-based detection and response, at the mission level, is a promising way to extend the reach of current intrusion detection technology and enhance the overall accuracy of true detection. One issue, still left unresolved, is the needed speed of cognitive processing component. Our goal was to respond in 250ms. In some cases we achieved that target during evaluation, but in others our reasoning took multiple seconds, a problem which needs further refinement.

References

1. Freuder, E., Wallace, R.: Partial constraint satisfaction. Artificial Intelligence, special issue on constraint-based reasoning 58(1-3), 21–70 (1992)

Monitoring the Execution of Third-Party Software on Mobile Devices (Extended Abstract)

Andrew Brown and Mark Ryan

School of Computer Science, University of Birmingham, UK. B15 2TT
`{A.J.Brown, M.D.Ryan}@cs.bham.ac.uk`

Abstract. The current security model for a third-party application running on a mobile device requires its user to trust that application's vendor and whilst mechanisms exist to mediate this relationship, they cannot guarantee complete protection against the threats posed. This work introduces a security architecture that prevents a third-party application deviating from its intended behaviour, defending devices against previously unseen malware more effectively than existing security measures.

In 2002, mobile device capabilities were expanded to permit users to install applications from sources other than the cellular network operator and they now mirror those of more traditional hosts. 2004 saw the first malware aimed at mobile devices hit and today over four hundred known entities exist. By 2009, it is estimated that 200 million "smart" mobile devices will be in operation, setting the scene for widespread malware infection.

Mobile device architectures commonly utilise code signing, discretionary access controls and signature-based anti-virus software to secure third-party software installations. Digitally signing code can confirm its author and guarantee that it has not been altered since it was signed, but does not guarantee the quality or security of code that the application will execute: determined attackers will go to many lengths to obtain a signature for their code. Access controls contribute to a systematic security framework, but are inflexible: default settings tend to leave the device vulnerable to numerous attacks and applying stricter controls impedes program functionality. Mobile anti-virus software can only detect *known* malware entities whose signatures exist in a virus dictionary and attack recovery simply deletes an application's executable files.

We propose an architecture for mediating third-party software that uses *execution monitors*, which operate in parallel (as a separate thread) with the target application in order to analyse and mitigate the events it invokes. This enables full regulation of the target's interaction with its host's resources, preventing and recovering from harmful behaviour in real-time. As most end-users do not have the technical capability to specify or deploy such monitors, we have developed ABML – a high-level policy language in which they can express *a priori* judgements about the type of application downloaded, which are translated by our compiler into a monitor specification. An ABML policy contains a set of rules which reason about temporally-ordered application events, sets of local and global variables, and can be categorised by the class of application it is applied to (e.g., an *editor*, a *browser*, a *game*, a *messenger*).

A policy is compiled into Java source code and then enforced on application bytecode by the Polymer [1] engine. This executes on the JVM and monitors calls the application makes to the Java ME and native device libraries (Fig. 1). Policy violations are recovered from by weaving a set of recovery events into application bytecode, which are derived from a rule at compile-time. Where a policy denies its triggering event, that event can be removed from the target's instruction stream and execution can continue. Our language is equipped with constructs that more precisely identify the context of an event, leading to more fine-grained application control. It can therefore mitigate some forms of information-flow and ensure that only data which is not deemed sensitive is transmitted by the application to the device's carrier network. In addition, our architecture ensures that an attacker cannot write application code to bypass our security measures and control the operating system directly. Third-party applications can only gain access to native device functions whilst an ABML policy is being enforced on them.

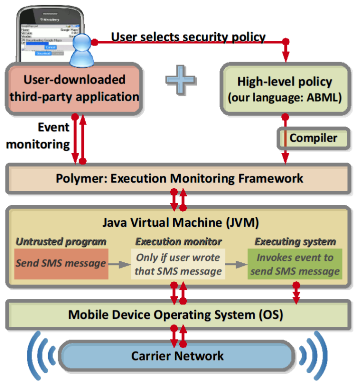

User selects security policy
User-downloaded third-party application
High-level policy (our language: ABML)
Event monitoring
Compiler
Polymer: Execution Monitoring Framework
Java Virtual Machine (JVM)
Untrusted program
Execution monitor
Executing system
Send SMS message
Only if user wrote that SMS message
Invokes event to send SMS message
Mobile Device Operating System (OS)
Carrier Network

Fig. 1. System architecture

We have proven this concept using the BlackBerry 8800-series mobile device, although our work is cross-platform (for it to work on other types of device, ABML's libraries are re-mapped to the APIs of the target platform). An example attack we recently studied allowed an application to intercept and forward SMS messages to an attacker and could occur despite that application being signed. Device access controls queried the user on the target's first attempt to send an SMS, but where the user agreed to this prompt, an SMS 'channel' to the attacker was created. Our countermeasure to this stated: *"the target may send an SMS message only if the data that message contains was entered manually by the user, and the recipient of that message exists as a contact in the user's personal information manager (PIM)"*. In order to enforce such a policy, a monitor must precisely identify the context in which the triggering event occurred: *was the data contained in that SMS message typed by the user? At some time after the entry of this data, did the user press "send" in reference to this message?* and *is that message to be sent to recipient in the device's PIM*? Where any of these conditions evaluates to false, the device's operating system never receives the command to send that SMS message and the application continues executing.

Reference

1. Bauer, L., Ligatti, J., Walker, D.: Composing security policies with Polymer. In: PLDI 2005: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation, New York, USA, pp. 305–314 (2005)

Streaming Estimation of Information-Theoretic Metrics for Anomaly Detection[⋆] (Extended Abstract)

Sergey Bratus, Joshua Brody, David Kotz, and Anna Shubina

Institute for Security Technology Studies
Department of Computer Science, Dartmouth College, USA
sergey@cs.dartmouth.edu, jbrody@cs.dartmouth.edu,
dfk@cs.dartmouth.edu, ashubina@cs.dartmouth.edu

Abstract. Information-theoretic metrics hold great promise for modeling traffic and detecting anomalies if only they could be computed in an efficient, scalable way. Recent advances in streaming estimation algorithms give hope that such computations can be made practical. We describe our work in progress that aims to use streaming algorithms on 802.11a/b/g link layer (and above) features and feature pairs to detect anomalies.

Information-theoretic statistics applied to monitoring of network traffic can be useful in detecting changes in its character [7,5,4]. These metrics make few assumptions about what constitutes normal and abnormal traffic (e.g., [3]), and so should do well at adapting to traffic characteristics of specific networks, realizing the "home network advantage" of prior knowledge that defenders have over outside attackers.

However, necessary computations place a heavy load on both the sensor CPU and RAM. Thus, scalability of methods that rely on precise real-time computations of entropy and other related statistics remains a challenge. Luckily, a new class of streaming algorithms produce practically usable estimated results with much smaller requirements to CPU and RAM [6,2]. They have the potential to allow information-theoretic metrics to be scalably used in practice.

Several experimental systems (including Wi-Fi link layer anomaly detectors being developed at Dartmouth) apply entropy of pre-selected packet or session features to produce alerts. Such mechanisms rely on the idea that a change in the character of a feature distribution is suspicious. In our experience, watching a set of features as if they were independent is highly prone to false positives. A change in the entropy of a feature may be due to factors such as normal business day and other workflow cycles. Even the simplest cases of single protocol features require, e.g., some modeling of when a particular protocol is normally expected to be in use.

[⋆] This research program is a part of the Institute for Security Technology Studies, supported by Intel Corporation, NSF grant CCF-0448277, and by Award number NBCH2050002 from the U.S. Department of Homeland Security, Science and Technology Directorate. Points of view in this document are those of the authors and do not necessarily represent the official position of the U.S. Department of Homeland Security, Intel Corporation, or any other sponsor.

Conditional entropy between pairs of features are likely to provide a better metric of normal use, because it relies on tracking the average "predictability" of one feature given the knowledge of another. Such relationships are more likely to persist through diurnal cycles, because they are less related to volumes of traffic.

Unusual use of protocol fields is characteristic of many exploits, but sophisticated attackers take pains to disguise it, as IDSes might be watching for it. It is much harder to disguise unusual payloads in such a way that does not introduce unusual statistical effects in pairs of protocol features. Note that rule-based IDS evasion techniques themselves (e.g., [8]) can produce just such effects.

Streaming estimation algorithms open up the possibility of a scalable sampling-based system that allows tracking of joint distributions, and thus of mutual information-type statistics. Furthermore, the sampling scheme used in the estimation algorithm can be adjusted dynamically depending on how much precision is meaningful and practicable for a particular network.

The 802.11a/b/g link layer is feature-rich and complex. Besides the frame type and subtype fields, the link layer header may contain one to four MAC address fields, eight bit flags, and two 16-bit fields, frame sequence number and duration (the distribution of which has been shown[1] to identify wireless chipset–driver combination as a distinctive fingerprint).

Thus this link layer allows a range of interesting attacks and related statistical distibution anomalies. We distinguish between the four levels of features, based on the sensor RAM and CPU requirements to follow them: (a) PHY layer errors as calculated and reported by the firmware, (b) frequency of basic events, such as observing deauthentication frames, (c) single header field values' frequency distributions, and (d) joint and conditional distributions of pairs of features. Anomalies in (a) may indicate inteference or jamming, and (b) frequency serves as good indicators of various DoS-type flooding and resource consumption attacks, whereas (c) and especially (d) expose other attacks that involve unusual headers and payloads.

References

1. Cache, J.: Fingerprinting 802.11 implementations via statistical analysis of the duration field. Uninformed Journal 5(1) (September 2006)
2. Chakrabarti, A., Cormode, G., McGregor, A.: A near-optimal algorithm for computing the entropy of a stream. In: SODA 2007: Proceedings of the eighteenth annual ACM-SIAM symposium on Discrete algorithms, pp. 328–335 (2007)
3. Gu, G., Fogla, P., Dagon, D., Lee, W., Skoric, B.: Towards an information-theoretic framework for analyzing intrusion detection systems. In: Gollmann, D., Meier, J., Sabelfeld, A. (eds.) ESORICS 2006. LNCS, vol. 4189, pp. 527–546. Springer, Heidelberg (2006)
4. Gu, Y., McCallum, A., Towsley, D.: Detecting anomalies in network traffic using maximum entropy estimation. In: IMC 2005: Proceedings of the 5th ACM SIGCOMM conference on Internet measurement, pp. 1–6 (2005)

5. Lakhina, A., Crovella, M., Diot, C.: Mining anomalies using traffic feature distributions. In: SIGCOMM 2005: Proceedings of the 2005 Conference on Computer Communication, pp. 217–228. ACM, New York (2005)
6. Lall, A., Sekar, V., Ogihara, M., Xu, J., Zhang, H.: Data streaming algorithms for estimating entropy of network traffic. SIGMETRICS Performance Evaluation Review 34(1), 145–156 (2006)
7. Lee, W., Xiang, D.: Information-theoretic measures for anomaly detection. In: Proc. of the 2001 IEEE Symposium on Security and Privacy, pp. 130–143 (2001)
8. Ptacek, T.H., Newsham, T.N.: Insertion, evasion, and denial of service: Eluding network intrusion detection, January 1998. Secure Networks, Inc. (1998)

Bots Behaviors vs. Human Behaviors on Large-Scale Communication Networks (Extended Abstract)

Wei Lu[1,2] and Ali A. Ghorbani[1]

[1] Faculty of Computer Science, University of New Brunswick, Fredericton, NB Canada
[2] Department of Electrical and Computer Engineering, University of Victoria, BC Canada
{wlu,ghorbani}@unb.ca

Abstract. In this paper we propose a hierarchical framework for detecting and characterizing any types of botnets on a large-scale WiFi ISP network. In particular, we first analyze and classify the network traffic into different applications by using payload signatures and the cross-associations for IP addresses and ports. Then based on specific application community (e.g. IRC, HTTP, or Peer-to-Peer), we present a novel temporal-frequent characteristic of flows that leads the differentiation of malicious behaviors created by bots from normal network traffic generated by human beings. We evaluate our approach with over 160 million flows collected over five consecutive days on a large-scale network and preliminary results show the proposed approach successfully detects the IRC botnet flows from over 160 million flows with a high detection rate and an acceptable low false alarm rate.

1 Problem Statement, State of the Art and Contributions

Detecting botnets behaviors on large-scale networks is a very challenging problem. This is because: (1) botnets are often hidden in existing applications, and thus their traffic volume is not that big and is very similar with normal traffic behaviors; (2) identifying network traffic into different applications becomes more challenging and is still an issue yet to be solved due to traffic content encryption and the unreliable destination port labeling method. The observation on a large-scale WiFi ISP network over a half year period showed that even exploring the flow content examination method, there are still about 40% network flows that cannot be classified into specific applications. Investigating such a huge number of unknown traffic is very important since they might stand for the abnormalities in the traffic, malicious behaviors or simply the identification of novel applications.

Current attempts on detecting botnets are mainly based on honeypots, passive anomaly analysis and traffic application classification. The anomaly analysis for detecting botnets on network traffic is usually independent of the traffic content and has the potential to find different types of botnets. However, anomaly

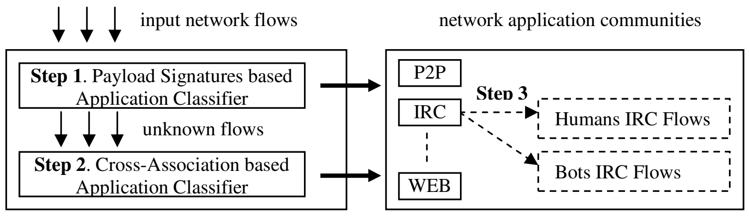

input network flows
Step 1. Payload Signatures based Application Classifier
unknown flows
Step 2. Cross-Association based Application Classifier
network application communities
P2P
IRC
WEB
Step 3
Humans IRC Flows
Bots IRC Flows

Fig. 1. The proposed hierarchical framework for botnets detection

detection tends to generate a large volume of false alarms traditionally when deployed on a large-scale communication network. The traffic application classification based botnets detection focuses on classifying traffic into IRC traffic and non-IRC traffic, offering a potential to reduce number of false alarms, but can detect IRC based botnets only.

In this paper, we focus on traffic classification based botnets detection. Instead of labeling and filtering traffic into non-IRC and IRC, we propose a hierarchical framework illustrated in Fig. 1 for discriminating malicious behaviors generated by any types of bots from normal behaviors generated by human beings. The major contributions of this work include: (1) a novel application discovery approach for classifying traffic into different network application communities (e.g. P2P, Chat, Web, etc.) on a large-scale WiFi ISP network, in which the input flows are first labeled through payload signatures (i.e. Step 1 of Fig.1) and unknown flows are then labeled through the cross-associations of IP addresses and port numbers (i.e. Step 2 of Fig.1); (2) a novel temporal-frequent metric based on N-gram (frequent characteristic) of flow payload over a time period (temporal characteristic) for discriminating bots behaviors from humans behaviors on a large-scale network (i.e. Step 3 of Fig.1).

2 Preliminary Evaluation Results and Conclusions

We implement a prototype system for the proposed hierarchical framework and then evaluate it on a large-scale WiFi ISP network over five consecutive business days. Our traffic classification approach can classify the unknown IRC flows into the IRC application community with a 100% classification rate on the five days evaluation. The detection rate for differentiating bots IRC traffic from normal human beings IRC traffic is 100% on four days testing, while an exception happens on the third day's testing on which our prototype obtained a 77.8% detection rate with a 3.1% false alarm rate. The best evaluation over the five days testing is a 100% detection rate with only 1.6% false alarm rate. Moreover, the preliminary evaluation results show that the average standard deviation of bytes frequency over the 256 ASCIIs on the flow payload is an important metric to indicate normal human IRC traffic and malicious IRC traffic generated by machine bots. In the near future, we will conduct an experimental evaluation with the web based botnets and new appeared P2P botnets.

Anomalous Taint Detection
(Extended Abstract)*

Lorenzo Cavallaro[1] and R. Sekar[2]

[1] Department of Computer Science, University of California at Santa Barbara, USA
`sullivan@cs.ucsb.edu`
[2] Department of Computer Science, Stony Brook University, USA
`sekar@cs.sunysb.edu`

Abstract. We propose *anomalous taint detection*, an approach that combines fine-grained taint tracking with learning-based anomaly detection. Anomaly detection is used to identify behavioral deviations that manifest when vulnerabilities are exercised. Fine-grained taint-tracking is used to target the anomaly detector on those aspects of program behavior that can be controlled by an attacker. Our preliminary results indicate that the combination increases detection accuracy over either technique, and promises to offer better resistance to mimicry attacks.

1 Introduction

A number of approaches have been developed for mitigating software vulnerabilities. Of these, learning-based anomaly detection has been popular among researchers due to its ability to detect novel attacks. Although the basic assumption behind anomaly detection, which states that attacks manifest unusual program behaviors, is true, the converse does not hold: unusual behaviors are not necessarily attacks. As a result, anomaly detection techniques generally suffer from a high rate of false positives, which impact their practical deployment.

Recently, fine-grained taint-tracking has become popular in software vulnerability defense. Its strength lies in its ability to reason about the degree of control exercised by an attacker on data values within the memory space of a vulnerable program. This enables the development of security policies that can, with high confidence, detect dangerous uses of such "tainted" data in security-critical operations. This technique is capable of defeating a wide range of attacks, including code injection, command injection and cross-site scripting[1]. Its main drawback is the requirement for manual policy development, which can be hard for some classes of attacks, e.g., non-control data attacks[2] and directory traversals.

We propose a new taint-based approach in this paper that avoids the need for policies by leveraging an anomaly detector. By targeting the anomaly detector on tainted data

* This research was supported in part by an NSF grant CNS-0627687, and performed while the first author was a PhD student from Università degli Studi di Milano, Italy visiting Stony Brook University.

[1] See, for instance, XU, BHATKAR and SEKAR, *"Taint-enhanced Policy Enforcement: a Practical Approach to Defeat a Wide Range of Attacks,"* USENIX Security Symposium, 2006.

[2] These attacks corrupt security-critical data without subverting control-flow. Chen *et al.* (*"Non-Control-Data Attacks Are Realistic Threats,"* USENIX Security Symposium, 2005) showed that they can achieve the same results as code injection attacks, while evading many code injection defenses.

and/or events, our approach can avoid a large fraction of false positives that occur due to benign anomalies, i.e., behavioral deviations that are not under the attacker's control.

2 Anomalous Taint Detection

Our starting point is a system-call based program behavior model, e.g., the one used by Forrest *et al.* ("A Sense of Self for Unix Processes," IEEE Security and Privacy '96). We enhance this model with information about system call arguments and taint. As in Bhatkar *et al.* ("Dataflow Anomaly Detection," IEEE Security and Privacy '06), this learning technique leverages the control-flow context provided by system-call models.

Our technique learns information about system calls (or other interesting functions) and their arguments at multiple granularity. At a coarse granularity, it learns whether an event's argument is tainted. At a finer granularity, it learns whether structure fields (or array elements) are tainted. Furthermore, we also generate application-specific taint-enhanced profiles, such as expected maximum and minimum argument lengths, structural inference with character class mapping, and longest common prefix models.

We briefly illustrate our technique[3] using a format-string vulnerability existing in the WU-FTPD program. This program elevates its privileges temporarily, and then uses the following code snippet to revert its privilege to that of a normal user. Chen *et al.* demonstrated a non-control data attack that overwrites pw->pw_uid field with zero. As a result, the server does not revert to user privilege.

```
FILE *getdatasock(...) {
    ...
    seteuid(0);
    setsockopt(...);
    ...
    seteuid(pw->pw_uid);
    ...
}
```

Our approach can detect this attacks in two ways. First, the attack causes this seteuid's argument to be tainted, whereas the argument is untainted under normal operation. Second, the attack causes deviations in the structure of a (tainted) argument to a printf-like function. While the latter method is tied to the specifics of the underlying vulnerability, the former technique is able to detect the effect of corruptions that may be caused by other vulnerabilities as well.

By leveraging on taint information, our approach is less vulnerable to mimicry-like attacks than, for instance, a learning-based anomaly detection approach which relies *only* on statistical properties of the observed data, e.g., Mutz *et al.* ("Anomalous System Call Detection," ACM TISSEC 2006). With a purely learning-based approach, if a limited number of authenticated users were observed during training, then a mimicry attack would be possible that may allow an attacker to impersonate any one of these users.

We have been able to detect other non-control data attacks described by Chen *et al.* using models that reason about the structure and/or lengths of tainted arguments. Our future work is aimed at (a) extending the technique to work on other attack types that require application-specific taint policies (e.g., directory traversals), and (b) deriving taint policies from the taint-enhanced behavioral models that can provide the basis for preventing (rather than just detecting) exploits.

[3] Additional details can be found in CAVALLARO AND SEKAR, *"Anomalous Taint Detection"*, Tech Report SECLAB08-06 at http://seclab.cs.sunysb.edu/pubs.html.

Deep Packet Inspection Using Message Passing Networks
(Extended Abstract)

Divya Jain, K Vasanta Lakshmi, and Priti Shankar

Indian Institute of Science
divya@csa.iisc.ernet.in, kvasanta@csa.iisc.ernet.in,
priti@csa.iisc.ernet.in

Abstract. We propose a solution based on message passing bipartite networks, for deep packet inspection, which addresses both speed and memory issues, which are limiting factors in current solutions. We report on a preliminary implementation and propose a parallel architecture.

1 The Problem, Our Solution and Results

Packet content scanning at high speed is crucial to network security and network monitoring applications. In these applications, the packet payload is matched against a given set of patterns specified as regular expressions to identify specific classes of applications, viruses, protocol definitions, etc. Unfortunately, the speed requirement cannot be met in many existing NFA based payload scanning implementations because of the inefficiency in regular expression matching. Deterministic finite automata (DFAs) for certain regular expression types suffer from state blow up limiting their practical implementation. To solve the problem of state blow up we propose the following two part process.
1. In the first step the regular expressions are divided into subexpressions and these subexpressions of all regular expressions are then categorized into "sub expression modules" depending on their type. These modules are essentially scanners that run on the input and return the positions where the subparts occur in the input. Every regular expression is composed of one of constant strings, closures, length restrictions , a class of characters , a query or a combination of these. Thus the regular expression can be broken down into these components and all components of same type can be combined into a single module. For example all constant strings occurring in all regular expressions will be in one module with a single DFA scanning for constant strings in the input. Many of these components especially closures and length restriction are common to several regular expressions and thus running a single DFA for all of them is beneficial.

2. In the second step we construct *pattern modules* for each regular expression, which collect events consisting of subexpression matches along with matching positions (each such pair termed as an event) generated by the sub expression modules in the first step and string them together to find a match for the actual regular expression. Each such pattern module is modeled as a *timed automaton*. Each subexpression knows which pattern messages it needs to send messages to

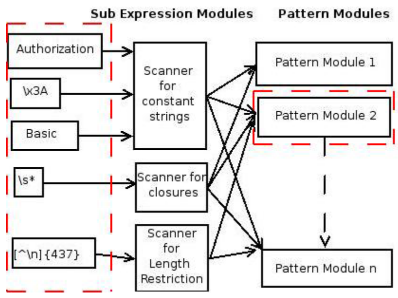


Fig. 1. Fragmenting RE 'Authorization\s*\x3A\s*Basic\s*[^\n]{437}'

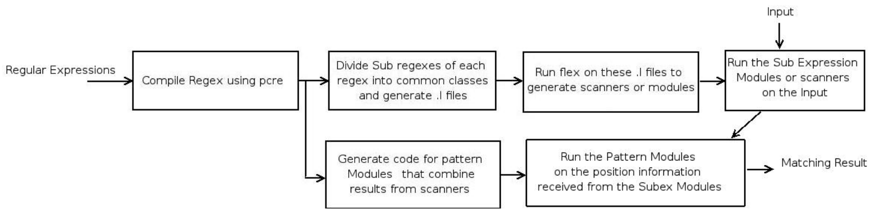

Regular Expressions
Compile Regex using pcre
Divide Sub regexes of each regex into common classes and generate .l files
Run flex on these .l files to generate scanners or modules
Input
Run the Sub Expression Modules or scanners on the Input
Generate code for pattern Modules that combine results from scanners
Run the Pattern Modules on the position information received from the Subex Modules
Matching Result

Fig. 2. Block Diagram of our solution

so each timed automaton gets only events of potential interest to it. In a timed automaton each symbol represents a timed event, namely a sub expression of the regular expression along with its "time" or position in the input stream, transitions being based on such events. The timed automaton is triggered when the first subexpression of the regular expression is found in a position buffer maintained for logging these events.The pattern modules are independent of each other and hence can be run in parallel in a hardware setup or on a parallel machine. Currently there is one pattern module per regular expression. As the number of regular expressions is large, in future, we plan to combine multiple regular expressions into single pattern modules based on the subexpressions they share. As a trial setup we have run the implementation for a selected set of 32 regular expressions from more than four thousand such rules that make up SNORT PCRE rulesets. The flex generated scanners are in C and the rest has been implemented in C++. The expressions chosen cover almost all types of regular expressions. The number of states for the traditional DFA for each of the 32 regular expressions ranges from 30 to almost 2^{400} where 400 is the length restriction in one of the expressions. The total number of NFA states for these regular expressions is 11924. In our solution, the total number of states for all DFAs in the sub expression modules is just equal to 533. The number of states in the pattern modules is equal to total sub expressions which is 300. This solution appears scalable though proper parallel communication protocols and a fast simulation of a timed automaton are critical for efficient functioning. We plan to do the simulation of the parallel implementation on the Blue Gene-L machine and later will examine hardware implementations.

System Call API Obfuscation (Extended Abstract)

Abhinav Srivastava[1], Andrea Lanzi[1,2], and Jonathon Giffin[1]

[1]School of Computer Science, Georgia Institute of Technology, USA
[2]Dipartimento di Informatica e Comunicazione, Università degli Studi di Milano, Italy
{abhinav,giffin}@cc.gatech.edu, andrew@security.dico.unimi.it

Abstract. We claim that attacks can evade the comprehension of security tools that rely on knowledge of standard system call interfaces to reason about process execution behavior. Our attack, called *Illusion*, will invoke privileged operations in a Windows or Linux kernel at the request of user-level processes without requiring those processes to call the actual system calls corresponding to the operations. The Illusion interface will hide system operations from user-, kernel-, and hypervisor-level monitors mediating the conventional system-call interface. Illusion will alter neither static kernel code nor read-only dispatch tables, remaining elusive from tools protecting kernel memory.

1 Illusion Attack

Honeypots and other utilities designed to audit, understand, classify, and detect malware and software attacks often monitor process' behavior at the system call interface as part of their approach. Past research has developed a widespread collection of system-call based systems operating at user or kernel level [1,5,2,4] and at hypervisor level [3]. Employing reference monitors at the system call interface makes intuitive sense: absent flaws in the operating system (OS) kernel, it a non-bypassable interface, so malicious code intending to unsafely alter the system will reveal its behavior through the series of system calls that it invokes.

Current malware increasingly makes use of kernel modules or drivers that help the user-level process perform malicious activities by hiding the process' side effects. For example, the rootkits *adore* and *knark* hide processes, network connections, and malicious files by illegitimately redirecting interrupt or system call handling into their kernel modules. Redirection can alter the semantic meaning of a system call—a problem for any system that monitors system calls to understand the behavior of malware. Jiang and Wang address this class of attack:

> Syscall remapping requires the modification of either the interrupt descriptor table (IDT) or the system call handler routine... [3]

Systems like that of Jiang and Wang assume that protections against illegitimate alteration of the IDT or system call handler will force malicious software to always follow the standard system-call interface when requesting service from the kernel.

Unfortunately, this assumption does not hold true. Malicious code can obfuscate the Windows or Linux system call interface using only legitimate functionality commonly used by kernel modules and drivers. Our *Illusion* attack will allow malicious processes to invoke privileged kernel operations without requiring the malware to call the actual system calls corresponding to those operations. In contrast to prior attacks of the sort considered by Jiang and Wang, Illusion will alter neither static kernel code nor read-only dispatch tables such as the IAT or system call descriptor table (SSDT). During the execution of malware augmented with the Illusion attack, an existing system-call analyzer will see a series of system calls different that those actually executed by the malware.

The Illusion attack is possible because a number of system calls allow *legitimate* dispatch into code contained in a kernel module or driver, and this permits an attacker to alter their semantics. Consider `ioctl`: this system call takes an arbitrary, uninterpreted memory buffer as an argument and passes that argument to a function in a kernel module that has registered itself as the handler for a special file. Benign kernel modules legitimately register handler functions for such files; a malicious module performing the same registration exhibits no behaviors different than the benign code. However, a call to `ioctl` will be directed into the malicious module's code together with the buffer passed to `ioctl` as an argument. In user-space, we marshal a malware's actual system call request into this buffer, and we then use the kernel module to unmarshal the request and invoke the appropriate kernel system call handler function. With this interface illusion in place, the kernel still executes the same operations that the malware instance would have executed without the obfuscation. However, system call monitoring utilities would observe a sequence of `ioctl` requests and would not realize that malicious operations had occurred.

References

1. Forrest, S., Hofmeyr, S.A., Somayaji, A., Longstaff, T.A.: A sense of self for UNIX processes. In: IEEE Symposium on Security and Privacy, Oakland, CA (May 1996)
2. Giffin, J.T., Jha, S., Miller, B.P.: Efficient context-sensitive intrusion detection. In: Network and Distributed System Security Symposium (NDSS), San Diego, CA (February 2004)
3. Jiang, X., Wang, X.: Out-of-the-box monitoring of VM-based high-interaction honeypots. In: Kruegel, C., Lippmann, R., Clark, A. (eds.) RAID 2007. LNCS, vol. 4637, pp. 198–218. Springer, Heidelberg (2007)
4. Krohn, M., Yip, A., Brodsky, M., Cliffer, N., Kaashoek, M.F., Kohler, E., Morris, R.: Information flow control for standard OS abstractions. In: Symposium on Operating System Principles (SOSP), Stevenson, WA (October 2007)
5. Sekar, R., Bendre, M., Dhurjati, D., Bollineni, P.: A fast automaton-based method for detecting anomalous program behaviors. In: IEEE Symposium on Security and Privacy, Oakland, CA (May 2001)

Author Index